# 論文 / 著書情報
# Article / Book Information

| | |
|---|---|
| 題目(和文) | オーバーサブスクライビングスケジューリング： HPCシステムにおける多様化するワークロードの効率性と応答性の両立 |
| Title(English) | Oversubscribing Scheduling: Balancing Efficiency and Responsiveness in HPC Systems with Diverse Workloads |
| 著者(和文) | 南将平 |
| Author(English) | Shohei Minami |
| 出典(和文) | 学位:博士(理学),<br>学位授与機関:東京科学大学,<br>報告番号:甲第233号,<br>授与年月日:2025年3月26日,<br>学位の種別:課程博士,<br>審査員:遠藤 敏夫,増原 英彦,坂本 龍一,安永 憲司,脇田 建 |
| Citation(English) | Degree:Doctor (Science),<br>Conferring organization: Institute of Science Tokyo,<br>Report number:甲第233号,<br>Conferred date:2025/3/26,<br>Degree Type:Course doctor,<br>Examiner:,,,, |
| 学位種別(和文) | 博士論文 |
| Type(English) | Doctoral Thesis |

# Oversubscribing Scheduling: Balancing Efficiency and Responsiveness in HPC Systems with Diverse Workloads

Shohei Minami

School of Computing Department of Mathematical and Computing Science

Institute of Science Tokyo

A thesis submitted for the degree of

*Doctor*

March 2025

# Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor Toshio Endo, for his unwavering guidance, encouragement, and support throughout my doctoral journey. Over the past six years, his regular research meetings have provided me with invaluable feedback and direction, helping me navigate complex challenges and refine my ideas. I am especially thankful for the well-equipped research environment and facilities he ensured, which created the optimal conditions for me to focus fully on my studies. His dedication and mentorship have been a cornerstone of my academic achievements, and I am truly fortunate to have had his support.

I am also profoundly grateful to Professor Akihiro Nomura for his collaboration and the generous sharing of his extensive expertise. His deep insights and specialized knowledge have been a constant source of inspiration and learning. Over the course of our six years of joint research discussions, he has consistently offered thoughtful advice and constructive feedback that have significantly enhanced the quality and scope of this dissertation. I deeply value the opportunity to have worked alongside him and have gained a great deal from his mentorship.

To my family, I owe the deepest appreciation for their unwavering belief in me. Their support, understanding, and encouragement have been the foundation that kept me motivated through the ups and downs of this academic journey. Their patience and love have been a constant source of strength, reminding me of the importance of perseverance and dedication.

I would also like to extend my heartfelt thanks to my employer, Prometech Software, Inc., for their generous support and understanding throughout my studies. They provided me with the flexibility to balance my professional responsibilities with my academic goals, as well as the resources needed to make this dual commitment possible. Their encouragement and recognition of the value of lifelong learning have been truly meaningful to me.

This dissertation represents the culmination of years of effort, and it would not have been possible without the contributions of all these individuals. I am sincerely grateful to each of them for their invaluable support, guidance, and encouragement, which have made this achievement a reality.

# Abstract

High-Performance Computing (HPC) systems face significant challenges in managing increasingly diverse workloads, particularly with the growing prevalence of AI/ML applications alongside traditional scientific computing tasks. While these systems have historically excelled at handling batch-oriented workloads, they now struggle to efficiently support interactive computing needs essential for AI/ML development while maintaining system efficiency. This dissertation presents a comprehensive validation of oversubscribing (OSub) as a solution to these emerging challenges in HPC environments, demonstrating that it can simultaneously provide users with immediate resource access and reduced waiting times, while maintaining acceptable system throughput and enabling simplified management for operators.

Our research makes four major contributions. First, we provide systematic quantification of performance impact under OSub through rigorous analysis. Through detailed performance evaluation, we show that parallel application performance degradation remains manageable when gang scheduling is employed, while hardware counter analysis enables prediction of sequential application performance under resource sharing. Second, through extensive simulation studies using real production workload traces, we demonstrate that waiting time reduction can be achieved while maintaining acceptable system-wide performance. Third, we establish OSub's effectiveness in environments dominated by interactive workloads, demonstrating stable performance for both batch and interactive jobs without parameter tuning even as interactive jobs comprise up to 76% of the total workload. This robustness in interactive-job-dominant scenarios represents a significant advance over traditional multiple-queue approaches, which require complex optimization as interactive job ratios increase. Fourth, we validate our findings through implementation and experimentation on a physical HPC system, demonstrating OSub's practicality through actual deployment and confirming the performance benefits and operational improvements observed in simulation studies.

These findings establish OSub as a practical solution for modern HPC environments. Our research provides crucial insights for HPC centers adapting to support the growing demands of AI/ML workflows alongside traditional scientific computing tasks, demonstrating that efficient resource sharing through OSub can effectively balance immediate resource access needs with overall system efficiency.

# Contents

# List of Figures

# 1 Introduction

## Contents



This chapter first presents background and challenges of High-Performance Computing (HPC) systems which is a target of this dissertation. Then, it introduces oversubscribing - the core concept of this research - and its limited adoption in HPC environments. Following this, the chapter presents the contributions of this doctoral dissertation, which provide new insights into resource management strategies for modern HPC environments. The chapter concludes with an outline of the overall research structure.

## 1.1 Background of High-Performance Computing (HPC) systems

### 1.1.1 Important role in scientific simulations

First and foremost, computer simulations on HPC systems enable scientific discoveries across various fields, from quantum physics to molecular biology [1], [2]. These simulations allow researchers to explore phenomena that are too dangerous, expensive, or physically impossible to study through traditional experiments, leading to breakthrough discoveries and advancing our understanding of the universe. As such, HPC is a crucial tool for science discoveries.

Beyond pure scientific research, HPC significantly accelerates product design and development processes across industries. Engineers can perform complex simulations for fluid dynamics, structural analysis, and thermal behavior, reducing the need for physical prototypes [3]–[5]. This acceleration shortens development cycles, reduces costs, and allows the exploration of innovative designs that would be impractical with traditional methods.

In addition, HPC systems form the backbone of critical social infrastructure through weather forecasting and disaster prediction [6], [7]. Modern weather forecasting requires processing vast amounts of sensor data and running sophisticated atmospheric models. Similarly, disaster prediction systems utilize HPC resources to model earthquakes, tsunamis, and other natural phenomena, significantly contributing to public safety and emergency preparedness.

In this way, HPC system plays an important role in leading a prosperous society.

### 1.1.2 Significance of HPC systems in AI/ML fields

In recent years, generative artificial intelligence has attracted attention. Training large language models is known to require massive computational resources. Figure 1.1 shows the number of GPUs held by artificial intelligence (AI)/machine learning (ML) organizations [8]. Modern language models with hundreds of billions of parameters require extended training periods on multiple GPU clusters. However, except for organizations shown in Figure 1.1, many individual organizations cannot maintain such a amount of computing resources, which makes shared HPC infrastructure essential for many researchers/developers. Figure 1.2 shows the recent trend in the number of GPU-equipped systems in TOP 500 [9], [10]. Obviously, many systems are moving to support GPU resources.

AI workloads have become a primary application of HPC systems, which makes several effects for HPC usages. First, mentioned earlier, training large

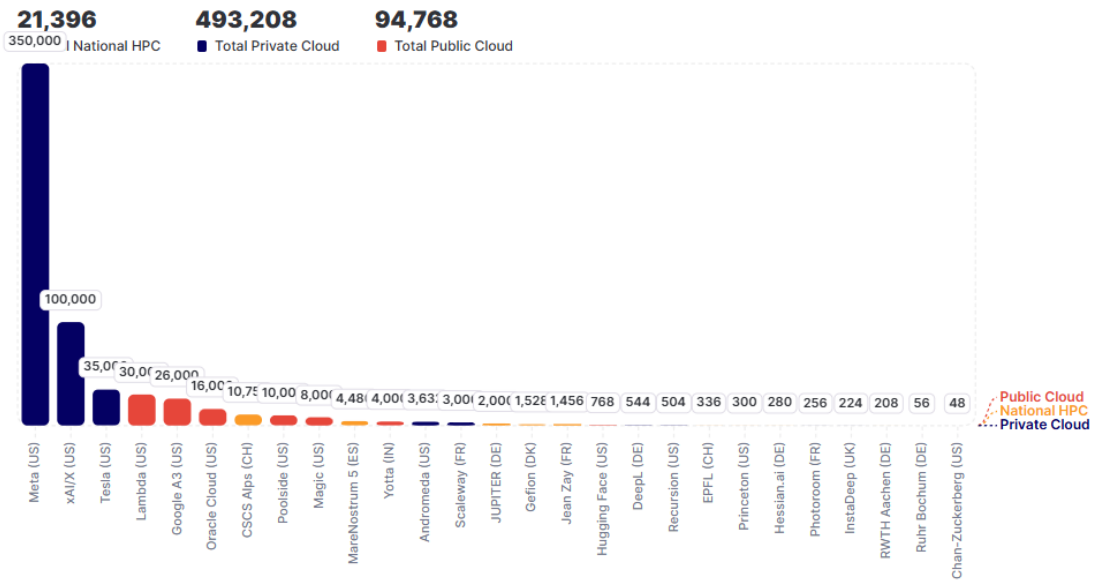



**Figure 1.1:** The number of NVIDIA H100 units held by institutions worldwide, as publicly disclosed [8]. Please note that this does not necessarily include all institutions. Rather than national HPCs, the ranking of private companies stands out.

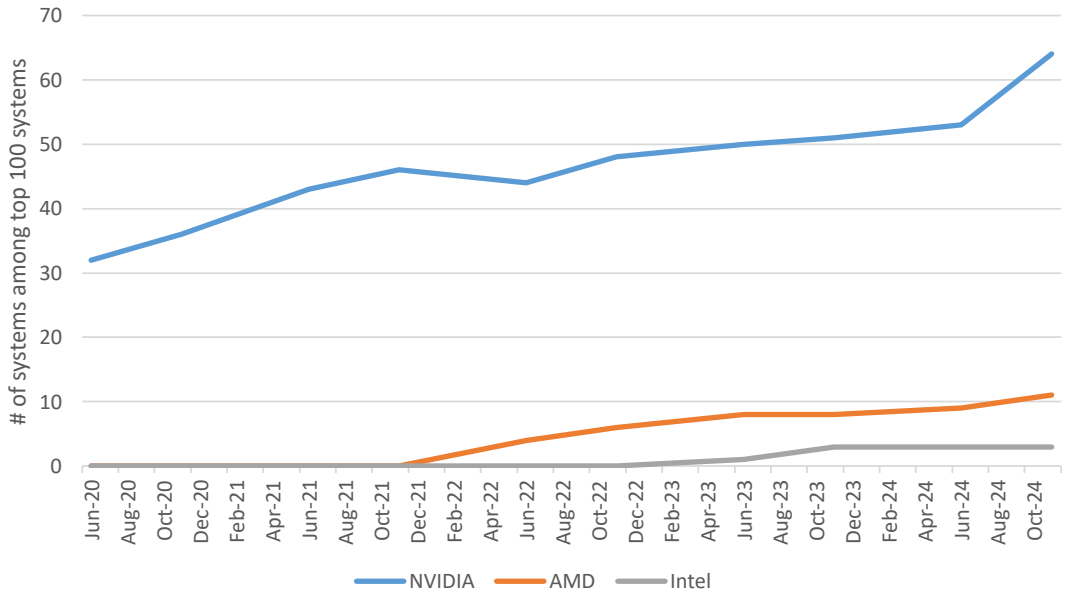



**Figure 1.2:** Number of GPU equipped systems in TOP 100. TOP100 refers to the top 100 sites from the TOP500 list [9], [10]. Each legend represents the manufacturer of the installed GPUs. While NVIDIA is the dominant player, the adoption of AMD and Intel GPUs has also been increasing in recent years.

models needs large amount of computing resources. Second, in addition to these training workloads, HPC systems should support AI inference services that require rapid response times and efficient resource management. These services must handle varying workload intensities while maintaining consistent performance. Furthermore, HPC resources have become indispensable in AI-focused research and development [11]. Researchers utilize these systems for model prototyping, hyperparameter optimization, and experimental validation. The interactive nature of AI development workflows demands flexible resource allocation and quick turnaround times, transforming traditional HPC usage patterns.

This trend toward AI/ML workloads is fundamentally changing the job composition in HPC systems. While traditional HPC workloads primarily consisted of batch jobs, AI/ML workflows introduce a significant interactive component through inference services, model development, and experimentation. This dramatic shift in workload composition challenges traditional resource management approaches, which were primarily developed and validated in batch-job-dominant environments. The emergence of these interactive-job-dominant scenarios represents a significant change in HPC system usage patterns that demands careful investigation, yet remains largely unexplored in existing research.

## 1.2 Challenges of HPC systems

This section mentions the several challenges of HPC systems from various viewpoints. The challenges are categorized into those from the user side and those from the operator side.

### 1.2.1 Challenges from users side

The challenges from the users' perspective are primarily related to productivity losses stemming from the shared nature of these systems, system usage complexities, and environmental constraints.

#### Due to shared nature of HPC systems

The shared nature of HPC systems introduces significant productivity challenges. If system is busy, users often experience substantial waiting times before their jobs can execute, which disrupts development cycles and impedes iterative processes. This waiting time is particularly problematic for interactive workloads such as debugging sessions or AI model development, where quick feedback is essential [12]. In addition, the uncertainty in resource availability due to waiting time

makes it difficult for users to predict turned around time accurately, complicating project planning and scheduling.

### Due to system usage complexities of HPC systems

System usage itself introduces significant barriers that distinguish HPC systems from regular computing environments. Unlike personal computers where users can directly execute programs, HPC systems require users to submit their work as *jobs* through specialized job scheduling systems. This fundamental difference in execution model requires users to adapt their workflows and invest considerable time in learning system-specific concepts and commands. The learning curve is particularly steep for new users, including domain experts who may be highly skilled in their field but unfamiliar with HPC environments. This educational overhead represents a substantial investment for both users and organizations [12].

These system usage complexities, when combined with the waiting time issues discussed above, create a compound challenge that severely impacts productivity. For instance, while troubleshooting on a personal computer might involve rapid iterations of code modification and execution, the same process in an HPC environment requires each iteration to be submitted as a job, potentially facing significant queue times. Error handling becomes particularly challenging - when jobs fail, users must not only diagnose problems with limited direct feedback but also resubmit their jobs and face the waiting time again. This combination of system complexity and waiting time makes even simple debugging tasks time-consuming and severely hampers efficient trial-and-error processes [13]. This challenge becomes particularly acute in environments where interactive jobs or on-demand execution options are limited, further impeding the development and debugging cycles that are essential for productive scientific computing.

### Due to environmental constraints

Environmental constraints in HPC systems primarily manifest in challenges related to execution consistency and performance portability. When applications need to run across different HPC environments, users might encounter variations not only in performance characteristics but also in execution results due to differences in system architectures, compiler implementations, and numerical libraries. These inconsistencies can make it difficult to validate scientific results and debug applications across different systems.

Beyond result consistency, performance portability presents another significant challenge. Applications often exhibit different performance characteristics between

systems due to variations in hardware architectures, interconnect technologies, and system software stacks. This variation makes it difficult to predict application performance when moving from one system to another. Users frequently need to reinvest time in system-specific performance analysis and optimization when transitioning between different HPC environments, as optimizations developed for one system may not translate effectively to others [14]. The execution environment dependencies can create unexpected behavior or performance degradation, even when running the same application with identical input parameters across different systems.

### 1.2.2 Challenges from operators side

From the system operators' perspective, there are three major categories of challenges: resource and power efficiency, adaptation to increasing workload diversity, and security requirement compatibility.

#### System efficiency and the productivity trade-off

Resource and power efficiency represents a primary concern in managing modern computing environments [15], [16]. HPC systems require massive investments in hardware, infrastructure, and operational costs, making efficient operation critical for operators. This efficiency broadly encompasses two major aspects: system utilization and power efficiency.

High system utilization is essential for maximizing return on infrastructure investment, requiring operators to process workloads efficiently while minimizing idle resources. Power efficiency, driven by carbon neutrality goals and environmental concerns, demands optimal workload processing while minimizing energy consumption. Both aspects significantly impact operational costs - underutilized systems waste investment resources, while poor power efficiency leads to excessive energy costs.

However, while system efficiency is crucial from an operational perspective, operators must also consider user productivity. Traditionally, these objectives have been viewed as fundamentally conflicting - high system efficiency typically requires jobs to wait for optimal scheduling conditions, while maintaining user productivity demands immediate resource access. This perceived trade-off between system efficiency and user responsiveness has significantly influenced HPC system design and operational policies. As systems continue to grow in scale and complexity, balancing these competing demands becomes increasingly critical, particularly when systems experience varying load patterns or when resources remain idle due to scheduling constraints.

### Acceptance of diverse workloads

The increasing diversification of workloads presents another significant challenge. Modern HPC systems must accommodate both traditional batch processing and an increasing volume of interactive workloads, particularly from AI/ML applications. As discussed in the productivity challenges above, traditional scheduling approaches optimized for batch processing are becoming inadequate, causing significant waiting times that impede research and development cycles. This shift in usage patterns requires operators to develop more sophisticated resource management strategies [17]. While these strategies are necessary to effectively handle heterogeneous workloads, they often lead to increased operational complexity and administrative overhead. This complexity highlights the need for management approaches that minimize operational burden while maintaining effective workload handling. On the other hand, different systems may require different solutions based on their specific workload characteristics and user requirements, necessitating comprehensive monitoring and analysis of usage patterns, resource utilization, and job characteristics [12]. These monitoring systems are essential for understanding the impact of different workload types on system performance and user productivity, enabling operators to make informed decisions about resource allocation and policy adjustments.

### Security and availability

Security requirements introduce the third major challenge in HPC system operation/administration. While basic security measures such as process isolation and data protection are standard features in HPC systems, the authentication and access control requirements can affect system usability and thereby impact overall productivity [18].

For example, strict authentication procedures and access controls, while necessary for security, can complicate the already challenging job submission process discussed earlier. Users may need to manage multiple authentication credentials, navigate through security protocols when accessing different system components, or deal with session timeouts during long-running interactive sessions. These security measures, intended to protect system integrity, often create additional barriers that can interrupt development workflows, particularly in interactive computing scenarios where immediate system access is crucial.

The challenge for operators lies in maintaining an appropriate balance between security requirements and productivity. Overly strict security controls can exacerbate the challenges discussed in previous sections, while also potentially

deterring users from fully utilizing the system's capabilities. This balance becomes increasingly critical as HPC systems support more diverse workloads, particularly interactive AI/ML development workflows that require frequent system access and rapid response times.

## 1.3 Oversubscribing as a solution option

The challenges faced by modern HPC systems - from managing diverse workloads to maintaining resource efficiency while improving user productivity - call for innovative resource management approaches. Oversubscribing (also known as oversubscription), which allows multiple computational tasks to share the same physical resources through time-division multiplexing, is a well-established concept in other computing domains [19], [20]. This approach, which enables logical resource allocations to exceed the available physical resources, emerges as a promising solution that could address multiple aspects of these challenges through a single, unified mechanism. By enabling controlled sharing of computational resources among multiple jobs, this approach offers potential benefits in resource utilization, user responsiveness, and operational simplicity.

While HPC systems face various challenges, oversubscribing primarily targets two of the most critical issues: user productivity degradation, which is the most serious concern from users' perspective, and accommodation of diverse workloads, which is the fundamental challenge for operators.

### 1.3.1 Potential for addressing HPC challenges

Oversubscribing presents a potential unified solution to these emerging challenges while maintaining system efficiency.

#### Adaptation to diversifying workloads

As mentioned in Section 1.2.2, modern HPC systems must accommodate an increasingly diverse range of workloads, particularly the coexistence of traditional batch jobs and interactive sessions. OSub's ability to dynamically share resources makes it particularly well-suited to these hybrid workload patterns, offering a unified mechanism for managing all workloads rather than maintaining separate resource pools or complex scheduling policies for different job categories [21].

#### Addressing waiting time issues

The reduction of waiting times represents another crucial advantage of OSub in HPC environments. Traditional scheduling approaches often result in significant delays, particularly for interactive workloads, as jobs must wait for sufficient resources to become available [22]. OSub addresses these challenges by enabling immediate execution through resource sharing [21], particularly valuable for development sessions, debugging, and AI model experimentation where immediate response is crucial for maintaining productive workflows.

### 1.3.2 Limited adoption in HPC systems

Despite these potential benefits, OSub adoption in HPC environments has been limited compared to other computing domains such as cloud computing. This limited adoption stems from several key concerns:

#### Performance variation concerns

Performance variability presents a significant barrier. HPC applications are often highly optimized for maximum performance, and any interference between jobs sharing resources could lead to unpredictable execution times and degraded performance [23]. This variability is particularly concerning for large-scale parallel applications where performance degradation in one process can cascade through the entire application.

#### Fairness issues

Fairness becomes a critical concern under OSub. In traditional HPC environments, users requesting the same amount of computational resources receive equivalent computing capability. However, under OSub, multiple users making identical resource requests might achieve significantly different amounts of actual computation due to varying interference from co-located jobs [24]. This inequity challenges the fundamental principle of fairness in HPC systems.

### 1.3.3 Research objectives

This research aims to establish the practical viability of oversubscribing in HPC environments through systematic evaluation. The primary objective is to demonstrate that oversubscribing can effectively address the emerging challenges in modern HPC systems - particularly the need to support diverse workloads including

interactive jobs while maintaining system efficiency - without requiring complex system configurations or compromising essential HPC performance characteristics. This objective specifically targets the growing demands of AI/ML workflows in HPC environments, where traditional resource management approaches have proven inadequate in balancing immediate resource access needs with overall system efficiency.

## 1.4 Contributions of this research

This dissertation provides the first systematic validation of oversubscribing effectiveness in HPC environments through comprehensive performance analysis and real system implementation. Our research combines theoretical insights with practical validation, establishing both the performance characteristics and operational viability of oversubscribing while paying particular attention to interactive job requirements.

### 1.4.1 Key contributions of this dissertation

**Demonstration of OSub's capability to simultaneously satisfy both user productivity and operator efficiency**

The primary contribution of this research lies in demonstrating that oversubscribing can effectively benefit both users and system operators of HPC systems while maintaining a balance between their potentially competing interests.

For users, our research demonstrates significant improvements in productivity through:

- Elimination of waiting times through immediate job execution
- Predictable performance trade-offs that enable effective planning

For system operators, we show substantial operational advantages including:

- Acceptable performance degradation even while prioritizing users' demands for immediate execution
- Simplified system management through configuration-free operation, eliminating complex parameter tuning and static resource partitioning

Crucially, we demonstrate through systematic evaluation that these benefits for both users and operators can be achieved simultaneously in a balanced way. This finding demonstrates that user responsiveness and system efficiency can coexist in shared computing environments, contrary to traditional assumptions.

**Empirical assessment of OSub effects with data from production environments**

This research provides the first comprehensive assessment of OSub behavior with data from real HPC environments. While previous research relied primarily on theoretical analysis or limited test cases, our work demonstrates OSub's effectiveness through multiple dimensions of empirical evidence. We validate OSub's viability using real production workload traces and actual interactive job characteristics, providing concrete evidence of its practicality in operational settings. Furthermore, our physical system implementation and experimentation provide crucial validation beyond simulation studies, establishing OSub as a practical solution for production HPC environments.

**Validation in interactive-job-dominant environments**

Traditional HPC research has primarily focused on batch-job-oriented environments. Our research demonstrates for the first time that OSub can provide robust operation without parameter tuning even in environments where interactive jobs constitute the majority of workloads. This finding has significant implications for modern HPC system design, particularly as AI/ML development workflows become increasingly prevalent. Our analysis reveals that OSub maintains consistent performance benefits across various interactive workload ratios, demonstrating its broad applicability in diverse HPC environments.

### 1.4.2 Key research achievements

This research provides four major contributions that advance our understanding and implementation of oversubscribing in HPC environments:

**Quantitative evaluation of application performance impact**

Our first major contribution is a comprehensive quantitative evaluation of how oversubscribing affects application performance in HPC environments. Through systematic performance analyses using the NAS Parallel Benchmark suite [25], [26], we demonstrated:

- Fundamental performance characteristics under OSub:
  - Parallel applications showed performance impacts strongly influenced by synchronization requirements and communication patterns

  - Sequential applications exhibited varying interference patterns depending on job combinations

- For parallel applications, detailed analysis revealed the relationship between communication patterns and performance degradation, contributing to our understanding of OSub performance characteristics

- For sequential applications, detailed performance counter analysis revealed memory access and cache utilization as key factors in interference patterns, enabling development of an accurate performance prediction model

This comprehensive understanding of both parallel and sequential application behavior enables more informed decisions about resource sharing strategies in mixed workload environments.

**System-wide effects validation**

The second key achievement is a thorough validation of system-wide effects through large-scale simulation studies using real workload traces. Our evaluation demonstrates the practical benefits of oversubscribing at the system level through:

- Quantitative analysis of waiting time reduction across different workload patterns

- Comprehensive evaluation of system utilization changes under various configurations

- Assessment of the impact on both batch and interactive workloads

Notably, our results show that system-wide performance degradation remains within acceptable limits even under aggressive oversubscribing configurations, challenging previous assumptions about the impracticality of oversubscribing in HPC environments.

**Accurate evaluation considering interactive job characteristics**

Our third major contribution is a novel evaluation methodology that accurately accounts for the unique characteristics of interactive jobs. We developed a comprehensive model of interactive job behavior based on detailed analysis of real usage patterns, including:

- CPU utilization fluctuations over time

- Patterns of active and idle periods
- Resource usage variations during interactive sessions

By incorporating these characteristics into our evaluation framework, we provide more realistic assessments of oversubscribing effectiveness in mixed workload environments. Particularly significant is our analysis of systems with high proportions of interactive jobs, where we demonstrated that:

- Traditional multiple-queue approaches require complex parameter optimization as the ratio of interactive jobs increases
- Oversubscribing maintains stable performance for both batch and interactive jobs without requiring parameter adjustments, even when interactive jobs comprise up to 76% of the total workload
- System performance remains predictable and manageable even under interactive-job-dominant scenarios

These results are particularly valuable for modern HPC centers adapting to support the growing demands of AI/ML development workflows.

**Physical system validation of OSub effectiveness**

The fourth major achievement is a thorough validation of OSub through physical system implementation and experimentation. This validation provides crucial empirical evidence that bridges the gap between simulation studies and practical deployment through:

- Implementation of a complete OSub system incorporating production job scheduler integration, gang scheduling support, and reproducible workload generation
- Systematic evaluation using controlled experiments that demonstrated consistent responsiveness for interactive jobs and manageable performance impact on batch jobs
- Validation of key findings from simulation studies including predicted performance characteristics and waiting time reduction benefits

This physical system validation provides essential empirical evidence for OSub's practical viability in production HPC environments, complementing the insights gained from simulation studies and theoretical analysis. The successful implementation and operation of OSub in a real system environment represents a crucial step toward broader adoption in production HPC facilities.

## 1.5 Organization of this dissertation

This dissertation is organized into eight chapters that systematically examine the implementation and effectiveness of oversubscribing in HPC environments:

Chapter 1 introduces the background and motivation for this research, describing the evolving challenges in HPC systems and presenting oversubscribing as a potential solution. It outlines the research objectives and summarizes the key contributions.

Chapter 2 provides essential background knowledge about HPC systems and job scheduling. It explains the fundamental concepts of supercomputing systems, job schedulers, different types of jobs, and HPC applications. This chapter establishes the technical foundation necessary for understanding the subsequent analysis.

Chapter 3 examines existing approaches to managing HPC workloads and their limitations, then presents oversubscribing as an alternative solution. After analyzing conventional resource management strategies, the chapter introduces the fundamental mechanisms of oversubscribing and conducts a systematic analysis of both its potential benefits and implementation challenges in HPC environments.

Chapter 4 presents a detailed analysis of oversubscribing's impact on application performance. Through systematic evaluation using benchmarks and performance profiling, it quantifies performance degradation patterns and develops predictive models for both parallel and sequential applications.

Chapter 5 examines the system-level effects of oversubscribing through large-scale simulation studies. Using real workload traces, it evaluates the impact on waiting times, system utilization, and overall efficiency across various configurations and workload patterns.

Chapter 6 provides a rigorous evaluation of oversubscribing in environments with significant interactive workloads. It introduces novel methodologies for modeling interactive job behavior and demonstrates the effectiveness of oversubscribing in handling diverse workload combinations.

Chapter 7 validates the findings from previous chapters through implementation and evaluation on a physical HPC system. This chapter bridges the gap between simulation studies and practical deployment by demonstrating OSub's effectiveness in a real operational environment, providing crucial empirical evidence for its viability in production systems.

Chapter 8 concludes the dissertation by summarizing the key findings and contributions. It discusses the implications of this research for HPC system design and operation, and outlines future research directions and remaining challenges.

# 2

# Background: High-Performance Computing Systems and Resource Management

## Contents



This chapter presents essential background knowledge about High-Performance Computing (HPC) systems. We first introduce supercomputer systems as the fundamental computing infrastructure. Then, we examine job schedulers - the

software systems that manage resource allocation and job execution. Following this system-level overview, we explore jobs, the fundamental units managed by the schedulers. The chapter concludes with an analysis of HPC applications that execute within these jobs, including their characteristics and performance considerations.

## 2.1 Supercomputer systems

This section examines supercomputer systems - large-scale computing facilities designed for processing complex scientific calculations. Modern supercomputers represent significant technological and operational investments, combining thousands of computing nodes with sophisticated networking and management systems. Understanding their fundamental architecture and operational characteristics is essential for appreciating the resource management challenges addressed in this research.

### 2.1.1 System architecture

#### Large-scale integration of server-grade computers

Modern supercomputer systems represent massive collections of high-performance computing nodes, each equipped with server-grade components optimized for scientific computing workloads. These nodes typically feature several key characteristics that distinguish them from conventional computing systems:

First, they incorporate high-performance CPUs specifically designed for computational workloads. These processors often include advanced features such as extensive vector processing capabilities, large cache hierarchies, and sophisticated out-of-order execution mechanisms to maximize computational throughput. The trend toward multi-core architectures has enabled significant parallel processing capabilities [27], [28].

Second, these systems implement large-capacity memory configurations to support data-intensive computations. The memory subsystems are typically optimized for both bandwidth and capacity, enabling efficient handling of the massive datasets common in scientific computing applications. Modern systems often employ sophisticated memory hierarchies, including high-bandwidth memory (HBM) and traditional DRAM, to balance performance and capacity requirements [29].

Third, the integration of computational accelerators, particularly GPUs, has become increasingly common in modern supercomputing systems. As shown in Figure 1.2, the number of GPU-equipped systems in the TOP500 has grown significantly in recent years. These accelerators provide exceptional performance for specific computational patterns, particularly those found in AI/ML workloads and certain scientific simulations [30], [31].

#### Interconnection networks

The interconnection network forms a crucial component of supercomputer architecture, enabling efficient communication and coordination between computing nodes [32], [33]. These networks employ specialized technologies to achieve several critical characteristics:

- High bandwidth capabilities to support rapid data transfer between nodes, essential for parallel applications that require significant data movement
- Low latency communication to minimize delays in inter-node coordination, particularly crucial for tightly coupled parallel applications
- Advanced topology designs that maintain network performance as system size scales, often implementing sophisticated routing algorithms and redundant paths

The quality of these interconnection networks significantly influences the system's ability to maintain performance efficiency as applications scale across multiple nodes. Modern interconnects often incorporate advanced features such as remote direct memory access (RDMA) and hardware-level collective communication support to optimize parallel application performance.

### 2.1.2 System performance metrics

Performance evaluation of supercomputer systems relies on standardized benchmarks and metrics that enable objective comparisons between different systems. The TOP500 list [9], [10], updated biannually, represents the primary ranking system for global supercomputing capabilities. This ranking primarily uses the High-Performance LINPACK (HPL) benchmark [34], which solves a dense system of linear equations, as its key performance metric.

Beyond raw computational performance, modern evaluation increasingly considers power efficiency, as reflected in the Green500 ranking [35]. This metric, measuring performance per watt, has gained importance as systems grow larger and energy costs become a more significant factor in total operating expenses.

### 2.1.3 Shared facility operation

Supercomputer systems typically operate as shared facilities, serving diverse user communities across multiple scientific domains and application areas. This shared nature introduces several key operational considerations:

- Multiple users must be able to access system resources simultaneously, requiring sophisticated resource management mechanisms
- Various types of computational tasks, from large-scale simulations to interactive development work, need to be accommodated efficiently
- System resources must be allocated fairly while maintaining overall system efficiency

The complexity of managing these shared resources necessitates sophisticated job scheduling systems, which handle resource allocation, job prioritization, and system monitoring [36]–[38]. These schedulers play a crucial role in maintaining efficient operation while ensuring fair access across the user community.

## 2.2 Job schedulers

Job schedulers are essential software systems that manage resource allocation and job execution in HPC environments [36]–[38]. They serve as intermediaries between users and computing resources, ensuring efficient system utilization while maintaining fairness and security. This section examines the fundamental purposes and roles of job schedulers in modern HPC operations.

### 2.2.1 Purposes and roles

#### Optimization of system performance and resource utilization

The primary purpose of job schedulers is to maximize both system performance and resource utilization of expensive HPC infrastructure through systematic resource management. These systems must balance multiple competing demands:

- Optimal allocation of computational resources across multiple simultaneous jobs
- Minimization of resource fragmentation that could lead to underutilization

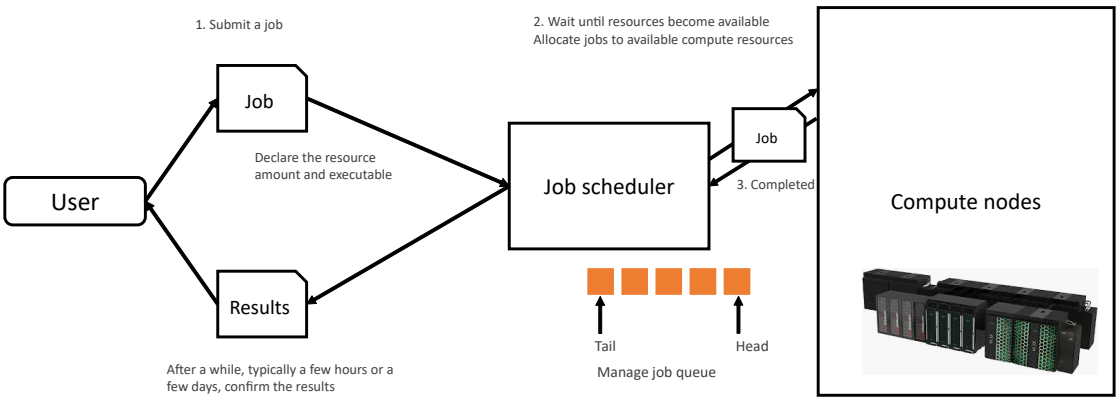



**Figure 2.1:** Basic workflow of job scheduling in HPC systems. Users submit jobs to the scheduler by declaring resource requirements and executable specifications. The scheduler maintains a job queue and allocates jobs to available compute resources based on scheduling policies. This process, from job submission to result retrieval, typically spans several hours to days depending on system load and job characteristics.

This optimization becomes particularly challenging in modern HPC environments where resources like CPU cores, memory, GPUs, and network bandwidth must be coordinated across thousands of nodes [36]. As shown in Figure 2.1, job schedulers manage this complexity by acting as intermediaries between users and compute resources, maintaining job queues and implementing sophisticated allocation policies. Job schedulers employ various algorithms and policies to achieve these goals, including backfilling strategies, which will be explained later, which allow smaller jobs to utilize otherwise idle resources while larger jobs wait for sufficient resources to become available.

### Ensuring fairness in resource allocation

Maintaining fairness among users represents another crucial role of job schedulers [39]. In shared HPC environments, multiple users and projects compete for limited resources, requiring careful arbitration to ensure equitable access. Schedulers typically implement several mechanisms to maintain fairness:

- Queue prioritization policies that balance different user and project requirements
- Resource usage accounting and quota enforcement
- Fair-share scheduling algorithms that consider historical resource usage patterns

These fairness mechanisms must operate transparently and predictably, allowing users to understand how their resource requests will be handled and enabling them to plan their computational workflows effectively [40].

#### Usability and security management

Job schedulers must balance system accessibility with security requirements, providing interfaces that enable efficient system utilization while maintaining proper access controls. This balance manifests in several key areas:

- User interfaces that facilitate job submission and management while hiding system complexity
- Access control mechanisms that enforce security policies and resource usage limits
- Audit logging systems that track resource usage and maintain accountability
- Implementation of organizational security policies and compliance requirements

These interfaces must be both powerful enough to support complex workflow requirements and intuitive enough to minimize barriers to system utilization [41]. Additionally, the scheduler must maintain comprehensive logging and monitoring capabilities to support both security requirements and operational analysis.

### 2.2.2 Core functions

Job schedulers implement several essential functions to effectively manage HPC resources and maintain system operations. These functions directly support the key purposes discussed in Section 2.2.1: system performance optimization, fairness assurance, and usability/security management. Here, we examine how each core function contributes to these fundamental purposes.

#### Scheduling

The scheduling function represents the core decision-making component of job schedulers, primarily supporting both system optimization and fairness objectives. At its most fundamental level, HPC job scheduling is an online problem where jobs arrive continuously and require immediate scheduling decisions. This online nature makes First-In-First-Out (FIFO) scheduling the foundational approach, where jobs are primarily processed in order of arrival.

Conceptually, job scheduling can be viewed as a two-dimensional packing problem [42]. Each job represents a rectangle in a space defined by requested time (duration) and requested resources (e.g., number of nodes or cores). The scheduler must efficiently pack these rectangles into the available system space while respecting both temporal and resource constraints. This packing must be done immediately upon job arrival, making it a particularly challenging online optimization problem.

From a system efficiency perspective, while FIFO scheduling serves as the foundation, backfilling algorithms enable improvements in resource utilization by allowing smaller jobs to utilize gaps in the schedule [43]. Importantly, backfilling maintains the fundamental FIFO principle by ensuring that these gap-filling jobs do not delay the scheduled start times of larger waiting jobs. Conservative backfilling takes this protection further by maintaining strict guarantees about job start times [44]. Beyond these basic scheduling mechanisms, queue management strategies provide another framework for optimization, preventing resource fragmentation and underutilization through sophisticated job ordering and resource allocation policies [45].

Fairness in resource allocation represents another crucial aspect of scheduling decisions. Fair-share scheduling mechanisms balance resource allocation across users and projects by considering historical usage patterns and adjusting job priorities accordingly [39]. Job prioritization is dynamically managed based on multiple factors including waiting time, resource requirements, and user/project priorities [46]. Additionally, preemption support helps maintain service level agreements and accommodate emergency computing needs [47], providing mechanisms to ensure critical jobs can access resources when needed while maintaining overall fairness in resource distribution.

The scheduling system must also address specific user requirements. Advanced reservation mechanisms enable users to secure resources for time-critical computations or specific experimental windows [48]. Additionally, topology-aware scheduling optimizes job placement by considering network proximity and communication patterns [49], particularly crucial for parallel applications that require intensive inter-node communication. These user-centric features ensure that the scheduling system not only maintains system efficiency and fairness but also supports the specific computational needs of the user community.

Gang scheduling represents a specialized form of runtime scheduling in HPC systems where multiple processes of a parallel job are synchronized to execute simultaneously across different nodes [50]. Unlike basic job scheduling that determines execution order and resource allocation, gang scheduling operates during job

runtime to coordinate process execution. When parallel processes are not scheduled simultaneously, communication and synchronization operations can experience significant delays. Gang scheduling addresses this by ensuring that all processes of a parallel job are either all running or all suspended together, maintaining the temporal alignment necessary for efficient parallel execution. Modern job schedulers like Slurm implement gang scheduling through their process management infrastructure [36].

### Resource management

Resource management functions serve multiple purposes within the job scheduler. For system optimization, they handle the dynamic allocation and deallocation of computational resources to maximize utilization [36] and coordinate resource state information across distributed system components [49]. Supporting both fairness and security objectives, these functions enforce resource usage limits and quotas at user and project levels [30] while maintaining proper resource isolation between different users and projects [41].

### Error handling

Error handling capabilities maintain system reliability while addressing both optimization and usability requirements [37]. From an optimization standpoint, these functions detect and respond to job failures or abnormal terminations [36], manage system-level failures such as node crashes or network issues, and implement job retry policies and failure recovery mechanisms. Supporting usability objectives, error handling provides user notification systems for job failures [36] and implements automated recovery procedures to maintain system availability.

### Monitoring and logging

Monitoring and logging functions support all three primary purposes of job schedulers [36], [40]. For system optimization, they provide real-time tracking of resource utilization, collect performance metrics, and generate usage reports for capacity planning. Supporting fairness objectives, these functions track resource usage patterns across users and projects, generate fairness metrics, and maintain historical data for fair-share scheduling algorithms. From a security and usability perspective, they maintain audit logs for compliance, provide user-accessible job status monitoring, and implement system health monitoring with alerting capabilities.

Together, these core functions create a comprehensive framework that enables job schedulers to fulfill their primary purposes while effectively managing modern

HPC environments. The sophistication of these functions continues to evolve as HPC systems grow in both scale and complexity, particularly with the introduction of new resource types and workload patterns.

## 2.3 Jobs as fundamental execution units

Jobs represent the fundamental units through which users access and utilize HPC resources [36], [37]. Unlike traditional computing environments where users directly execute programs on their local machines, HPC systems manage computational work through the abstraction of jobs. This abstraction serves several essential purposes in shared computing environments:

- Resource allocation tracking and management at a well-defined granularity
- Standardized interfaces for submitting and managing computational tasks
- Clear boundaries for security isolation and access control
- Consistent units for accounting and usage monitoring

Each job encapsulates not only the computational work to be performed but also the complete specification of resource requirements and execution parameters. This encapsulation enables the job scheduler to make informed decisions about resource allocation and execution timing while maintaining system-wide efficiency and fairness. In addition, jobs in HPC environments primarily fall into two broad categories based on their execution patterns and user interaction requirements: batch jobs that typically run without user intervention and interactive jobs that require real-time user interaction. These distinct job types, with their different characteristics and requirements, present unique challenges and opportunities for system resource management and scheduling policies.

### 2.3.1 Resource requirements and job types

A distinguishing characteristic of HPC jobs is the requirement for explicit specification of resource needs prior to execution [40]. At the fundamental level, job schedulers require detailed information about resource requirements, including:

- Number of compute nodes to be allocated
- Number of CPU cores per node

- Memory requirements per node
- Number of accelerators (e.g., GPUs) if needed

This explicit declaration serves several crucial purposes in HPC operations. For the job scheduler, it enables efficient resource allocation and optimization of system utilization through advanced scheduling algorithms. For system operators, it provides the basis for capacity planning and resource management policies. For users, it enforces disciplined resource usage planning and helps prevent wasteful over-allocation of system resources.

In practical HPC system implementations, these resource requirements are often abstracted through predefined configurations, commonly referred to as either "queues" or "job types" [30]. These abstractions represent standardized combinations of resources and execution policies. For example:

- A "cpu" type might allocate only CPU resources with specific core counts
- A "gpu" type might automatically pair CPU cores with appropriate GPU resources
- An "interactive" type might provide immediate access with specific resource limits

This abstraction through queues or job types simplifies the job submission process while maintaining the job scheduler's ability to perform detailed resource management. It provides a practical balance between the system's need for precise resource specification and users' need for straightforward resource access methods. Furthermore, these predefined configurations help prevent inefficient or inappropriate resource specifications that might arise from users unfamiliar with system architecture details.

### 2.3.2 Execution time specification

The specification of execution time represents a particularly important aspect of job definition in HPC environments [44]. Unlike typical computing environments where processes run until completion or user intervention, HPC jobs require upfront declaration of their maximum running time. This requirement serves multiple operational purposes:

- Enables schedulers to perform sophisticated planning and optimization

- Prevents indefinite resource occupation by failed or stuck jobs
- Allows users to express urgency through shorter time limits
- Facilitates different quality of service levels based on time requirements

However, in practice, accurate time specification presents significant challenges. Users often find it difficult to precisely predict their jobs' execution times, leading to conservative estimates that exceed actual requirements. This tendency to overestimate execution times can adversely affect scheduling efficiency. When users specify substantially longer time limits than necessary, it restricts the scheduler's ability to optimize resource allocation effectively, potentially leading to increased wait times for other jobs and reduced overall system utilization [51]. This discrepancy between specified and actual execution times represents an ongoing challenge in HPC resource management, highlighting the tension between providing users flexibility in time specification and maintaining efficient system scheduling.

### 2.3.3 Batch jobs

Batch jobs represent the traditional and predominant workload type in HPC environments. These jobs exhibit several distinctive characteristics that have historically shaped HPC system design and operation.

**Features**

- **Large-scale computation focus:**
  Batch jobs are predominantly used for large-scale computations, often utilizing hundreds or thousands of compute cores simultaneously. These jobs commonly implement parallel processing to maximize computational efficiency, whether through distributed memory parallelism (MPI), shared memory parallelism (OpenMP), or hybrid approaches [52].
- **Long execution duration:**
  Batch jobs typically run for long periods, ranging from hours to days or even weeks. This long duration characteristic stems from the computational complexity of the problems they address.The extended execution time makes these jobs less sensitive to initial scheduling delays, as the waiting time typically represents a small fraction of the total job duration [53].

- **High CPU utilization:**
  As shown in Figure 2.2, batch jobs typically maintain consistently high CPU utilization throughout their execution. This sustained high resource usage pattern emerges from their focus on computational efficiency and their design for maximum throughput. Batch applications are typically optimized to maximize resource utilization, implementing efficient algorithms and data structures that maintain continuous computation with minimal idle periods.

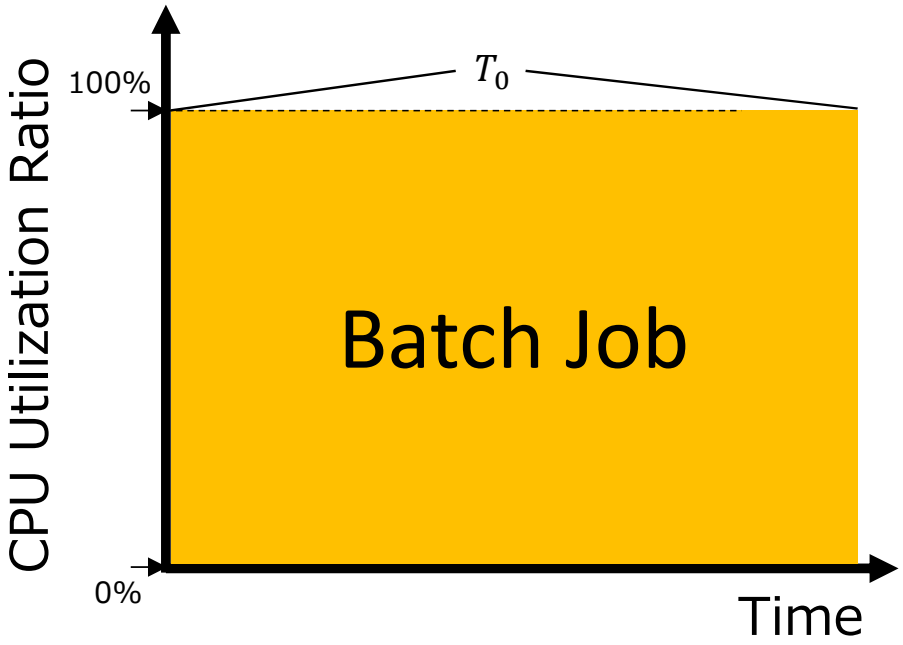



**Figure 2.2:** Typical CPU utilization pattern of batch jobs. These jobs maintain consistently high CPU utilization throughout their execution period, reflecting their compute-intensive nature and optimization for sustained performance.

These characteristics make batch jobs particularly well-suited for traditional HPC scheduling approaches. Their predictable resource usage patterns and tolerance for scheduling delays enable effective system-wide optimization.

### Usage for AI/ML training

Traditionally, batch jobs have been primarily used for large-scale numerical simulations, such as computational fluid dynamics [3], structural analysis [4], and climate modeling [6], where sustained computational power is required over extended periods. In recent years, however, AI/ML training workloads have emerged as a significant category of batch jobs in HPC environments.

Large-scale AI/ML training exhibits strong alignment with the characteristic features of batch jobs discussed earlier. The training of large language models

and deep neural networks requires significant parallel computing capabilities, often utilizing hundreds of GPUs simultaneously, which matches the large-scale computation focus of traditional batch jobs. As shown in Figure 1.1, major AI companies maintain massive GPU clusters for their model training needs. These training processes typically run for extended periods, particularly for models with hundreds of billions of parameters, demonstrating the same long-duration characteristics as traditional batch workloads. Furthermore, AI training maintains consistently high resource utilization throughout its execution, particularly of GPU resources, mirroring the high CPU utilization patterns seen in traditional HPC applications.

This natural alignment with batch processing patterns has enabled HPC centers to accommodate AI/ML training workloads without fundamental changes to their scheduling approaches. However, the increasing diversity of AI/ML workflows, particularly in development and experimentation phases, presents new challenges that extend beyond traditional batch processing patterns. The next section examines the interactive jobs used for broader AI development lifecycle such as model prototyping, hyperparameter optimization, and experimental validation.

### 2.3.4 Interactive jobs

Interactive jobs represent a distinct category of HPC workloads that differ significantly from traditional batch jobs in their execution patterns and resource requirements. In HPC systems, an interactive job generally refers to the entire session from when a user begins accessing their allocated computational resources until they complete their interactive work and release those resources. During this session, users can execute multiple commands or computational tasks without needing to wait for additional resource allocation - once the initial access is granted, the resources remain available throughout the session. Therefore, waiting time is defined as the duration between job request submission and when the user can actually begin accessing the resources.

**Features**

- **Real-time responsiveness requirements:**
  Interactive jobs demand immediate system response to maintain productive user workflows [13]. This responsiveness is crucial for activities such as code development, debugging sessions, and exploratory data analysis, where user productivity directly depends on system response time.

- **Short and intermittent execution patterns:**
  Unlike batch jobs that typically show continuous execution patterns, interactive jobs are expected to exhibit different characteristics. As shown in Figure 2.3, these jobs operate in shorter bursts of activity interspersed with periods of inactivity. Such patterns would be consistent with interactive workflows where computation phases might be followed by periods of result analysis, code modification, or thinking.

- **Diverse computational intensity and parallelism:**
  Interactive jobs typically alternate between busy and idle periods, with the intensity of resource usage varying significantly during active periods. While users perform different computational tasks - from compilation to visualization to test runs - each task may require different levels of computational resources. Furthermore, these jobs often utilize multiple cores simultaneously, meaning that the actual system load can be several times higher than the per-core utilization pattern shown in Figure 2.3. This combination of varying parallelism and task-dependent resource intensity results in complex utilization patterns that go beyond the simplified busy/idle representation in the figure.

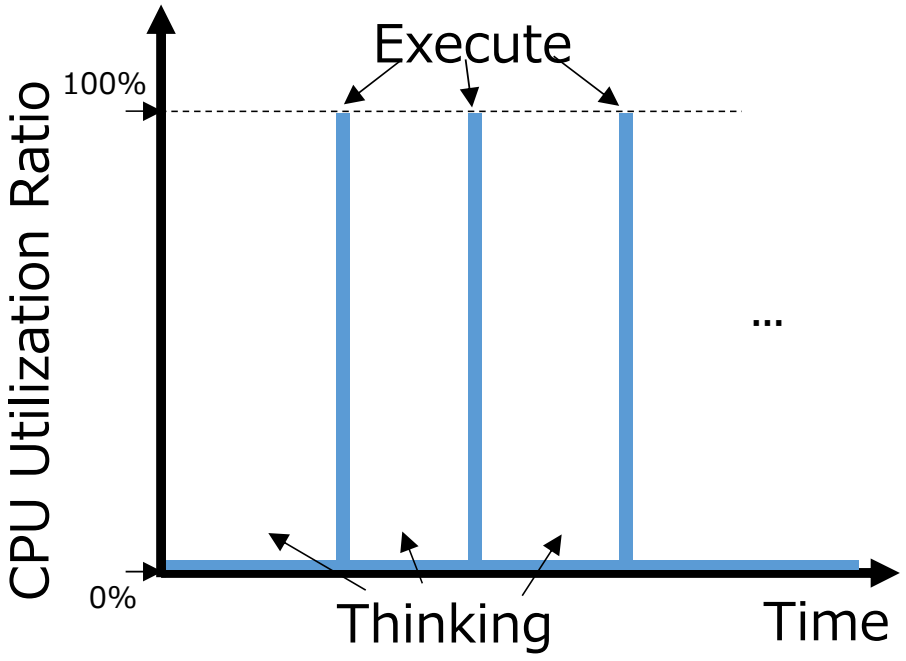



**Figure 2.3:** Conceptual comparison of CPU utilization patterns of interactive jobs. Interactive jobs show intermittent patterns of resource usage.

#### Typical applications

Interactive jobs serve various essential functions in modern HPC environments:

- **Development and debugging work:**
  Interactive sessions are crucial for code development and debugging tasks, where developers need immediate feedback on code changes and the ability to inspect program state in real-time [13]. This includes compilation testing, unit testing, and interactive debugging sessions.

- **Visualization processing:**
  Real-time visualization and data exploration tasks often require interactive access to compute resources. These applications might involve rendering complex scientific datasets, adjusting visualization parameters, or performing interactive data analysis [54].

- **AI inference workloads:**
  With the increasing prevalence of AI applications, interactive sessions are frequently used for model inference tasks and rapid prototyping of AI models. These workloads require quick response times but typically utilize resources in short bursts rather than sustained computation [11], [55]. AI development workflows often involve frequent iteration between model adjustment, testing, and hyperparameter optimization, making interactive access to computing resources essential for productive development cycles.

The growing importance of these interactive workloads, particularly in AI/ML development contexts, has led to increased focus on supporting their unique requirements within HPC environments. This shift challenges traditional HPC resource management approaches, which were primarily designed around the characteristics of batch workloads, and necessitates new strategies for efficient resource allocation and scheduling.

## 2.4 HPC Applications

HPC applications are distinctly different from conventional software applications due to their specialized requirements and characteristics. These applications are specifically designed to harness the massive parallel processing capabilities of supercomputer systems, often implementing sophisticated algorithms that can efficiently distribute computation across hundreds or thousands of processing units.

The majority of HPC applications are designed for execution as batch jobs, implementing sophisticated parallel algorithms that can efficiently distribute computation across hundreds or thousands of processing units. These applications require careful management of parallel execution, inter-process communication, and data distribution to achieve optimal performance at scale. Even when run in interactive sessions for development or testing purposes, these applications often utilize the same parallel programming models and optimization techniques, albeit typically at a smaller scale.

This section examines these fundamental characteristics of HPC applications, focusing on their parallel processing requirements, programming models, and performance evaluation methodologies. Understanding these characteristics is crucial for both application developers who need to create efficient parallel programs and system operators who must manage resources effectively for various execution patterns, from large-scale batch processing to interactive development and testing.

### 2.4.1 Importance of parallel processing

#### Scalability considerations

Scalability represents a fundamental requirement for HPC applications, directly impacting their ability to effectively utilize large-scale computing resources. Applications must maintain efficiency as they scale across increasing numbers of processors, which requires careful attention to:

- Load balancing across computational resources
- Communication overhead management
- Memory access patterns and data locality
- Synchronization requirements between parallel processes

#### Parallel efficiency significance

Parallel efficiency, measuring how effectively an application utilizes additional computational resources, plays a crucial role in both application design and system investment decisions [52]. This efficiency is typically expressed as the ratio of speedup achieved to the number of processors used, with two primary scaling metrics: strong scaling efficiency examines performance with fixed problem sizes, while weak scaling considers scenarios where the problem size increases proportionally with the number of processors.

From a system investment perspective, parallel efficiency directly impacts the return on infrastructure investments in HPC systems. When applications demonstrate good parallel efficiency, organizations can confidently invest in larger systems, knowing that additional computational resources will translate into proportionally better performance. This scalability ensures that infrastructure expansions deliver meaningful performance improvements, justifying the substantial costs involved in HPC system procurement and operation. Conversely, poor parallel efficiency can limit the practical benefits of system expansion, potentially leading to suboptimal utilization of expensive computational resources.

### 2.4.2 Parallel programming models

In HPC environments, achieving parallel execution typically requires users to explicitly modify their application source code to incorporate parallel processing capabilities. This modification often involves significant redesign of algorithms and data structures to effectively utilize parallel resources. While this approach demands considerable expertise in parallel programming, it remains the predominant method for achieving high performance in HPC applications.

Performance portability presents a significant challenge in parallel programming, as applications need to maintain efficiency across different architectures and system configurations [14]. While numerous frameworks and programming models have emerged to address this challenge, including domain-specific languages and high-level parallel programming interfaces [56], [57], the fundamental parallel programming models described below continue to form the foundation of most HPC applications. These basic models provide a balance between performance control and code maintainability, though achieving optimal performance across different architectures often requires careful consideration of system-specific characteristics.

This section focuses on three fundamental parallel programming models that form the backbone of HPC application development. While modern HPC environments support various specialized frameworks and higher-level abstractions, understanding these basic models remains crucial for effective parallel application development.

#### OpenMP (Shared memory parallelism)

OpenMP provides a widely-adopted programming model for shared memory parallel computing [58]. Its directive-based approach allows developers to incrementally parallelize existing code, making it particularly suitable for initial steps into parallel As shown in Figure 2.4, multiple CPU cores share direct access to the same memory space within a compute node. programming. Key characteristics include:

- Simplified parallelization through compiler directives (pragmas)
- Efficient utilization of multi-core processors
- Minimal code modification requirements

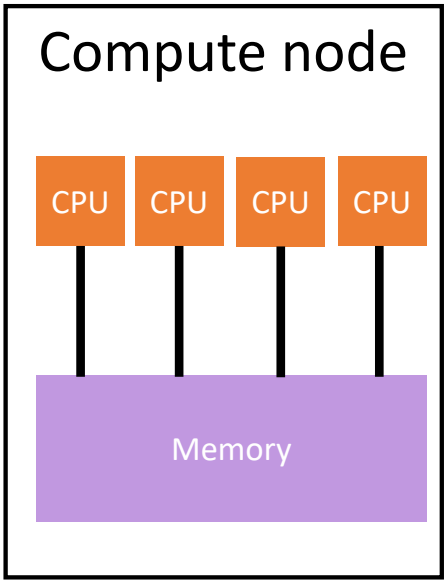



**Figure 2.4:** Shared memory architecture used by OpenMP within a compute node.

### MPI (Distributed memory parallelism)

The Message Passing Interface (MPI) stands as the de facto standard for distributed memory parallel programming [59], [60]. While traditionally requiring detailed knowledge of parallel programming concepts, modern implementations like mpi4py [61] have made it more accessible. As illustrated in Figure 2.5, MPI enables parallel computation across multiple compute nodes through message passing. Key features include:

- Explicit message-passing communication between processes
- Scalability across multiple compute nodes
- Comprehensive collective operations
- Available in traditional compiled languages and modern interpreted languages

MPI supports various communication patterns commonly found in scientific applications [59]:

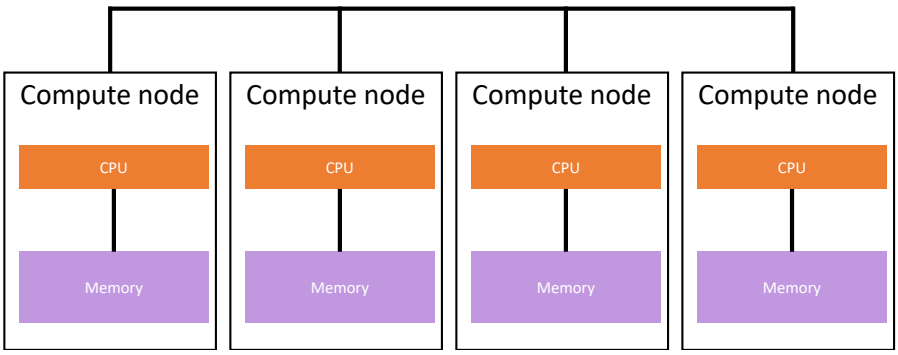



**Figure 2.5:** Distributed memory architecture used by MPI across compute nodes.

- Point-to-point communication:
  - Near-neighbor communication for domain decomposition
  - Long-distance communication for irregular data dependencies
- Collective communication operations:
  - All-to-all for global data redistribution
  - Broadcast for sharing common data
  - Reduction for computing global results
- Synchronization primitives:
  - Barrier synchronization for coordinating computational phases
  - Non-blocking operations for overlapping computation and communication

### CUDA (GPU-oriented parallelism)

CUDA represents NVIDIA's parallel computing platform for GPU programming [62]. While core CUDA programming requires detailed understanding of GPU architecture, higher-level frameworks like PyCUDA [63] and Numba [64] provide more accessible interfaces. As shown in Figure 2.6, CUDA programs manage computation and data transfer between CPU (host) and GPU (device) memory. The platform offers:

- High-performance computing on GPU architectures
- Specialized memory hierarchies for GPU computing
- Integration with both low-level and high-level programming languages
- Ecosystem of libraries and tools for various application domains

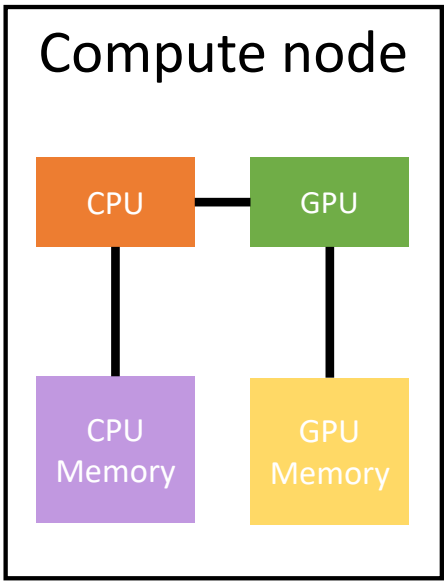



**Figure 2.6:** GPU architecture used by CUDA showing CPU-GPU memory separation.

### 2.4.3 Benchmark applications

Performance evaluation of HPC systems requires comprehensive benchmarking approaches that can assess different aspects of system capabilities. These benchmarks range from synthetic micro-benchmarks that measure specific system characteristics to more complex application benchmarks that simulate real scientific computations. The choice of benchmarks significantly influences our understanding of system performance, as different applications stress different aspects of the system architecture.

Performance metrics collected through these benchmarks typically include execution time, floating-point operations per second (FLOPS), memory bandwidth utilization, and communication overhead. These metrics help characterize various computational patterns, from compute-intensive kernels to memory-bound operations and communication-intensive patterns. This diversity in benchmarking approaches enables comprehensive evaluation of HPC systems across different usage scenarios.

#### Micro-benchmarks

Micro-benchmarks focus on measuring specific aspects of system performance:

- LINPACK [34]: Measures peak floating-point computational performance through dense linear algebra operations, serving as the primary metric for the TOP500 ranking

- STREAM [65]: Evaluates sustainable memory bandwidth and basic vector operations performance
- Intel MPI Benchmarks (IMB) [66]: Measures the performance of various MPI operations, including point-to-point communication and collective operations
- OSU Micro-Benchmarks [67]: Provides detailed assessment of MPI operations with various message sizes and communication patterns
- IOR [68]: Assesses parallel filesystem performance through various I/O patterns

These benchmarks provide fundamental insights into system capabilities and are particularly valuable for comparing different system architectures and configurations. The inclusion of communication benchmarks is especially important for understanding the performance characteristics of the interconnection network, which can significantly impact the performance of parallel applications.

### Application benchmarks

Application benchmarks simulate real-world scientific computations, providing more realistic assessments of system performance for typical HPC workloads. The NAS Parallel Benchmarks (NPB) [25], [26] represent a widely-used suite in this category, implementing a set of computational kernels derived from computational fluid dynamics applications. The NPB suite:

- Implements common scientific computing patterns
- Provides standardized problem sizes and configurations
- Includes both MPI and OpenMP implementations
- Enables evaluation of both computational and communication performance

**NAS Parallel Benchmarks in detail** The NPB suite consists of two main categories of benchmarks extracted from CFD applications [25], [26]:

- Computational kernels:
  - MG (Multigrid) for sparse, structured grid computations
  - CG (Conjugate Gradient) for sparse, unstructured grid computations
  - FT (Fast Fourier Transform) for spectral methods

  - IS (Integer Sort) for evaluating parallel sorting
  - EP (Embarrassingly Parallel) for evaluating baseline parallel performance
- Simulated CFD applications:
  - LU (Lower-Upper symmetric Gauss-Seidel): Solves systems representing Navier-Stokes equations using SSOR method
  - SP (Scalar Pentadiagonal): Solves pentadiagonal equation systems using a scalar approach
  - BT (Block Tridiagonal): Solves block-tridiagonal systems using a block approach

The NPB suite's significance extends beyond its CFD origins, as its computational and communication patterns align well with various scientific domains. For example, the computational patterns found in NPB's kernels - such as structured and unstructured grid computations, spectral methods, and particle methods - are fundamental to many scientific applications. Additionally, its communication patterns are representative of various scientific applications:

- Weather forecasting models like WRF (Weather Research and Forecasting Model), which employ near-neighbor communication among decomposed domains and reduction operations for error calculations [69]
- Molecular dynamics simulations such as LAMMPS (Large-scale Atomic/-Molecular Massively Parallel Simulator), utilizing particle-based near-neighbor communication and reduction operations for energy calculations [70]
- Quantum chemistry calculations in Quantum ESPRESSO, requiring long-distance and all-to-all communications, along with reductions for error convergence [71]
- Modern AI workloads, particularly in distributed training scenarios where collective operations and barrier synchronization are essential [72], [73]

Many production applications are commonly used as benchmarks to evaluate system performance, particularly domain-specific applications such as molecular dynamics simulations [70], [74], weather modeling systems [69], and quantum chemistry computations [71], each providing valuable insights into system performance characteristics for their respective scientific domains.

### 2.4.4 Performance analysis methodologies

Understanding and optimizing HPC application performance requires systematic characterization of application behavior. Multiple complementary approaches exist for this characterization, each providing different insights into application performance characteristics. These analyses help developers optimize their applications and assist system operators in understanding resource requirements for different application types.

#### Byte/FLOP ratio analysis

The Byte/FLOP ratio provides crucial insights into application performance characteristics [75]. This metric characterizes applications based on their computational intensity - the ratio of memory operations to floating-point operations:

- Identifies whether applications are memory-bound or compute-bound
- Provides insights into potential performance limitations
- Guides optimization strategies based on architectural characteristics
- Enables performance prediction across different system architectures

#### Performance counters

Hardware performance counters enable detailed analysis of application behavior at the processor level. The Performance Application Programming Interface (PAPI) [76] provides a standardized and portable interface to access these hardware counters across different architectures. These counters provide quantitative measurements of various hardware events during program execution:

- Cache utilization patterns and memory access characteristics
- Instruction mix and execution efficiency
- Branch prediction behavior and pipeline utilization
- Communication and I/O patterns

Modern profiling tools leverage PAPI and other performance counter interfaces to provide comprehensive performance analysis capabilities:

- Automated collection and analysis of performance metrics

- Visualization of performance data
- Identification of performance bottlenecks
- Correlation of hardware events with source code regions

The standardization provided by PAPI has been particularly important in HPC environments, enabling consistent performance analysis methodologies across different hardware platforms and architectures.

# 3

# Oversubscribing in HPC Systems: Fundamentals and Research Strategy

## Contents



As discussed in Chapter 1 and Chapter 2, modern HPC systems face significant challenges in managing increasingly diverse workloads, particularly with the growing prevalence of AI/ML applications alongside traditional scientific computing tasks. While these systems have historically developed various approaches to handle

different workload types, existing solutions often struggle to maintain both system efficiency and user productivity. This chapter examines these existing approaches, their limitations, and presents oversubscribing as a potential comprehensive solution.

## 3.1 Current solutions and their limitations

The diversification of HPC workloads, particularly the increasing importance of interactive computing needs described in Section 1.1.2, has led to several conventional approaches for managing mixed workload environments. However, these traditional solutions present significant limitations that impact both system efficiency and user productivity.

### 3.1.1 Preemption and suspend/resume mechanisms

One traditional approach to managing mixed workloads involves preemption mechanisms, where job schedulers control the suspension and resumption of jobs at the job level [47], [77]. This approach has proven successful in many operational contexts, particularly in workflow management systems where task preemption enables efficient handling of urgent computing needs [78]. When implemented effectively, preemption mechanisms offer several significant advantages for workload management.

The primary benefit of preemption mechanisms is their ability to enforce job priorities directly. Through preemption, high-priority or urgent jobs can access resources immediately when needed, making this capability particularly valuable in environments where certain workloads require guaranteed response times. Additionally, preemption provides system operators with effective resource reclamation capabilities, allowing them to dynamically reallocate resources from lower-priority jobs when necessary in response to changing system demands. In workflow management systems, this flexibility proves especially valuable, as preemption facilitates efficient scheduling by enabling the temporary suspension of less critical tasks to accommodate more urgent workflow components.

However, while these mechanisms provide valuable capabilities for workload management, they face several significant limitations in modern HPC contexts. The most significant challenge is the high overhead costs associated with job preemption, particularly for large parallel applications. Memory paging becomes a major bottleneck, as suspended processes may be swapped to disk when their memory pages are evicted [79]. For large-scale parallel applications that often utilize hundreds of gigabytes of memory across multiple nodes, this paging overhead can lead to substantial performance degradation when jobs are resumed. The situation

becomes even more complex with GPU-equipped systems, where the preemption mechanism must handle GPU memory management [80]. This introduces additional challenges such as whether to transfer GPU memory contents to CPU memory first and then to disk, or to manage GPU memory state directly. Given that modern AI workloads often utilize multiple GPUs with substantial memory, this GPU memory management adds another layer of complexity and potential performance overhead to the preemption process. This multi-level memory management overhead, combined with the cost of context switching and state restoration, can substantially impact overall system efficiency [81].

Another significant limitation lies in job termination risks. Many implementations of preemption actually terminate running jobs rather than truly suspending them, requiring users to implement their own checkpoint-restart mechanisms [82]. This approach can lead to lost computation and reduced productivity. Furthermore, resource utilization often suffers during the preemption process, as system resources typically remain idle while state saving and restoration operations complete. This idle time during state transitions reduces overall system efficiency and can impact the performance benefits originally sought through preemption mechanisms.

### 3.1.2 Checkpointing mechanisms

Checkpointing is a mechanism that saves the complete state of a running application, creating a snapshot that can be used to restore the application's execution at a later time. This state preservation typically includes memory contents, register states, and other execution-related information, saved either as checkpoint files or through specialized storage mechanisms. Checkpointing can be implemented at different levels of the software stack: system-level solutions like BLCR (Berkeley Lab Checkpoint/Restart) [83] that operate within the kernel, and application-level implementations that manage state preservation within the application code [84].

The primary advantage of checkpointing lies in its ability to resume computation from an intermediate state. When jobs are interrupted, whether due to system requirements or failures, they can later restart from their last checkpoint rather than from the beginning. This capability is particularly valuable when combined with suspend/resume mechanisms, providing a robust solution for preemptive scheduling scenarios. While suspend/resume operations handle immediate resource reallocation, checkpointing ensures that computation progress is preserved during these transitions [47]. This combination capability serves not only as a mechanism for planned job interruption but also as an insurance against unexpected process failures or system crashes, which are particularly concerning in long-running scientific computations.

By maintaining periodic checkpoints, applications can minimize the amount of lost computation in case of failures, significantly improving the reliability and efficiency of large-scale scientific workflows. Furthermore, checkpointing enables useful workflow patterns such as parameter sweeps, where multiple job variations can branch from a common intermediate state, improving overall computational efficiency.

However, checkpointing in HPC environments faces several significant challenges. System-level checkpointing, while offering transparent state preservation without requiring application modifications, has seen limited adoption in practice. This is primarily due to its high overhead when saving complete process information, lack of user-friendly interfaces for controlling checkpoint operations, and strong dependencies on specific operating system versions and kernel modules that limit portability across different systems.

Given these limitations, application-level checkpointing has emerged as the predominant approach, but it too faces significant challenges. A fundamental challenge lies in its development requirements. Applications must be specifically designed to manage their own state preservation, placing additional burden on developers. This requirement becomes particularly problematic when working with legacy applications or third-party software where source code modification is not feasible.

The storage implications of checkpointing present another significant challenge, particularly in large-scale HPC environments. Checkpoint data must capture complete application state, resulting in large storage requirements, especially for applications with substantial memory footprints. This storage overhead becomes particularly problematic in systems running multiple large-scale applications simultaneously. The management of checkpoint data requires careful consideration of storage allocation, cleanup policies, and data retention strategies.

### 3.1.3 Dedicated interactive nodes

Many HPC facilities address the mixed workload challenge by providing dedicated resources for interactive use [29], [30]. This can be implemented either by reserving a portion of an existing system's computational resources, or by maintaining entirely separate systems specifically for interactive workloads. In these approaches, dedicated resources ensure immediate availability for development, debugging, and other interactive tasks. This separation of resources provides a clear division between batch and interactive workloads, simplifying system management and providing predictable resource availability for interactive users. Some systems like

TSUBAME allow oversubscribing on these dedicated interactive nodes to improve their utilization [30], enabling multiple interactive jobs to share the same resources.

Figure 3.1 illustrates this approach, where the system is divided into two distinct partitions: interactive nodes and batch nodes. The job scheduler maintains separate queues for each partition, with interactive jobs directed to OSub-enabled interactive nodes and batch jobs to the traditional batch nodes. This separation enables immediate job execution for interactive workloads through the dedicated queue, while batch jobs continue to be processed through conventional scheduling mechanisms. The interactive nodes, being OSub-enabled, can accommodate multiple jobs simultaneously, helping to maximize the utilization of these dedicated resources.

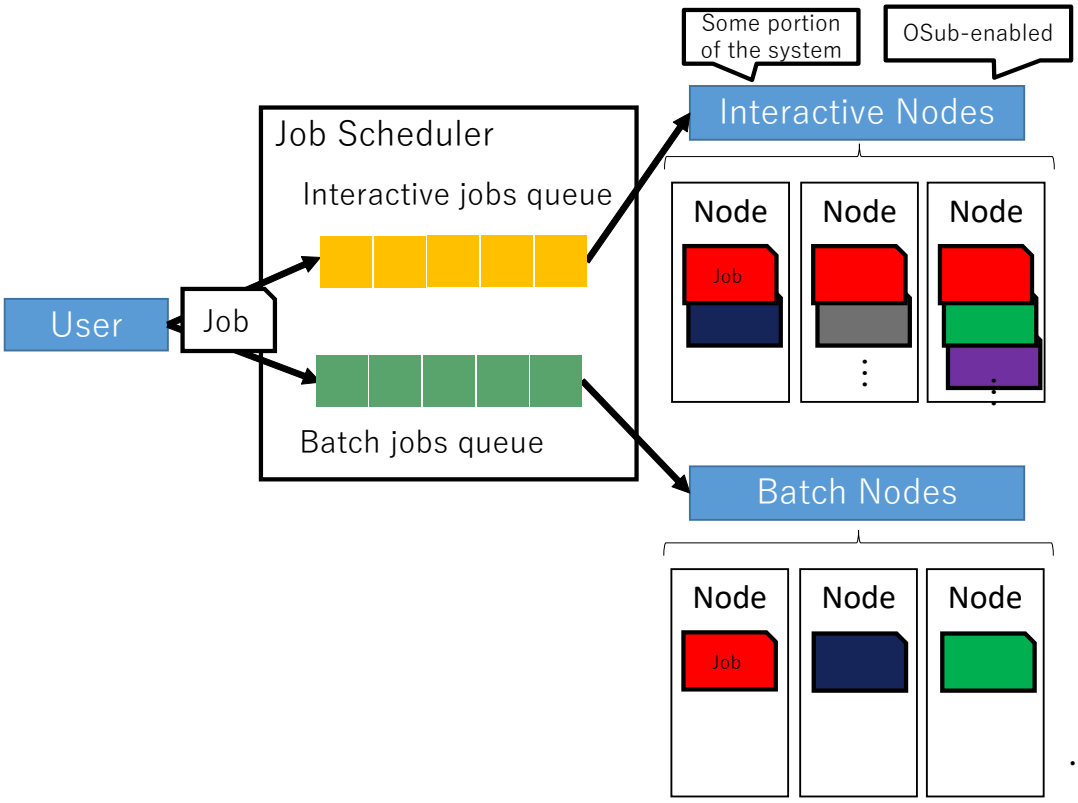



**Figure 3.1:** Illustration of dedicated interactive nodes approach. The system is partitioned into interactive and batch nodes, with separate queues for each. Interactive nodes support OSub to improve resource utilization, while batch nodes maintain traditional exclusive allocation.

However, this static resource partitioning introduces significant efficiency challenges, even when oversubscribing is enabled on the dedicated interactive nodes. During periods of low interactive demand, these nodes may be underutilized, while these same resources remain inaccessible to batch jobs even when the batch queue is heavily loaded. Conversely, during peak periods of interactive usage, users

may find the dedicated resources insufficient despite the ability to share resources through oversubscribing, particularly for larger-scale development or testing tasks that require more resources than available in the dedicated partition. This leads to queuing delays even when computational resources are available in the batch partition. This inflexibility in resource allocation can result in suboptimal overall system utilization, creating artificial barriers to productive system use. This limitation becomes particularly problematic in modern HPC environments where interactive workflows, such as AI model development, may require varying amounts of resources at different stages of development.

The approach also presents substantial configuration challenges for system operators. Determining the appropriate allocation of nodes between interactive and batch partitions requires careful analysis of usage patterns and user demands. This sizing problem becomes particularly challenging as workload patterns evolve over time, especially with the growing prevalence of AI/ML workflows that may blur the distinction between batch and interactive usage. Moreover, any static partitioning inherently limits the system's ability to adapt to changing workload patterns, potentially requiring frequent reconfiguration to maintain efficient operation.

These conventional approaches, while providing mechanisms for immediate resource access, face significant limitations in modern HPC environments. Preemption mechanisms suffer from high overhead costs, particularly for large parallel applications, while checkpointing requires substantial application-level modifications and careful management of checkpoint data. Dedicated interactive nodes, though providing guaranteed resource availability, lead to potential resource inefficiencies through static partitioning and face challenging configuration requirements.

The limitations of these traditional solutions have motivated exploration of alternative resource management strategies that can more effectively balance the needs of different workload types while maintaining system efficiency. The following sections examine oversubscribing as a potential solution, offering a simpler, more unified approach to resource management that could address many of these challenges in modern HPC environments.

## 3.2 Basic technology of oversubscribing

### 3.2.1 Basic mechanism of resource sharing

Oversubscribing (OSub) is a resource management approach where multiple computational tasks are allocated to share the same physical computational resources through time-division multiplexing. This concept has been widely adopted across

various computing domains, from operating systems to cloud computing environments [19], [20]. At its core, OSub allows logical resource allocations to exceed the available physical resources, enabling more efficient resource utilization through controlled sharing.

In traditional resource management, computational resources are often exclusively allocated to individual processes, ensuring that only one process can access a given resource at any time. This exclusive allocation model, while guaranteeing predictable performance, can lead to significant resource underutilization, particularly when processes do not consistently utilize their allocated resources at full capacity [85].

The fundamental premise of OSub is that computational processes typically exhibit varying resource utilization patterns, with periods of high activity interspersed with idle periods. These variations occur naturally in most computing environments - processes may be waiting for I/O operations [86], user input, and so on. By allowing multiple processes to share resources, OSub can utilize these idle periods more effectively.

### Logical resource allocation

At its core, OSub introduces the concept of logical resource slots that exceed the number of available physical resources. For example, a system with $N$ physical CPU cores might be configured to accept up to $M * N$ logical process allocations, where $M$ represents the multiplicity factor. This oversubscription of resources enables multiple processes to share the same physical resources through time-division multiplexing, as illustrated in Figure 3.2 using the example of OS-level process scheduling. The implementation spans from the hardware level to the system software level.

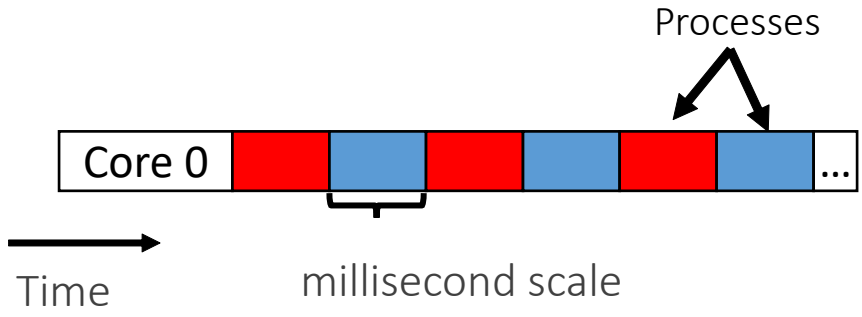



**Figure 3.2:** Example of time-division multiplexing in process scheduling. Multiple processes (shown in red and blue) share a single physical core through millisecond-scale time slices.

**Implementation layers**

OSub can be implemented at different layers of the system stack, each operating at different time scales:

- Hardware level: Through technologies like Simultaneous Multi-Threading (SMT), operating at microsecond scales with typical quantum lengths of 1-100 microseconds. This enables fine-grained resource sharing with minimal context switching overhead.

- Operating system level: Process scheduling and context switching typically operate at millisecond scales, with default time slices often ranging from 10-100 milliseconds. This provides a balance between responsiveness and context switching overhead.

- System software level: Coordinates resource allocation across multiple jobs at much coarser granularity, typically with time quanta in the range of seconds to minutes. This longer interval helps amortize the overhead of job state management and resource reallocation. In typical implementations, this functionality is built into the job scheduler itself. A notable approach at this level is gang scheduling [87], which coordinates the scheduling of parallel processes across multiple nodes to maintain synchronization and minimize performance degradation.

Each implementation layer offers different trade-offs between granularity of control, overhead costs, and ease of implementation. The choice of time quantum at each layer significantly impacts both the effectiveness of resource sharing and the overhead associated with context switching or job state management.

### 3.2.2 Previous research on oversubscribing

Research on oversubscribing spans multiple computing domains, with significant contributions in operating systems, cloud computing, and limited exploration in HPC environments. Each domain has approached resource sharing with different objectives and constraints.

### Research on operating systems

In operating systems, oversubscribing emerged as a fundamental concept with the advent of multi-process and multi-user computing systems in the 1960s. This resource sharing approach is commonly known as Time-Sharing System (TSS) in the operating systems field, representing one of the earliest and most successful implementations of resource multiplexing. The development of time-sharing systems, pioneered by projects like CTSS (Compatible Time-Sharing System) [88] and Multics [89], established the foundations for running multiple processes on shared computing resources. These early systems demonstrated that through careful scheduling and resource management, a single computer could effectively serve multiple users simultaneously, leading to more efficient resource utilization and improved system accessibility. These foundational developments in operating systems provided both the theoretical and practical basis for all subsequent resource sharing implementations. The core concepts of time-division multiplexing, process scheduling, and resource management developed during this period continue to influence how we approach resource sharing across all computing domains, from personal computers to cloud systems and HPC environments.

### Research on cloud computing

In cloud computing environments, oversubscribing (also commonly referred to as "overcommitment" or "overbooking" in the virtualization context) manifests primarily as the allocation of more virtual machines (VMs) than the underlying physical hardware can simultaneously support at full capacity [20]. Cloud providers routinely deploy multiple VMs on shared physical servers, with the total requested resources of these VMs often exceeding the actual physical capacity of the host machine. This approach is particularly effective because typical VM workloads rarely utilize their full allocated resources continuously, allowing providers to leverage the natural variations in resource consumption patterns across different VMs. The success of this model in cloud environments relies on sophisticated resource management systems that can dynamically monitor and adjust resource allocations based on actual usage patterns [19]. Cloud providers have developed pricing models that explicitly acknowledge different levels of resource guarantees, from dedicated instances with guaranteed resources to spot instances that accept potential resource contention in exchange for lower costs. This economic framework has enabled cloud providers to effectively balance resource utilization and service quality while maintaining profitable operations.

### Research on HPC systems

Within the HPC domain, research on oversubscribing has seen relatively limited adoption compared to other computing domains. The historically limited acceptance stems primarily from concerns about performance impacts and system complexity in HPC environments.

Research efforts in HPC oversubscribing can be broadly categorized into user-focused studies examining performance implications, and operator-focused studies investigating system-wide efficiency. This dual perspective reflects the inherent tension between maintaining application performance and improving system utilization through resource sharing.

From the user perspective, studies have addressed performance concerns using NAS Parallel Benchmark applications to examine various aspects of resource sharing. These investigations span hardware-level mechanisms like hyper-threading [23], co-scheduling strategies through gang scheduling with observed performance degradation of only 0-10% [90], and application synchronization characteristics in multicore environments [91]. These systematic evaluations using common benchmark applications have provided valuable insights into the practical feasibility of resource sharing in HPC contexts.

From the operator perspective, simulation-based studies have explored system-wide impacts of resource sharing in HPC workloads [21]. These studies have established foundational understanding of resource management mechanisms in HPC contexts, while highlighting opportunities for more detailed modeling of complex production environments.

A notable historical example of operator-focused implementation occurred in the 1990s with gang scheduling work [92]. This approach, implemented on a Cray T3D system, prioritized system efficiency through coordinated scheduling of parallel jobs. While initial results showed promising utilization improvements from 33.4% to 60.9%, subsequent systems did not widely adopt this approach. The challenges faced included memory constraints and scalability issues [93], highlighting the complexity of balancing system efficiency with practical operational requirements.

In terms of current software environments, commercial job schedulers like Slurm do provide basic oversubscription capabilities [36]. However, these implementations typically lack sophisticated mechanisms for protecting application performance during resource sharing, limiting their practical utility in production environments where performance predictability is crucial.

While these various studies and implementations have provided valuable insights into specific aspects of resource sharing in HPC environments, a comprehensive

evaluation framework incorporating both user and operator perspectives remains to be established. This gap is particularly significant given the evolving nature of HPC workloads and the increasing importance of supporting diverse computational patterns efficiently.

## 3.3 Adoption barrier of OSub in HPC systems

The adoption of oversubscribing has been limited compared to other computing domains such as cloud computing. This section discusses the reasons comparing success in cloud computing.

### 3.3.1 Key concerns of OSub in HPC systems

The limited adoption stems from several key concerns in the HPC field.

First, performance variability presents a significant barrier. HPC applications are often highly optimized for maximum performance, and any interference between jobs sharing resources could lead to unpredictable execution times and degraded performance [23]. This variability is particularly concerning for large-scale parallel applications where performance degradation in one process can cascade through the entire application.

Figure 3.3 illustrates this fundamental performance concern in OSub systems. Batch jobs typically maintain high CPU utilization throughout their execution, as shown on the left side of the figure where two jobs exhibit sustained computational loads. When these jobs are executed under OSub (right side), they must share the same physical resources through time-division multiplexing. This sharing introduces uncertainty in the actual processing speed achieved by each job, as indicated by the "???%" notation in the figure. The performance degradation can vary significantly based on factors such as:

- Resource contention patterns between co-located jobs
- Overhead from context switching between jobs
- Cache interference and memory access patterns
- Synchronization requirements in parallel applications

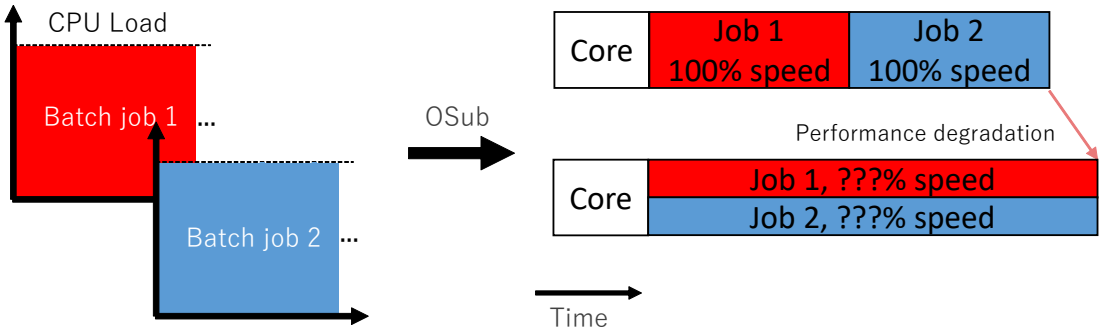



**Figure 3.3:** Illustration of performance variability concern with batch jobs under OSub. The left side shows the typical high CPU utilization patterns of batch jobs, while the right side demonstrates how OSub introduces uncertainty in processing speeds when these jobs share resources.

This unpredictability in performance makes it challenging for users to effectively plan their computational workflows and estimate completion times, particularly for time-sensitive or deadline-driven projects.

Second, fairness becomes a critical concern under oversubscribing. In traditional HPC environments, users requesting the same amount of computational resources (e.g., $N$ cores for $T$ hours) receive equivalent computing capability, ensuring that equal resource requests yield equal benefits. However, under oversubscribing, multiple users making identical resource requests might achieve significantly different amounts of actual computation due to varying interference from co-located jobs [24]. This inequity in the relationship between resource investment and computational benefit challenges the fundamental principle of fairness in HPC systems, where users expect their resource allocations to directly correspond to a consistent amount of computational work completed.

These concerns have historically led HPC centers to favor more conservative resource management approaches, typically preferring exclusive resource allocation despite its limitations in addressing modern workload requirements. As discussed in Section 3.2.2, while theoretical analyses and limited case studies exist [21], [23], systematic validation of oversubscribing in production HPC environments remains insufficient. Without comprehensive empirical evidence based on real workload patterns, the hesitancy to adopt oversubscribing approaches continues, despite their potential benefits for modern HPC operations.

### 3.3.2 Success cases of OSub in cloud computing

While HPC environments have been hesitant to adopt oversubscribing, cloud computing platforms have successfully implemented and benefited from this approach.

Their experience offers valuable insights for HPC systems, though the differences in usage patterns and requirements must be carefully considered. This section analyzes the success factors and considers the application to HPC domain.

### Key success factors

First, cloud computing environments differ fundamentally from HPC systems in their approach to resource management and performance guarantees. Cloud platforms commonly employ oversubscribing as a core strategy to optimize resource utilization across diverse workloads [20], [94]. Unlike HPC systems, where applications typically demand consistent, high-performance access to resources, cloud workloads often exhibit more variable resource demands and can tolerate some performance variation [95]. This flexibility in performance requirements has enabled cloud providers to implement aggressive resource sharing strategies while maintaining acceptable service levels.

Second, a key factor in cloud computing's successful adoption of oversubscribing lies in its pricing models. Cloud providers have developed sophisticated pricing strategies that align with different levels of resource guarantees. For instance, spot instances and burstable instances explicitly acknowledge potential performance variations in exchange for reduced costs [96], [97]. This economic framework provides users with clear trade-offs between performance consistency and cost, allowing them to choose resource allocation models that best suit their specific needs.

The success of this approach in cloud environments demonstrates several important principles:

- Resource sharing can be effectively managed when paired with appropriate economic incentives
- Users are willing to accept performance variations when the trade-offs are clear and beneficial
- Sophisticated resource management systems can maintain acceptable service levels even with significant oversubscribing

#### Lessons from cloud computing

While several key success factors exist in cloud computing, the pricing model concept provides particularly valuable insights for addressing fairness concerns in HPC environments. Similar to how cloud providers offer different service tiers, HPC systems could implement OSub-enabled queues where users accept potential performance variations in exchange for benefits like reduced waiting times. However, this approach requires clear specification of expected performance impacts and degradation ranges, unlike cloud environments where performance variations are quantified through pricing tiers. The key challenge lies in establishing transparent frameworks that maintain fairness while clearly communicating the trade-offs between resource availability and performance guarantees.

## 3.4 Potential of OSub in HPC systems

Despite the substantial barries, the introduction of oversubscribing in HPC environments offers promising solutions to several critical challenges faced by modern supercomputing facilities. As HPC workloads continue to diversify and user requirements evolve, traditional resource management approaches are becoming increasingly inadequate. OSub presents a unified approach to address these emerging challenges while maintaining system efficiency.

### 3.4.1 Adaptation to diversifying workloads

As mentioned in Section 1.2.2, modern HPC systems must accommodate an increasingly diverse range of workloads, particularly the coexistence of traditional batch jobs and interactive sessions. While batch jobs typically require sustained access to computational resources for long periods, interactive jobs demand immediate response but often exhibit intermittent resource utilization [13]. This fundamental difference in resource requirements presents a significant challenge to conventional management strategies, which OSub addresses through its inherent flexibility in resource allocation, enabling interactive jobs to efficiently utilize the idle periods inherent in batch workloads.

From the users' perspective, research has shown that development and debugging environments represent a significant portion of interactive computing needs in HPC systems [13]. These workflows typically require immediate access to resources but exhibit intermittent usage patterns, making them particularly suitable for resource sharing approaches. The ability to start debugging sessions promptly and

maintain interactive responsiveness during development cycles directly impacts user productivity, highlighting the potential benefits of OSub in these scenarios.

The emergence of AI/ML workflows in HPC environments, as described in Section 1.1.2, further emphasizes the need for flexible resource management. These workflows often combine elements of both batch processing (e.g., model training) and interactive computing (e.g., model development and debugging). For example, data scientists can conduct interactive development work alongside running training jobs, with the system automatically balancing resource allocation based on instantaneous demands [12]. OSub's ability to dynamically share resources makes it particularly well-suited to these hybrid workload patterns.

A key advantage of OSub in this context is its ability to provide unified resource management across different workload types. Rather than maintaining separate resource pools or complex scheduling policies for different job categories, OSub offers a single, coherent mechanism for managing all workloads [21]. This simplification not only provides greater flexibility in resource allocation but also reduces system management overhead.

### 3.4.2 Addressing critical waiting time issues

As discussed in Section 1.2, the reduction of waiting times represents another crucial advantage of OSub in HPC environments. Traditional scheduling approaches often result in significant delays, particularly for interactive workloads, as jobs must wait for sufficient resources to become available [22]. This waiting time directly impacts user productivity and can significantly impede the development and debugging processes that are essential to scientific computing workflows [12].

OSub addresses these challenges by enabling immediate execution through resource sharing [21]. When users request resources, whether for interactive sessions or batch computations, the system can immediately allocate logical slots, even if physical resources are currently occupied by other jobs. This capability is particularly valuable for various scientific computing scenarios. Development and debugging sessions require immediate response to maintain productive workflows, and exploratory data analysis and visualization tasks demand real-time interaction with the system [13]. Similarly, small-scale batch computations, such as preliminary analysis runs, parameter sweeps, or testing jobs, can begin execution promptly rather than waiting in queue behind larger jobs [77]. In the context of AI development, both interactive experimentation and small training runs are essential for effective model development and optimization, making immediate resource access crucial for maintaining efficient development cycles [12].

Beyond the immediate benefits to interactive workloads, this approach contributes to overall system efficiency in several ways:

- Improved Resource Utilization: By allowing multiple jobs to share resources, the system can maintain higher overall utilization rates while simultaneously reducing waiting times.

- Enhanced Development Workflows: The ability to begin execution immediately enables more efficient development cycles, whether for interactive debugging sessions or rapid batch testing iterations.

- Enhanced Planning Predictability: Reduced waiting times and more predictable resource availability enable users to better plan their computational workflows and project timelines.

## 3.5 Research approach

The successful implementation of oversubscribing in HPC environments requires careful consideration of both technical feasibility and practical operational requirements. This section presents our research approach, which aims to provide comprehensive validation of OSub effectiveness while maintaining practical applicability in production environments.

### 3.5.1 Research objectives

While oversubscribing represents a potential solution to modern HPC challenges, its practical viability remains unvalidated in production environments. The primary objective of this research is to provide comprehensive validation of oversubscribing as an effective resource management strategy for modern HPC systems. Through this comprehensive validation, we aim to establish oversubscribing as a practical solution that can address the fundamental tension between immediate resource access and system efficiency in modern HPC environments. This would enable HPC centers to better support emerging computational patterns, particularly in AI/ML development, while maintaining the high-performance characteristics essential for traditional scientific computing workloads.

### 3.5.2 Comprehensive evaluation framework: overall approach

#### Limitations in existing evaluation approaches

The adoption barriers discussed in Section 3.3.1 highlight the critical importance of systematic validation before implementing OSub in production HPC environments. While previous research has examined aspects of resource sharing in HPC contexts [21], comprehensive evaluation incorporating both performance impacts and operational considerations remains lacking. This gap is particularly significant given the increasing prevalence of interactive workloads in modern HPC environments, where traditional evaluation approaches focused solely on batch processing may no longer suffice.

Individual application performance evaluation alone cannot provide sufficient validation for OSub adoption in production environments. Similarly, simulations based on simplified workload models may fail to capture the complex dynamics of real HPC operations, particularly regarding interactive job behaviors and system-wide effects. These limitations in existing evaluation approaches have intensified concerns about adopting OSub within HPC environments, despite its potential benefits.

Furthermore, interactive jobs are particularly promising candidates for OSub due to their distinct operational characteristics - intermittent resource usage patterns, immediate response requirements, and variable-length computing phases. These characteristics theoretically align well with resource sharing through OSub, as the idle periods in interactive workflows could be efficiently utilized by other jobs. However, realizing this potential requires detailed understanding of actual interactive job behavior patterns in production environments. With the increasing prevalence of AI/ML workflows in HPC systems, comprehending these patterns has become crucial for effective OSub implementation. This understanding would enable more precise resource allocation strategies that capitalize on the natural resource usage patterns of interactive jobs while maintaining system efficiency.

#### Proposed comprehensive evaluation framework

To address these limitations in existing evaluation approaches, we propose a new comprehensive evaluation framework that encompasses three key dimensions:

First, we establish rigorous analysis of inter-job resource interference through:

- Analysis of resource contention patterns between co-located jobs
- Detailed performance characterization using hardware performance counters

- Development of predictive models for performance degradation

Second, we implement comprehensive workload system evaluation incorporating:

- Systematic analysis of resource sharing effects across different job types
- Quantification of job waiting time impacts
- Evaluation of system throughput under various sharing configurations

Third, we assess operational feasibility through:

- Evaluation of management complexity and overhead
- Investigation of configuration-free operation possibilities
- Assessment of robustness across different workload compositions

This framework provides a key advantage over previous evaluation approaches through its integration of both user and operator perspectives. Unlike previous approaches that often focused on either application performance or system metrics, our framework explicitly considers both user experience metrics (such as waiting time and performance predictability) and operational concerns (such as resource efficiency and management complexity).

Through this comprehensive approach, our framework provides insights not only into the immediate performance impacts of OSub but also into its broader implications for system operation and user productivity. This understanding is crucial for addressing the adoption barriers identified in Section 3.3.1 while ensuring that OSub implementation decisions are grounded in practical operational realities.

#### Focus on production data and real workload characteristics

Our framework emphasizes the importance of validation using real production data rather than relying solely on theoretical models or isolated benchmarks. This approach enables us to capture the true complexity of HPC operations through:

- Comprehensive workload traces from operational HPC systems
- Actual system configurations and operational constraints
- Real resource utilization patterns and job interactions

Within this production-focused evaluation, we specifically analyze the promising characteristics of interactive jobs identified earlier in Section 3.5.2 through detailed examination of actual workload data. This investigation characterizes real interactive job patterns, including temporal variations in CPU utilization, resource demand fluctuations, and interactions between concurrent sessions. This understanding of actual usage patterns is essential for evaluating OSub's potential benefits.

This emphasis on real operational data and workload characteristics helps bridge the gap between theoretical analysis and practical implementation, ensuring that our evaluation framework provides actionable insights for HPC centers considering OSub adoption.

### 3.5.3 Basic design of OSub implementation

Our research adopts a straightforward implementation approach to OSub that prioritizes operational simplicity while maintaining effectiveness. This design philosophy emphasizes practical deployability in production environments.

#### Simple implementation strategy

Unlike conventional approaches that require complex mechanisms such as process state management, application modifications, or static resource partitioning, our OSub implementation achieves simplicity through a minimalist design:

- Implementation solely at the system software level, requiring no modifications to applications or system software
- Control through a single parameter ($M$) that serves dual purposes:
  - Determines the maximum number of jobs sharing each resource
  - Inherently bounds total memory usage by limiting concurrent jobs, preventing excessive memory contention without complex state management
- Standard gang scheduling support for parallel jobs without additional complexity
- No static resource partitioning or complex configuration requirements

This straightforward design contrasts sharply with traditional solutions:

- Preemption mechanisms require complex process state management:

  - Must handle preservation and restoration of memory state across multiple nodes
  - Faces significant overhead when switching between memory-intensive parallel jobs

- Checkpointing approaches demand application modifications and careful management of checkpoint data

- Dedicated node solutions need static resource partitioning and ongoing configuration management

By focusing on essential functionality at the scheduler level while providing built-in resource constraints through the multiplicity factor, our approach maintains simplicity while effectively addressing the core requirements of resource sharing in HPC environments.

**Implementation architecture**

The practical realization of our simple design is illustrated in Figure 3.4, which shows how the single multiplicity parameter ($M$) translates into system-level resource sharing. This implementation creates a logical system capacity of $M$ times the physical resources through:

- Resource view transformation: The job scheduler presents an expanded view of system capacity to users, allowing more concurrent job allocations than physical resources while maintaining internal tracking of actual resource usage.

- Transparent allocation mapping: When jobs are submitted, the scheduler automatically maps multiple logical allocations onto physical resources without requiring user awareness of the sharing mechanism.

- Dynamic coexistence management: Multiple jobs share physical nodes through the system software level time-division multiplexing described in Section 3.2, operating at the scale of seconds to ensure efficient parallel job execution.

This architecture demonstrates how conceptual simplicity in design translates to practical implementation, with the single control parameter $M$ determining both the degree of resource sharing and the system's apparent capacity.

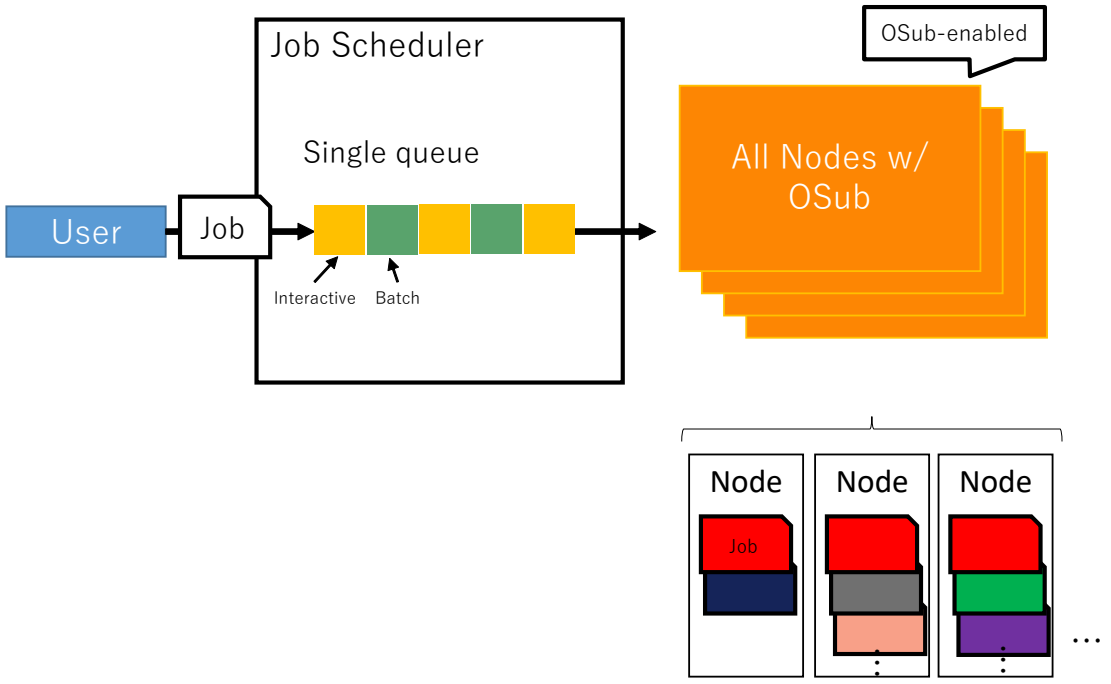



**Figure 3.4:** Implementation architecture showing how the multiplicity parameter $M$ enables system-wide resource sharing. The job scheduler transforms physical resources into an expanded logical view, enabling controlled concurrent access while maintaining operational simplicity.

### Focus on operational practicality

Our design emphasizes practical operational considerations through:

First, compatibility with existing HPC operations:

- Integration with standard job scheduling systems
- Support for existing job submission interfaces
- Preservation of current resource allocation policies

Second, incremental deployability:

- Selective implementation on specific system partitions, enabling initial deployment and validation in controlled environments while maintaining existing resource allocation methods for production workloads. This controlled approach allows system operators to verify OSub effectiveness without risking disruption to critical operations.
- Gradual expansion of OSub coverage based on operational validation results, allowing systematic evaluation of performance impacts and operational

procedures at each deployment phase. This step-by-step approach enables organizations to build confidence in OSub while developing operational expertise through practical experience.

This basic design provides a foundation for practical OSub implementation while enabling systematic evaluation of its effectiveness. The simplicity of the design, centered around the multiplicity factor, facilitates both research investigation and potential production deployment, allowing us to focus on validating fundamental benefits without being encumbered by implementation complexity.

### 3.5.4 Verification approaches

To validate the effectiveness of OSub in HPC environments, we implement four specific verification approaches. Each approach targets different aspects of OSub impact while maintaining methodological rigor through quantitative analysis and systematic evaluation procedures.

#### Application performance analysis methodology

Our first verification approach aims to establish that application performance under OSub remains predictable, even when resources are shared between multiple jobs. For parallel applications, this predictability critically depends on maintaining synchronized execution through gang scheduling, while for sequential applications, it relies on understanding resource interference patterns. This comprehensive predictability is crucial for practical deployment, as it enables system operators to provide performance guarantees and users to plan their computational workflows effectively. To achieve this goal, we employ the following methodology:

- Comprehensive performance measurements using the NAS Parallel Benchmark (NPB) suite [25], [26]:
  - Selection of representative parallel applications with different communication patterns
  - Analysis of synchronization requirements and their impact on performance
  - Implementation and evaluation of gang scheduling mechanisms
  - Inclusion of sequential applications with varying resource usage characteristics
  - Systematic variation of oversubscribing configurations

  - Controlled experiments combining batch and interactive workloads
- Hardware performance counter analysis to understand interference patterns:
  - Collection of detailed performance metrics using PAPI
  - Analysis of resource contention patterns
  - Measurement of interference effects between co-located jobs
- Development of performance prediction models:
  - Statistical analysis of performance counter data
  - Construction of regression models for interference prediction
  - Validation through cross-testing with different application combinations

**System-wide impact evaluation strategy**

Our second approach examines whether OSub can provide benefits for interactive jobs while maintaining acceptable system-wide performance. This balance between immediate resource access and overall system efficiency is essential for practical OSub deployment. To evaluate this trade-off, we implement:

- Development of a detailed system simulator incorporating:
  - Implementation of standard HPC scheduling algorithms
  - Support for various resource allocation policies
  - Integration of performance models from application-level analysis
  - Capability to process real workload traces
- Systematic evaluation methodology to assess both interactive job responsiveness and system efficiency:
  - Analysis of waiting time metrics across configurations
  - Measurement of system throughput under various scenarios
  - Assessment of resource utilization patterns
- Parameter sensitivity analysis framework to understand operational boundaries:
  - Systematic variation of oversubscription ratios
  - Testing of different queue configurations
  - Investigation of system size effects
  - Analysis of workload composition impacts

### Interactive workload analysis methodology

Our third approach focuses specifically on validating OSub's effectiveness for interactive workloads, which represent a growing proportion of HPC usage, particularly in AI/ML development. Given the distinct characteristics of interactive jobs - intermittent resource usage, immediate response requirements, and variable computing phases - we implement:

- Collection and analysis of interactive job data to understand real usage patterns:
  - Recording of CPU utilization patterns
  - Measurement of resource usage cycles
  - Tracking of idle periods and computation bursts
  - Analysis of resource demand variations
- Development of enhanced workload models that capture interactive job characteristics:
  - Incorporation of temporal utilization patterns
  - Modeling of interactive session behaviors
  - Integration of AI/ML development workflow characteristics
  - Support for varying resource demand patterns
- Systematic testing methodology to validate OSub effectiveness:
  - Progressive variation of interactive job ratios
  - Testing of different workload compositions
  - Evaluation across multiple system configurations

### Physical system validation methodology

Our fourth approach aims to validate that the benefits observed in simulation studies can be realized in actual HPC environments. This physical validation is crucial for establishing OSub as a practical solution for production systems. To achieve this, we implement:

- System implementation framework for practical deployment:
  - Integration with Slurm job scheduler

  - Implementation of gang scheduling support
  - Creation of workload generation tools

- Experimental design methodology for systematic validation:
  - Definition of clear experimental phases
  - Development of workload scaling procedures
  - Implementation of reproducible runtime environments
  - Setup of comprehensive monitoring infrastructure

- Validation procedures to verify simulation findings:
  - Design of controlled experiments
  - Development of performance measurement protocols
  - Implementation of systematic testing procedures
  - Creation of data collection and analysis pipelines

These four verification approaches together provide a comprehensive framework for evaluating OSub viability in HPC environments. Each approach is specifically designed to validate different aspects of OSub effectiveness, from performance predictability to practical deployability, while maintaining focus on the requirements of production HPC environments.

# 4

# Performance of Applications under Oversubscribing Scheduling

## Contents



## 4.1 Introduction to performance evaluation

As discussed in Section 3.3.1, performance variability and fairness represent fundamental barriers to adopting oversubscribing in HPC environments. These concerns stem from the significant performance impacts that can occur when multiple jobs share computational resources, particularly given HPC applications' need for consistent, high-performance execution. While oversubscribing has demonstrated success in cloud computing environments where performance variations can be managed through pricing models and service level agreements, HPC environments require more rigorous validation of these performance impacts before adoption can be considered.

This chapter presents a systematic analysis of performance impacts under oversubscribing, focusing particularly on establishing quantitative understanding of:

- Performance degradation patterns across different application types
- Resource interference mechanisms between co-located jobs
- Predictability of performance impacts under various sharing scenarios

Our analysis covers both parallel and sequential applications, as well as batch and interactive jobs, to provide comprehensive understanding of performance impacts across different workload types in modern HPC environments.

Of particular concern is the impact on parallel applications, which represent the predominant workload in HPC systems. In parallel applications, worker processes must coordinate their execution through communication and synchronization, as explained in Section 2.4.2. When even a single worker process experiences performance degradation due to resource sharing, it can slow down the entire application as other processes must wait for the delayed worker to complete its tasks. Moreover, when these workers are in an oversubscribed state, they may not effectively perform synchronization and communication operations. This synchronization requirement means that parallel applications may suffer more severe performance degradation than would be expected from the degree of resource sharing alone.

Sequential applications, while not subject to synchronization constraints, face different challenges under resource sharing. These performance impacts stem from multiple sources in the computing system hierarchy. At the hardware level, jobs compete for CPU execution units, cache space, and memory bandwidth. The sharing of cache memory can lead to increased cache misses as jobs evict each other's data, while memory bandwidth contention occurs when multiple processes attempt to access memory simultaneously through shared channels. The resource utilization patterns and resulting performance impacts vary significantly depending on application characteristics, making it crucial to understand the compatibility patterns between different types of applications.

Understanding and quantifying these performance impacts is essential not only for validating the feasibility of oversubscribing in HPC environments but also for developing practical implementation strategies that can maintain both performance predictability and fairness. The following sections present our methodology for measuring these effects and our findings regarding their impact on different types of applications. Furthermore, for sequential applications, we develop predictive models

that characterize application compatibility, providing a foundation for informed resource sharing decisions in production environments.

Through this systematic analysis, we aim to address the fundamental concerns about performance variability that have historically limited oversubscribing adoption in HPC environments. By establishing quantitative understanding of performance impacts and developing predictive capabilities, we provide the empirical foundation necessary for implementing oversubscribing while maintaining the performance characteristics essential for HPC operations.

## 4.2 Performance evaluation under oversubscribing

### 4.2.1 Experimental setup

#### Target applications

We are interested in the cases where parallel and sequential jobs, and batch jobs and interactive jobs share computing resources. We apply and use NAS Parallel Benchmarks (NPB) [25] as basis, and modify them for interactive jobs experiment. This benchmark set is provided by NASA and mimic various HPC applications. Table 4.1 summarizes the programs and classes (problem sizes), and the characteristics cited from [25]. In the following, when specifying a program including its class, we use the notation like "cg.B". In addition, we employed a simple matrix multiplication kernel (mm) as a CPU-intensive application. All the above applications are parallelized by OpenMP to utilize the processor core inside a compute node.

For interactive jobs, we modify the NPB applications to mimic behaviors of interactive usage, by repeating busy periods and idle periods. The modifications to the program are shown in Listing 1. As an example, the case of sp is shown. The main loop of each application is modified to sleep after a certain loop length (`STEP`) is calculated. `sleepT` in line 4 is generated by random number and different value for each time but we omit the part in the Listing for simplicity.

#### Environment

**Hardware and software settings** The specification of the machines we used is shown in Table 4.2. The compiler and performance profiler we used are shown in Table 4.3. PAPI [98] and Score-P [99] were used to obtain the performance monitoring counters (PMCs) to build a performance model.

**Listing 1:** Mimic interactive application. By inserting `sleep` it behaves similarly to Figure 2.3.

```
do step = 1, niter
    call adi                !! This is a compute kernel.
    if (mod(step, STEP) .eq. STEP-1) then
        call sleep(sleepT) !! Call sleep for a interval.
    endif
end do
```

**Table 4.1:** The specification of the workloads

| Program | Class | Characteristic |
|---|---|---|
| ft | A-D | Discrete 3D fast Fourier Transform, all-to-all communication |
| mg | A-D | Multi-Grid on a sequence of meshes, long- and short-distance communication, memory intensive |
| sp | A-D | Scalar Penta-diagonal solver |
| lu | A-D | Lower-Upper Gauss-Seidel solver |
| bt | A-D | Block Tri-diagonal solver |
| ep | A-D | Embarrassingly Parallel |
| cg | A-D | Conjugate Gradient, irregular memory access and communication |
| mm | - | Simple matrix multiplication kernel |

**Table 4.2:** The specification of the machine

| For sequential jobs experiment: Intel Xeon Gold 6140 | |
|---|---|
| Num. of Cores | 36 |
| Num. of Cores per socket | 18 |
| Num. of Sockets | 2 |
| Level 1 Cache | 32KB I + 32KB D |
| Level 2 Cache | 1024KB |
| Level 3 Cache | 25344KB |
| Operating Frequency | 2.30 GHz |
| Hyper-Threading Setting | Enabled |
| **For parallel jobs experiment: AMD EPYC 7513** | |
| Num. of Cores | 64 |
| Num. of Cores per socket | 32 |
| Num. of Sockets | 2 |
| Level 1 Cache | 2MiB I + 2MiB D |
| Level 2 Cache | 32MiB |
| Level 3 Cache | 256MiB |
| Operating Frequency | 2.60 GHz |
| Hyper-Threading Setting | Enabled |

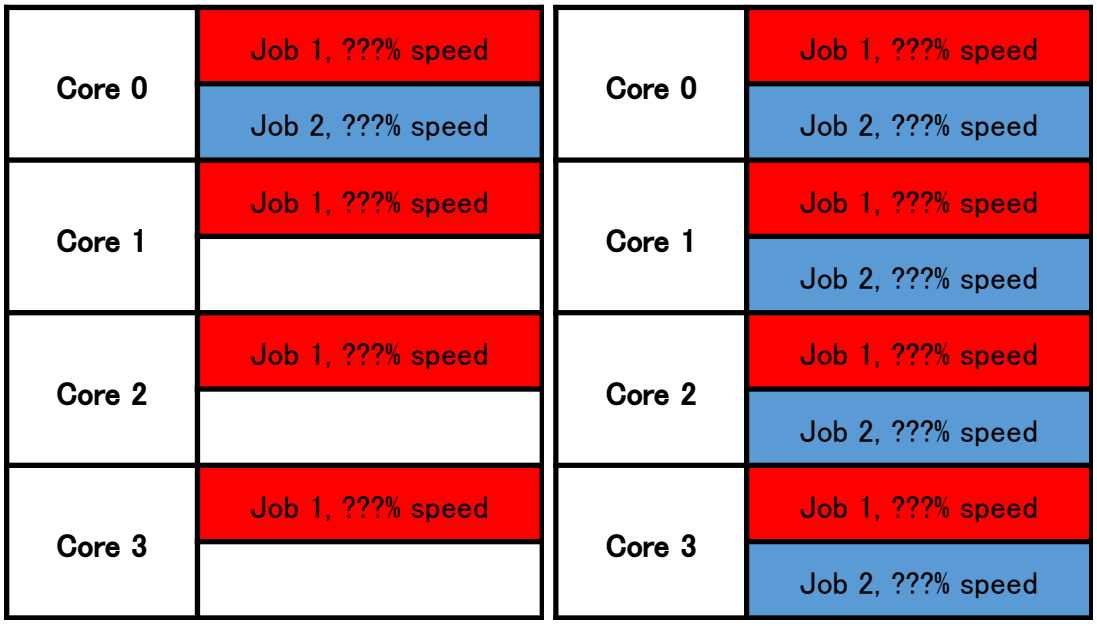
**Table 4.3:** A compiler and performance profiler

| Software | |
|---|---|
| Compiler | gcc 9.3.0 (For sequential jobs experiment), 11.4.1 (For parallel jobs experiment) |
| Compiler Options | `-O3 -fopenmp` |
| Performance Profilers | PAPI 5.7.0.0 [76], [98], Score-P 6.0 [99] |

**Settings for oversubscribing** On top of the above mentioned node, we evaluate performance of jobs when they share the same cores. In the following experiments, the number of jobs that shared resources (multiplicity $M$) is two. Also, we mainly use hardware and OS level oversubscribing, as shown in Section 3.2.

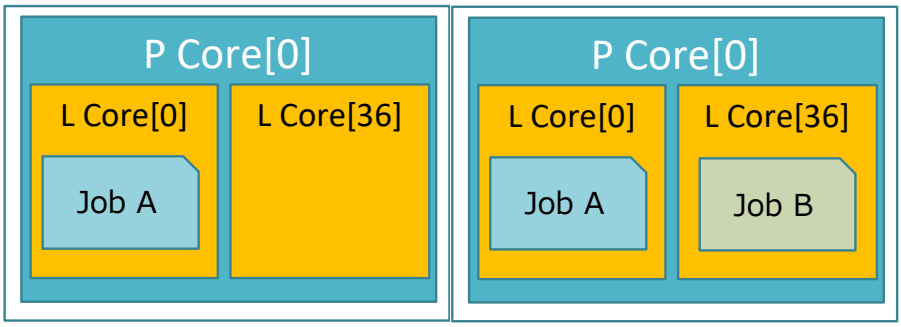
For paralell jobs experiment, we apply two experimental cases: partial oversubscribing and overall oversubscribing as shown in Figure 4.1, utilizing OS level oversubscribing. The former examines the effect when one process of parallel processes is oversubscribed while the latter when all processes are oversubscribed. In addition to OS level overusbscribing, we conduct the experiment with system software level oversubscribing, providing gang scheduling.

For sequential jobs experiment, the condition is quite simple. Figure 4.2 shows the case where a single physical core is shared by two jobs, utilizing hardware level oversubscribing. In the machine, logical core 0 and 36 share a single physical core 0. Thus each job process is bound to each logical core.




(a) Partial oversubscribing (b) Overall oversubscribing

**Figure 4.1:** Experimental conditions for parallel applications under oversubscribing.




(a) Using a dedicated core (b) Under oversubscribing

**Figure 4.2:** Oversubscribed state with Hyper-Threading enabled

### 4.2.2 Performance degradation of parallel applications

To quantify the performance impact of oversubscribing on parallel applications, we conducted experiments using the NAS Parallel Benchmarks (NPB). These experiments focused on measuring application speed under both dedicated and shared resource conditions. Our results demonstrate that resource sharing can significantly impact parallel application performance, with the degree of impact varying considerably across different applications. Our experimental condition encompasses two type: one is partial oversubscribing the other is overall oversubscribing as explained in Section 4.2.1.

**Table 4.4:** Effects of resource sharing on performance of NPB jobs on partial oversubscribing. Performance degradation bases the half speed of single speed.

| Benchmark | # of processes | Single speed | Speed with sharing | Performance degradation |
|---|---|---|---|---|
| bt.B | 9 | 55.28 | 30.71 | -11% |
| lu.B | 12 | 123.15 | 63.41 | -3% |
| mg.C | 8 | 50.15 | 22.91 | 9% |
| sp.B | 9 | 46.10 | 27.57 | -20% |

Table 4.5 shows the result of partial oversubscribing. For all benchmark applications, the performance degradation closely matches the expected 50% reduction in CPU time when sharing resources. The results demonstrate that slowest process dominates the entime application performance.

Table 4.5 shows the results of overall oversubscribing, where every core used by the parallel application is shared with another job. Under these conditions, the performance impact is substantially more severe than in partial oversubscribing, with all applications showing degradation well beyond the expected 50% reduction from simple time-sharing. This increased performance impact stems from the compounding effects of uncoordinated resource sharing across all participating

**Table 4.5:** Effects of resource sharing on performance of NPB jobs on overall oversubscribing

| Benchmark | # of processes | Single speed | Speed with sharing | Performance degradation |
|---|---|---|---|---|
| bt.B | 9 | 55.28 | 19.18 | 28% |
| lu.B | 12 | 123.15 | 30.76 | 50% |
| mg.C | 8 | 50.15 | 11.21 | 55% |
| sp.B | 9 | 46.10 | 9.37 | 59% |

processes. When every process independently competes for resources, the lack of synchronization leads to significant coordination overhead and idle time as processes wait for slower peers to complete their portions of computation. This demonstrates that naïve application of oversubscribing to parallel applications can lead to severe performance penalties, particularly for applications with tight synchronization requirements.

**Table 4.6:** Effects of resource sharing on performance of NPB jobs on overall oversubscribing with gang scheduling

| Benchmark | # of processes | Single speed | Speed with sharing | Performance degradation |
|---|---|---|---|---|
| bt.B | 9 | 55.28 | 27.18 | 2% |
| lu.B | 12 | 123.15 | 68.07 | -11% |
| mg.C | 8 | 50.15 | 20.07 | 20% |
| sp.B | 9 | 46.10 | 22.83 | 1% |

To address these synchronization challenges, we implemented gang scheduling to coordinate resource sharing across all processes of parallel applications. Table 4.6 demonstrates that this approach effectively mitigates the performance impact of oversubscribing on parallel applications. By ensuring all processes are scheduled simultaneously, gang scheduling eliminates the coordination overhead and idle time that severely impacted performance under uncoordinated sharing. Most applications now achieve performance close to or even better than the theoretical 50% of original speed expected from time-sharing, making oversubscribing with gang scheduling a viable approach for parallel applications.

### 4.2.3 Performance degradation of sequential applications

#### Definition of performance degradation ratio

For sequential jobs experiment, we introduce performance degradation ratio in an asymmetric style in order to capture that jobs may have different execution times and different CPU utilization ratios.

We assume that a batch job, which is called the *victim* job, is running for a long time. Then the other job, called the *opponent* job, starts execution using

the same cores. While we assume the victim job keeps high CPU utilization ratio, the opponent's CPU utilization may or may not fluctuate; the opponent may be either a batch job or an interactive job.

Then we define several variables as follows. $T_{\mathrm{v,Ded}}$ is the execution time of the victim on the dedicated cores (Figure 4.2 (a)) and $T_{\mathrm{v,OSub}}$ is the execution time of the victim running with the opponent job.

We also define similar variables for the opponent, but we should take care of fluctuation of CPU utilization since it may repeat busy and idle periods if it is an interactive job. $T_{\mathrm{o,ded}}$ is the execution time of the opponent job on the dedicated cores, excluding of idle periods. Thus it is the total time of busy periods[1]. Similarly, $T_{\mathrm{o,OSub}}$ is the total time of execution periods under oversubscribing.

Using these variables, we define performance degradation ratio $E_{\mathrm{v}}$ and $E_{\mathrm{o}}$ by the following equations:

$$E_{\mathrm{v}} = \frac{T_{\mathrm{v,OSub}} - T_{\mathrm{v,Ded}}}{T_{\mathrm{o,OSub}}} \tag{4.1}$$

$$E_{\mathrm{o}} = \frac{T_{\mathrm{o,OSub}} - T_{\mathrm{o,Ded}}}{T_{\mathrm{o,OSub}}} \tag{4.2}$$

Note that they are not simple relative values of extension of execution times, like $(T_{\mathrm{v,OSub}} - T_{\mathrm{v,Ded}})/T_{\mathrm{v,Ded}}$. Instead both have the denominator of $T_{\mathrm{o,OSub}}$ for the following reason. If the opponent is an interactive job with low CPU utilization (idle periods are dominant), it is natural that relative extension of the victim is minor if the denominator is $T_{\mathrm{v,Ded}}$. To avoid underestimation of effects of the opponents, our definitions of degradation ratio are formulated as above.

It is reasonable that the threshold for performance improvement is 0.5 both for the victim and the opponent. We can explain the reason for victim. First, $T_{\mathrm{v,OSub}}$ can be transformed as Eq.4.3. The first term is dedicated execution time and the second is oversubscribed execution time. Thus, Eq.4.4 holds ideally between $T_{\mathrm{v,OSub}}$ and $T_{\mathrm{v,Ded}}$. Eq.4.4 can be transformed as Eq.4.5. We can gain $E_{\mathrm{v}} = 0.5$ by using Eq.4.1 and Eq.4.5. We can explain in the same way for opponent, using Eq.4.2 and Eq.4.6.

$$T_{\mathrm{v,OSub}} = (T_{\mathrm{v,OSub}} - T_{\mathrm{o,OSub}}) + T_{\mathrm{o,OSub}} \tag{4.3}$$

$$T_{\mathrm{v,Ded}} = (T_{\mathrm{v,OSub}} - T_{\mathrm{o,OSub}}) + T_{\mathrm{o,OSub}}/2 \tag{4.4}$$

$$T_{\mathrm{v,OSub}} = T_{\mathrm{v,Ded}} + T_{\mathrm{o,OSub}}/2 \tag{4.5}$$

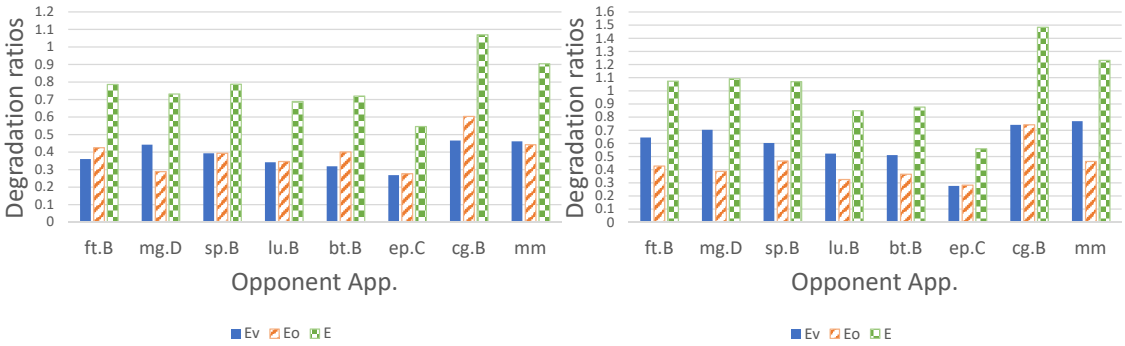
$$T_{\mathrm{o,OSub}} = 2T_{\mathrm{o,Ded}} \tag{4.6}$$

[1]If the opponent is a batch job, it is same as the execution time

From the above discussion, the expected degradation ratio is 0.5 for both if they do not suffer from effects of oversubscribing overhead (e.g. cache pollution or memory bandwidth contention) except for multiplicity. By harnessing this property, we can define well-matched job pairs for oversubscribing. If the sum of ratio of the victim and the opponent, $E = E_{\mathrm{v}} + E_{\mathrm{o}}$, is smaller than 1.0, we call the pair of the two jobs well-matched. If $E$ is larger, we should avoid oversubscribing for that pair.

### Performance when two batch jobs are oversubscribed

The measurement has been done basically based on methodology in the Section 4.2. In the experiments, we have executed both the victim and the opponent repeatedly. As the results, both jobs are always affected by the coexisting job.




(a) sp.B (b) cg.B

**Figure 4.3:** The degradation ratio when two batch jobs share a physical core. Each job uses a single thread. We extract sp.B and cg.B as typical victim applications.

**Table 4.7:** The percentages of cases degradation ratio is less than (or more than) the threshold. Each job uses a single thread. It shows as a percentage. The threshold $t$ is 0.5 for $E_v$ and $E_o$, 1.0 for $E$.

| | $\leq t$ | $> t$ |
|---|---|---|
| $E_{\mathrm{v}}$ | 92% | 8% |
| $E_{\mathrm{o}}$ | 89% | 11% |
| $E$ | 92% | 8% |

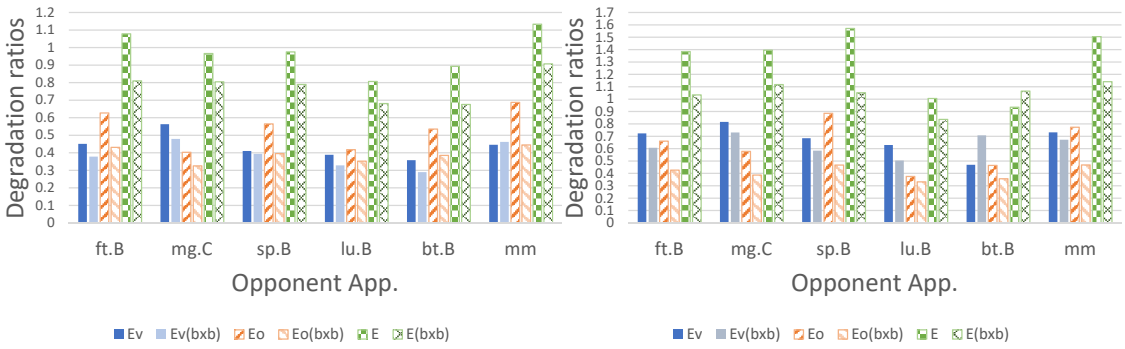
Two jobs share a single physical core as shown in Figure 4.2 (b). The applications shown in Table 4.1 are used for the victim and opponent, and the number of measured combinations is 800, including different problem sizes. The cases that cannot execute due to memory deficit or other reasons are excluded. Among the measurement, Figure 4.3 shows the results when the victim job is sp.B or

cg.B. The horizontal axis indicates the opponent job, and the bars indicate the degradation ratio $E_v, E_o$, and $E = E_v + E_o$.

According to Figure 4.3 (a), $E_v$ of sp.B is less than 0.5, which indicates that oversubscribing has advantageous in performance. On the other hand, Figure 4.3 (b) shows $E_v$ of cg.B is higher than 0.5, which indicates oversubscribing incurs larger performance degradation on cg.B, thus this application should be excluded from oversubscribing. We have observed that most of the NPB applications had $E_v$ below 0.5 like sp.B. Table 4.7 shows the percentage of cases when the degradation ratio is less than the threshold that indicates whether a job pair is well-matched for oversubscribing. We see that 89% or more pairs are well-matched.

### Performance when batch and interactive jobs are oversubscribed

We employ the interactive job as opponent job. Contrarily, the victim job is same as the previous experiments.




(a) sp.B (b) cg.B

**Figure 4.4:** The degradation ratio with a batch job as victim and a mimic interactive job as opponent. Batch-batch cases are also displayed for comparison.

**Table 4.8:** Differences in degradation ratio between batch versus interactive cases and batch versus batch cases ("bxb"). The mean and median values among all tested job pairs are displayed.

| Difference value | mean | median |
|---|---|---|
| $E_{\mathrm{v}} - E_{\mathrm{v}}(\mathrm{bxb})$ | 0.04 | 0.02 |
| $E_{\mathrm{o}} - E_{\mathrm{o}}(\mathrm{bxb})$ | 0.15 | 0.16 |
| $E - E(\mathrm{bxb})$ | 0.19 | 0.19 |

Figure 4.4 shows the results when the victim job is sp.B or cg.B. The horizontal axis indicates the opponent job, and the bars indicate the degradation ratio. The

graph also includes degradation ratio when two batch jobs case mentioned in the previous section ($E_v(bxb)$, and so on).

We observe that the degradation ratio in batch versus interactive cases tends to be larger than in batch versus batch cases. In order to discuss the difference, Table 4.8 shows statistics of the difference. From these results, we also see that the increase in $E_o$ is larger than that of $E_v$.

We can consider the reason for this asymmetric increase as follows. In all the cases under oversubscribing, coexisting jobs interact with each other by evicting the other's data out of the cache. This effect is especially severe for a (mimic) interactive job, which repeats busy and idle periods. During idle periods, its data is mostly evicted out of the cache and thus it largely suffers from a large number of cache misses in the subsequent busy period. On the other hand, in batch versus batch cases, the victim have been continued accessing its data during execution thus it is not completely evicted from cache by coexisting job.

## 4.3 Performance modeling of oversubscribed applications

In Section 4.2, we have demonstrated that performance degradation introduced by oversubscribing largely depends on the pair of coexisting jobs when sequential jobs are oversubscribed. Thus, when we conduct oversubscribing scheduling in supercomputer systems, it is expected to choose compute nodes so that well-matched jobs with less performance degradation would share computing resources. However, it is unrealistic to collect the performance degradation ratio for all possible combinations of applications beforehand. Therefore, we have developed a model to predict the performance under oversubscribing. The model takes information of performance monitoring counters (PMCs) of each application run on dedicated resources as input.

### 4.3.1 Overview of prediction model

The model we propose is composed of two phases:

1. The model that detects applications whose performance will be degraded significantly (called "dangerous applications") by oversubscribing

2. The model that predicts degradation for a given combination of applications

In phase 1, we take PMCs during each application as input, and classify applications using a clustering algorithm to pick up dangerous applications that should not be under oversubscribing. With the not suitable applications for oversubscribing beforehand, we can detect the similar application. This idea is based on an assumption that applications with similar PMCs are likely to show a similar degradation ratio. Such dangerous applications are excluded and the rest applications are inspected by phase 2. Phase 2 estimates the prediction of degradation ratio for a given job pair. In the oversubscribing scheduling, we could use the estimation to determine nodes for newly submitted jobs so that well-matched applications share computing resources.

### 4.3.2 Phase 1: Dangerous application detection model

We adopt non-hierarchical cluster analysis as a statistical method to classify the applications. As a preliminary examination, we conducted a cluster analysis of applications using the PMCs on the dedicated cores as input. For this purpose, we have obtained around ten counters including the number of L1, L2 and L3 cache misses and branch instructions. The clustering results are shown in Figure 4.5, where each application execution corresponds to a point, and the PMC space is mapped to two dimensional space.

In the right area, we see a cluster consisting of cg benchmarks with several problem sizes. Since we have observed that cg.B has a large degradation ratio in the previous section, we judge that this cluster is a group of dangerous applications.

This phase 1 works as a pre-processing step of phase 2 that excludes dangerous applications. This has two objectives: one is to reduce computation costs in phase 2, which considers all pairs of applications. The other is to exclude outliers from linear regression analysis used in phase 2. Generally, linear regression tends to be weak to inputs outside the explanatory space, thus we expect the results are improved by excluding dangerous applications as outliers.

### 4.3.3 Phase 2: Degradation prediction model

#### Model specification

We developed a model to predict the performance when a given applications share resources in oversubscribing scheduling. This phase 2 is based on the linear logistic regression, which is known as a method for solving binary discriminant problems. The regression model returns the probability that binary value $b$ equals 1 (True).

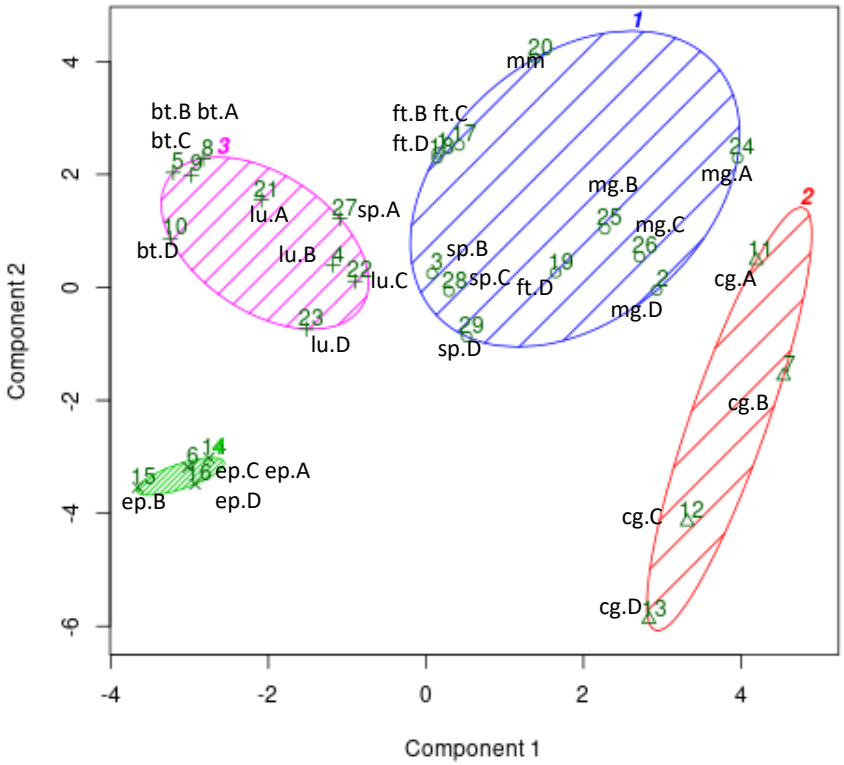



**Figure 4.5:** The result of non-hierarchical cluster analysis. In this analysis, we set the number of cluster four. Each circle corresponds the cluster.

Our strategy here is to estimate whether the pair of applications is well-matched or not, instead of estimating the degradation ratio directly.

In our model, the binary values correspond as follows:

1. If the degradation ratio of the victim application exceeds a certain threshold, the binary value $b$ is 1 (True)

2. Otherwise, binary value $b$ is 0 (False)

The threshold value of the degradation ratio $t$ is 0.5 as we assume that multiplicity is two. In the following, we employ this value. The input data are the PMCs of the victim and opponent applications, which are already available in phase 1. The output is the probability. Here, the model is formulated as follows:

$$X = \sum_{k=1}^{n} (\alpha_{k,\mathrm{vic}} C_{k,\mathrm{vic}} + \alpha_{k,\mathrm{opp}} C_{k,\mathrm{opp}}) \tag{4.7}$$

$$Pr(E_{\mathrm{v}} > t) = \pi = \frac{\exp(X)}{1 + \exp(X)} \tag{4.8}$$

where $X$ is a linear combination of PMC values, $n$ is the number of PMCs, $C_{k,\mathrm{vic}}$ is $k$-th PMC value of a victim application measured in the dedicated state, $C_{k,\mathrm{opp}}$ is that of an opponent application, and $\alpha_k$ is the regression coefficients.

To build a model, we must conduct selection of the appropriate PMCs. Among all PMCs obtained during dedicated executions, we have selected explanatory variables using a stepwise backward selection method. The selected PMCs are displayed in Section 4.3.3.

Let us now estimate the match between two applications A and B. First, we take application A as the victim and B as the opponent. Using the equation, we can predict the possibility of degradation of the victim (application A). Next, we swap applications A and B and repeat the computation to inspect application B. If both calculated values are less than the threshold described below, the pair of applications A and B is judged well-match, which is suitable for oversubscribing.

### Validation of the model

We use two different datasets for the validation of our model. One dataset consists of 800 records using 29 different applications shown in Table 4.1 (dataset1). The other dataset consists of 591 records, excluding the dangerous applications (dataset2). From the discussion in Section 4.3.2, we assume the dangerous applications are cg.A, cg.B, cg.C, and cg.D.

The number of selected counters are 31 and 11 for each dataset. We can build the model to capture the characteristic with fewer variables for dataset2. The selected counters for dataset2 are displayed in Table 4.9.

**Table 4.9:** Performance counters selected for dataset2 by stepwise method. The suffix V(O) show the value of victim(opponent). The counters related to memory(cache) and branch are mainly selected.

| Counter Name | Value |
|---|---|
| (Intercept) | -8.8e1 |
| PAPI_L3_TCM_V | -9.7e0 |
| PAPI_PRF_DM_V | -4.7e0 |
| PAPI_BR_UCN_V | -1.3e1 |
| PAPI_BR_TKN_V | 1.5e1 |
| PAPI_L2_TCW_V | -1.1e1 |
| PAPI_PRF_DM_O | 4.9e1 |
| PAPI_LST_INS_O | -2.4e1 |
| PAPI_L3_LDM_O | -2e1 |
| PAPI_BR_UCN_O | 1.7e1 |
| PAPI_BR_TKN_O | 1.4e1 |
| PAPI_BR_NTK_O | 1.1e2 |

Here we conduct 3-fold cross-validation. We compare the estimated results (well-matched or not) and the measured results as shown in Figure 4.3. Table 4.10

mentions the validation results. Generally, we observe the estimation by Phase 2 achieves good accuracy. When we compare dataset1 and dataset2, the latter shows even better accuracy, 96% or more. This result indicates the importance of excluding dangerous applications in Phase 1.

**Table 4.10:** Accuracy in three-fold cross validations.

| | Fold1 | Fold2 | Fold3 |
|---|---|---|---|
| dataset1 | 96% | 92% | 93% |
| dataset2 | 98% | 96% | 98% |

While the accuracy of the model is fairly good, we discuss cases with misprediction below. Here we let $b_t$ be true (measured) binary value and $b_p$ be the predicted value. Among the mispredicted cases, false negative cases with $b_t = 1$ and $b_p = 0$ are more critical than false positive cases with $b_t = 0$ and $b_p = 1$ for the oversubscribing scheduler. This is because the scheduler determines resource allocation considering $b_p$, and thus mispredicted jobs may suffer from large performance degradation by undesirable oversubscribing.

Table 4.11 shows the detail of all mispredicted cases with dataset2. Among them, cases 2, 305, 318, 462 are the false negative cases. Some of them can be corrected by changing configuration of the threshold. If we want to be more conservative, we can choose a smaller threshold. For example, if we set the threshold to 0.3 considering safety factor, cases 305 and 318 would be correct. Such configuration change can increase false positive cases, however, they are relatively harmless as discussed above.

**Table 4.11:** The detail of all wrong cases with dataset2

| Case No. | Fold | Victim | Opponent | True $E_{\mathrm{v}}$ | $b_t$ | $\pi$ | $b_p$ |
|---|---|---|---|---|---|---|---|
| 2 | 1 | ft.B | mg.D | 0.50 | 1 | 0.14 | 0 |
| 305 | 1 | ft.A | mg.D | 0.53 | 1 | 0.42 | 0 |
| 318 | 1 | ft.A | ft.D | 0.57 | 1 | 0.42 | 0 |
| 448 | 2 | mg.A | mg.D | 0.41 | 0 | 0.61 | 1 |
| 472 | 2 | mg.B | mg.D | 0.43 | 0 | 1.00 | 1 |
| 477 | 2 | mg.B | mm | 0.44 | 0 | 1.00 | 1 |
| 486 | 2 | mg.B | ft.D | 0.37 | 0 | 1.00 | 1 |
| 490 | 2 | mg.B | mg.A | 0.40 | 0 | 1.00 | 1 |
| 491 | 2 | mg.B | mg.C | 0.42 | 0 | 1.00 | 1 |
| 534 | 2 | sp.A | ft.D | 0.43 | 0 | 1.00 | 1 |
| 323 | 3 | ft.A | mg.B | 0.50 | 0 | 1.00 | 1 |
| 329 | 3 | ft.C | mg.D | 0.47 | 0 | 1.00 | 1 |
| 462 | 3 | mg.A | ft.D | 0.60 | 1 | 0.00 | 0 |

# 5

# System-wide Effect of Oversubscribing Scheduling

## Contents



Modern HPC systems face a fundamental tension between system efficiency and user productivity, particularly regarding waiting times, as discussed in Chapter 1. While previous research has primarily focused on application-level performance

impacts, comprehensive system-level evaluation remains crucial for validating OSub as a practical solution to this challenge.

## 5.1 Importance of system-wide evaluation

Chapter 4 demonstrated that performance degradation under OSub remains predictable and manageable through mechanisms like gang scheduling. However, this analysis was limited to controlled experiments with pairwise job combinations - scenarios where a single job shares resources with exactly one other job. While these experiments provided valuable insights into basic interference patterns, real HPC environments present significantly more complex scenarios where jobs may simultaneously share resources with multiple other jobs, particularly for large parallel applications spanning multiple nodes. Furthermore, as discussed in Chapter 3, traditional evaluation approaches focusing solely on application performance are insufficient for validating OSub effectiveness in production environments. System-level evaluation must address two critical aspects: First, the impact on system efficiency requires careful examination. While Chapter 4 showed acceptable performance trade-offs in controlled settings, system-wide effects such as resource fragmentation and scheduling overhead could potentially negate these benefits when scaled to full system operation. Second, user experience metrics, particularly waiting time reduction, demand rigorous evaluation. In modern HPC environments supporting interactive workflows, the ability to provide immediate resource access without significantly compromising system throughput represents a key operational requirement. In this way, high responsiveness of short-running jobs or interactive usage is critical and we call those jobs high-responsiveness-requesting jobs (*HRR jobs*) hereafter in this chapter.

To address these requirements, we employ simulation-based analysis using real workload traces from production supercomputer systems. This approach enables us to:

- Evaluate complex resource sharing scenarios involving multiple concurrent jobs
- Analyze system behavior under various configurations and workload compositions
- Quantify both efficiency impacts and user experience improvements

- Maintain experimental reproducibility while preserving real-world workload characteristics

This comprehensive evaluation strategy aligns with the methodological framework presented in Chapter 3, providing crucial validation of OSub effectiveness across both system efficiency and user productivity dimensions.

## 5.2 Methodology for simulating oversubscribing scheduling

The purpose of this chapter is to demonstrate the benefits of oversubscribing scheduling in system-level. For this purpose, we conduct experiments using our new simulator, node conscious oversubscribing scheduler simulator (NCS). NCS is designed to simulate oversubscribing job scheduling considering speed down of jobs considering multiplicity. The source code is available at the author's Github site [100].

NCS takes a system description configuration file and an SWF file [101], [102], will be described in Section 5.2.4, for information about a set of jobs as input. The configuration file contains the number of nodes, number of cores per node, and memory capacity per node. It also contains maximum multiplicity $M$ and a scheduling policy (Section 5.2.3).

Then NCS simulates the scheduling of jobs described in the SWF file. Oversubscribing is allowed if multiplicity does not exceed the maximum multiplicity $M$ at any cores. During the simulation, unlike the typical scheduling simulators, NCS needs to simulate performance degradation of jobs under oversubscribing.

In the rest of this section, we show our system model in the simulation and basic scheduling method in Section 5.2.1. Then we discuss how the performance degradation due to oversubscribing is estimated (Section 5.2.2). And then we describe the detailed scheduling algorithm of NCS, which includes modification of a well-known algorithm, EASY Backfilling to support oversubscribing (Section 5.2.3).

### 5.2.1 System model and basic scheduling

Hereafter, we use the notation such as *N16C8* (capitalized) to indicate a system configuration with 16 nodes, each of which has 8 cores. The notation *n2c8* (uncapitalized) corresponds to a job configuration that requests 2 nodes in total and occupies 8 cores in each node.

NCS internally maintains the status of the target system during the simulation as shown in Figure 5.1, which shows an example of $N2C4$ system with $M = 2$. Each core has $M = 2$ slots utilized to accommodate running jobs. Red slots have been occupied by running jobs. When NCS takes a new job submission, it needs to decide on nodes and cores for the job. This allocation is done in a hierarchical style as follows. In Figure 5.1, NCS is going to schedule a new n1c1 (serial) job. First, NCS determines an appropriate node for the job, which has sufficient empty slots and free memory. If there are several candidates for nodes, NCS consider the total occupied slots per node, and selects a node with the least occupied slots, which is Node 1 in the figure with 3 total occupied slots. If there are multiple candidate nodes, NCS chooses the lower number node. Second, NCS determines a core with the least occupied slots, Core 3. For parallel jobs, NCS determines nodes and cores similarly. While a detailed explanation is skipped in the figure, NCS also maintains memory amount, and thus the resource allocation may fail for memory capacity limitation. In such cases, other nodes/cores are examined.

If the above process fails, since there are not enough slots or memory for the new coming job, it is put in the waiting queue. Jobs in the queue are examined later when the status of slots is changed, in other words, when any job terminates (Section 5.2.3).

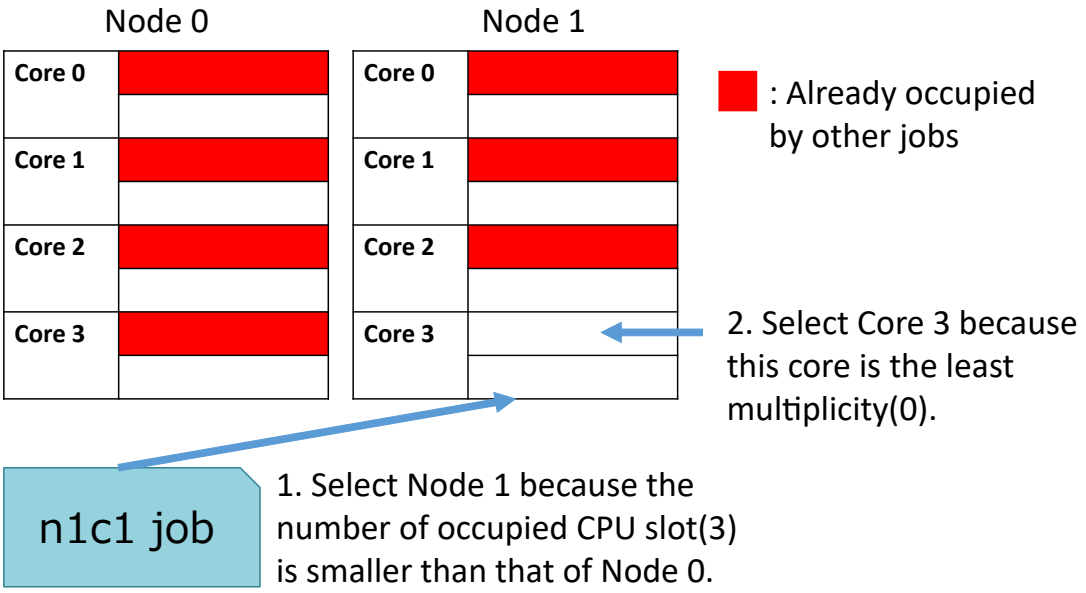



**Figure 5.1:** An example of oversubscribing scheduling on a N2C4 system. Two slots are prepared per core since the maximum multiplicity $M$ is two.

### 5.2.2 Simulation of job progress

Unlike typical job scheduling simulators, NCS for oversubscribing scheduling has to consider jobs' performance degradation. Here we are based on assumptions that degradation is determined by multiplicity $m$. We also need to consider that $m$ on each core changes dynamically.

Let us explain the behavior of NCS using a small system of N1C4 and $M = 2$ as shown in Figure 5.2. Note that we assume degraded speed under oversubscribing changes proportionally in this section, thus when $m = 2$, the degraded speed is $100/2 = 50\%$.

We consider two jobs, job0 = n1c4 and job1 = n1c2. Also, job0 is submitted at $t = 0$ and the original execution time, included in the input data, is 30. Job1 is submitted at $t = 10$ and the original execution time is 10.

- $t = 0$: Job0 is submitted and starts immediately.
- $0 < t < 10$: Job0 uses four dedicated cores. During this period, job0 does not suffer from degradation.
- $t = 10$: Job1 is submitted and starts immediately because there is room in the CPU core slots. Here, cores 0 and 1 are used.
- $10 < t < 30$: Both job0 and job1 share cores 0 and 1. Since we assume that the performance of each job is determined by the slowest threads, the performance of job0 and job1 becomes 1/2 of the original. Although cores 2 and 3 are used only by job0, the speed of entire job0 is halved.
- $t = 30$: Job1 finishes at this time, taking $10/(1/2) = 20$ time considering performance degradation.
- $30 < t \leq 40$: Job0 uses four dedicated cores again and the performance is recovered. Job0 finishes at $t = 40$.

In total, while the original execution time of Job0 is 30, its execution time with oversubscribing is 40. Note that in the instance, since Core 2 and 3 are spped down due to multiplicity, wasted CPU time occurs. The wasted CPU time approximate to 0.50 [speed] * 2 [core] * 10 [s] = 10 [CPU Core time]. However, it would not achieve the effectiveness sufficiently with avoiding speed down cores completely. Thus, system throughput and responsiveness is trade-off.

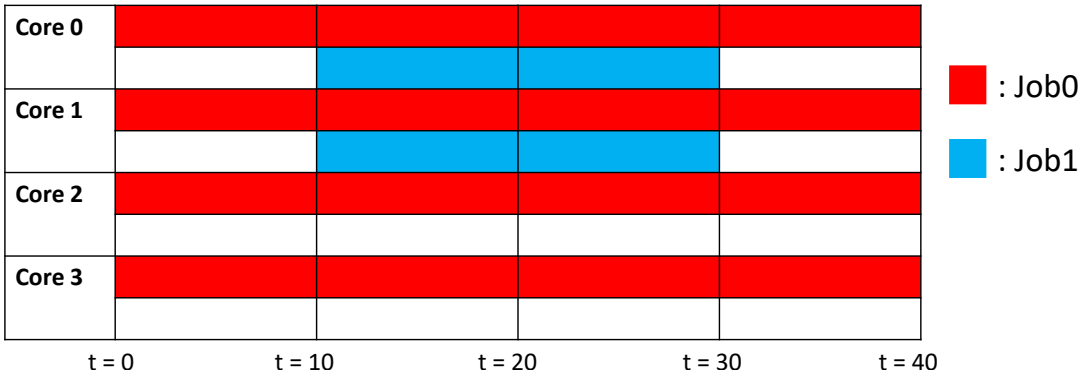



**Figure 5.2:** An example of time-series of oversubscribing scheduling. The speeds of Job0 and Job1 are degraded while they share CPU cores.

### 5.2.3 Scheduling considering oversubscribing

With our oversubscribing policies, a job may suffer from waiting time due to a lack of enough slots or memory. Thus NCS needs to maintain a job waiting queue like conventional schedulers. We have implemented two scheduling algorithms on NCS based on well-known algorithms, first-come first-served (FCFS) and EASY backfilling that are revised to support oversubscribing. This added implementation mainly focuses on logical consistency to execute oversubscribing scheduling to avoid any problem.

First, FCFS is easy to support oversubscribing. The main modification is the treatment of requesting time. Execution time may exceed the original requesting time owing to oversubscribing, which leads the unintentional termination. To avoid it, the scheduler scales the requesting time proportionally when oversubscribing. The functions except for it are similar to the conventional schedulers: the scheduler takes a job from the head of the waiting queue and tries to allocate resources for it. If there are not enough slots, considering multiplicity, the job is stuck. After the job is successfully scheduled, the scheduler can take the next job.

On the other hand, backfilling algorithms needs to be modified considerably for the following reason. After the scheduler makes future schedules of several jobs, a new coming job, which is typically a short-running job may be scheduled earlier (backfilled) than the existing jobs. Among backfilling algorithms, we adopt EASY backfilling (EB hereafter [43]), where backfilling is allowed only if *the start times of existing jobs are not delayed*. Note that the decision of backfilling requires information on the execution time of jobs, which may change due to oversubscribing.

We discuss how EB is modified for oversubscribing while keeping the above-mentioned condition. Figure 5.3 (a) is a simple case, where Job0, Job1, and Job2

have already been scheduled. When Job3 (n2c1) is submitted at "Current Time" in the figure, the scheduler temporarily allocates cores for it, core1 on node0 and core0 on node1 in this case. Then the scheduler estimates the finish times of Job3 if it is started immediately, considering the performance degradation. And it checks whether the start times of existing jobs are changed or not. If the degraded execution of Job3 is shorter than $T$, we can estimate that the start time of the existing Job2 is not changed. Consideration about Job0 is simple in this case. Since the speed of Job0 is already degraded to 50% due to the existence of Job1, starting Job3 does not degrade the speed of Job1 furthermore. From the above consideration, Job3 can be started immediately.

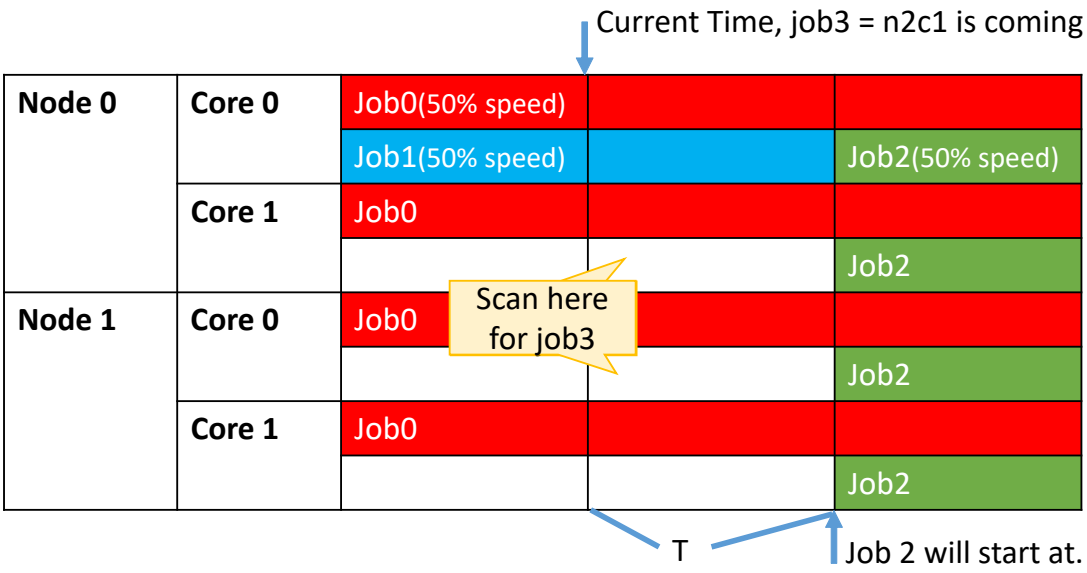
Figure 5.3 (b) shows a bit complicated case. Here, Job0, Job1, Job2, and Job3 have been scheduled. At "Current Time", Job4 (n2c1) is newly submitted, and core1 on node0 and core0 on node1 are allocated temporarily. In the case, starting Job4 would increase the multiplicity of Job1 and cause the performance degradation. To judge the feasibility of backfilling, we should check if either Job1 or Job4 finishes until Job3's expected start time. In this way, the case that new coming job causes the performance degradation of existing other jobs requires the large scheduling cost, thus we abandon backfilling. In the above case, we just additionally checked only Job1. Generally, however, it is even more complicated since invoking a new parallel job may cause multiple victims to be affected, which has been implemented in NCS.

### 5.2.4 Dealing with the standard workload format (SWF)

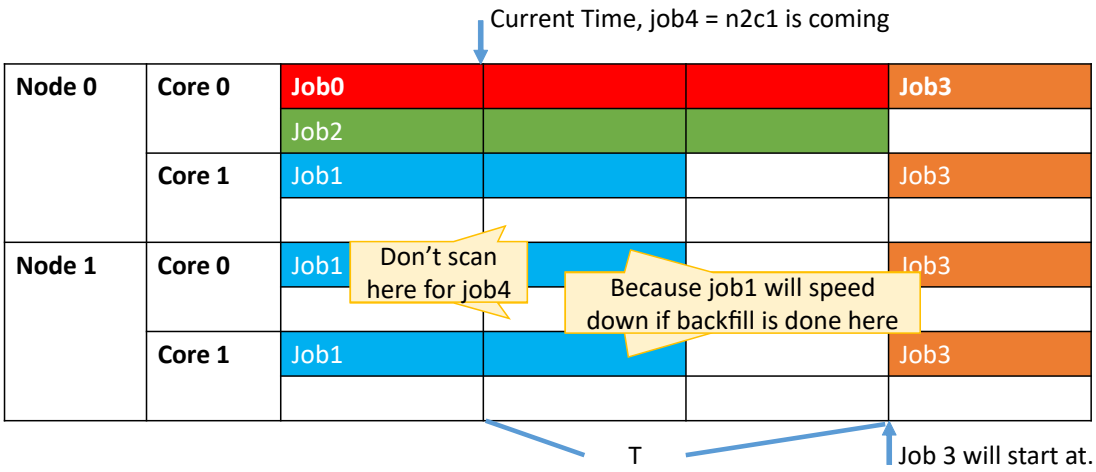
The standard workload format (SWF) is a widely-used format of the workload log[101]. SWF is used in the Parallel workload archive (PWA)[102], which provides the workload logs collected on several supercomputers and cluster systems. SWF includes jobs data of submission time, the number of requested processors (cores), requested memory size, running time, user ID, job queue ID, etc. Researchers can replay the execution of those jobs on top of their scheduling algorithm.

For our purpose, however, we have found a lack of required information: (1) job classification that shows whether a job is interactive or not, (2) transition of actual processor/memory utilization during job execution. Also while the data includes the numbers of requested processor cores, (3) the numbers of requested nodes are missing.

We do not need (1) job classification in simulation with NCS, since NCS does not distinguish interactive jobs from other jobs. On the other hand, several workload traces have been collected on systems that have a separated queue for interactive usage. Using those traces, we scan job queue ID information and can regard jobs in the interactive queue as interactive, while other jobs are regarded as batch jobs.




(a) Case 1




(b) Case 2

**Figure 5.3:** Behaviors of EASY backfilling with oversubscribing.

With regard to the issue (2), the SWF format contains a field of "Averaged CPU time used". In many traces, however, the fields have invalid data. Even if the fields are filled with valid data, they are insufficient to simulate fluctuation of resource utilization of interactive jobs as in Figure 2.3 (b). We make an assumption that processor utilization of a job is constant regardless the job is batch or interactive.

Concerning the issue (3), we need to reinterpret the "requested number of processors" field, since recent supercomputers consist of multiple multi-core nodes[1].

[1]This issue needs to be considered not only in our oversubscribing scheduling but in generic scheduling research

For example, if a system has multiple eight-core nodes. A job that requires 16 cores may occupy two nodes at least, and 16 nodes at most. Thus we need to solve the uncertainty for each job.

Hereafter, we use the notation such as *N16C8* (capitalized) to indicate a system with 16 nodes, each of which has 8 cores. The notation *n2c8* corresponds to a job that requests 2 nodes in total and occupies 8 cores in each node.

From the requested number of cores in SWF, we have the following policies to determine the number of nodes:

1. To select the minimum number of nodes: Basically, we consider maximizing the number of cores used in a node. If the system is N16C8 and the job requires 16 cores, two nodes are used, and n2c8 allocation is selected.

2. To allocate the same number of cores among nodes: For example, a 15-core job does not fit into 2 nodes as 15/2 is not an integer. In this case, n3c5. n5c3 and n15c1 are possible. Among these candidates, n3c5 has the highest priority, considering the policy (1).

3. To satisfy memory capacity constraint: The above choices may be further reconsidered if the memory capacity of nodes is insufficient. If allocating 5 cores per node leads out of memory, we check the feasibility of n5c3 and n15c1 in turn.

## 5.3 Experimental configuration

### 5.3.1 Comparison target system: multiple-queue system versus single-queue system

We evaluate the advantages of the oversubscribing system by comparing it with a system with another configuration.

While many production supercomputers do not use oversubscribing, some systems prepare several dedicated nodes for HRR jobs separated from the normal nodes for non-HRR jobs (normal jobs) [30], [103]. On such a system (which we call *multiple-queue* (MQ) system), it has been expected that responsiveness of HRR jobs are kept better. On the MQ system, we introduce a parameter $R$, which means the ratio of dedicated nodes for HRR jobs to the entire system, which is configured for each simulation. For example, if the system is N100C10 and $R = 5\%$, the N5C10 system is used for HRR jobs, N95C10 for the rest of the jobs.

On the other hand, oversubscribing system, which we call single-queue (SQ) system, do not distinguish between HRR jobs and non-HRR jobs. All jobs may be executed on any nodes, allowed oversubscribing. On the oversubscribing system, the maximum multiplicity $M$ is varied within $1 \leq M \leq 8$.

For both system configurations, the evaluation uses EASY backfilling (EB) scheduling algorithm that is modified for oversubscribing, as described in Section 5.2.3.

The parameter $R$ in MQ systems represents the ratio of nodes dedicated to interactive jobs relative to the total system size. In production environments, $R$ is configured by system operators based on observed system usage patterns. While $R$ can be adjusted during system operation, frequent changes are difficult due to operational overhead. Therefore, it is hard to maintain optimal $R$ values as workload patterns change over time.

### 5.3.2 Definition of evaluation criteria

To comprehensively evaluate scheduling systems with oversubscribing capabilities, we need multiple metrics that capture different aspects of system performance. We consider three key categories of metrics:

#### Responsiveness metrics

Job waiting time $T_{\mathrm{w}}$ serves as a direct measure of system responsiveness. We examine both:

- The maximum waiting time across all jobs (smaller is better)
- The percentage of jobs that experience waiting time (smaller is better)

These metrics are particularly important for HRR jobs where immediacy of execution is critical.

#### Job performance metrics

For individual job performance, we reconsider the traditional slowdown metric to account for oversubscribing effects. With oversubscribing, the running time $T_{\mathrm{r}}$ increases due to resource sharing. Using this directly in slowdown calculations could underestimate the overhead. Therefore, we introduce oversubscribing conscious slowdown $S_{\mathrm{OSub}}$ which uses $T_{\mathrm{a}}$, the running time with multiplicity of one:

$$S = (T_e - T_s)/T_r = (T_r + T_w)/T_r \quad (5.1)$$

$$S_{OSub} = (T_e - T_s)/T_a = (T_r + T_w)/T_a \quad (5.2)$$

where $T_s$, $T_e$, and $T_w$ represent job submission time, end time, and wait time respectively.

**System efficiency metrics**

To evaluate overall system efficiency, we examine percentile-based metrics of cumulative CPU time. Traditionally, the 90th and 95th percentiles are used as key indicators of system performance, as they effectively capture how efficiently the system processes the majority of the workload. While the 100th percentile (maximum completion time) can also be examined, it has several limitations:

- It can be insensitive to configuration differences when the final portion of the workload becomes fragmented
- It may overemphasize the impact of a single long-running job that happens to be scheduled last

Therefore, we primarily focus on the 90th and 95th percentiles while also reporting the 100th percentile for completeness.

These three categories of metrics together provide a balanced view of:

- Individual job performance through $S_{OSub}$
- System responsiveness through waiting time analysis, especially for interactive jobs
- Overall system efficiency through multi-percentile analysis

This comprehensive evaluation framework allows us to assess the effectiveness of different scheduling approaches while accounting for the unique characteristics of oversubscribing systems.

### 5.3.3 Target workload traces

In our simulation, we use workload traces in the SWF format that are publicly available, UniLu-Gaia-2014-1 [103] and KIT-FH2-2016-1 [104]. We choose these traces since they are collected on systems with multiple job queues.

**UniLu-Gaia-2014-1**

This data set (UniLu hereafter) contains three months data from the Gaia cluster at the University of Luxemburg [103]. It contains multiple queues prepared for interactive jobs and batch jobs, respectively. Thus we regard interactive jobs as HRR jobs.

Note that HRR jobs are originally interactive jobs, but they are regarded like (short) batch jobs in the simulation, whose CPU utilization is constant during resource allocation, as described before.

Table 5.1 shows the system information and the workload information. The ratio of HRR jobs is 3.4%, thus the ratio of HRR nodes $R$ on the MQ system is configured to be around it. In the evaluation, we use $R = 1, 3, 5, 7, 10, 12$ [%].

**Table 5.1:** UniLu-Gaia-2014-1

| | |
|---|---|
| System Configuration | N150C12 |
| # of Nodes | 150 |
| # of Cores per Node | 12 |
| Workload Characteristic | |
| # of Jobs | 51,871 |
| HRR Jobs | 1,762(3.4%) |
| Maximum Degree of Parallelism | 516 |
| HRR Jobs | 12 |
| Maximum Execution Time[s] | 1,800,012 |
| HRR Jobs[s] | 43,507 |
| # of Users | 82 |

**KIT-FH2-2016-1**

This data set (KIT hereafter) contains one and a half years worth of accounting records from the ForHLR II system located at the Karlsruhe Institute of Technology in Germany[104].

We observed that two queues are used differently depending on the job size; one queue is used for smaller scale jobs. We regard jobs in the queue as HRR jobs.

Table 5.2 shows the system information and the workload information. We evaluate the MQ system with $R = 0.5, 1, 3, 5$ [%].

## 5.4 Simulated results of oversubscribing scheduling

This section evaluates the effects of oversubscribing scheduling using two workload traces on our NCS simulator. While we execute all jobs in the workload traces,

**Table 5.2:** System and workload information for KIT-FH2-2016-1

| | |
|---|---|
| System Configuration | N1152C20 |
| # of Nodes | 1,152 |
| # of Cores per Node | 20 |
| Workload Characteristic | |
| # of Jobs | 114,347 |
| HRR Jobs | 2,349(2.1%) |
| Maximum Degree of Parallelism | 22,960 |
| HRR Jobs | 48 |
| Maximum Execution Time[s] | 604,800 |
| HRR Jobs | 259,200 |
| # of Users | 161 |

we focus more on HRR jobs. Basically, HRR jobs are smaller in size in terms of parallelism and running time, which are defined in detail in 5.3.3.

### 5.4.1 The responsiveness of HRR jobs

First, we evaluate the waiting time of jobs. Figure 5.4 (a) shows the maximum waiting time among all HRR jobs. If the value is zero, no HRR job suffers from waiting time, which is the best case for the users.

In the MQ systems, waiting times for HRR jobs get better as $R$ increases since the system is configured preferably for HRR jobs. When $R$ reaches 10%, all jobs can be processed immediately. Also in the SQ systems, the responsiveness improves with more aggressive oversubscribing with larger $M$. When $M$ reaches four, all jobs, including normal jobs though not displayed in the figure, can commence with zero waiting time.

Figure 5.4 (b) shows results with jobs in the KIT trace. The same trade-off patterns are observed in the MQ systems - increasing $R$ leads to better HRR job responsiveness, with zero waiting times achieved at $R = 5\%$. In the SQ systems, while the general trend of improved responsiveness with higher $M$ values holds true, complete elimination of waiting times proves more challenging than in the UniLu case, with a small maximum waiting time of about 6 minutes persisting even at $M = 8$.

Table 5.3 provides a more detailed view by showing the percentage of HRR jobs that experience any waiting time. This metric offers complementary insight to the maximum waiting time analysis. For the KIT trace, while the maximum waiting time cannot be completely eliminated even with $M = 8$, Table 5.3 reveals that this affects only 0.042% of HRR jobs - equivalent to a single job in the entire trace. This suggests that the SQ system maintains excellent responsiveness

for the vast majority of interactive workloads, even when it cannot completely eliminate waiting times for all jobs.

From the discussion, oversubscribing scheduling can immediately provide the resources for HRR jobs as well as a multiple-queue system with larger $R$.

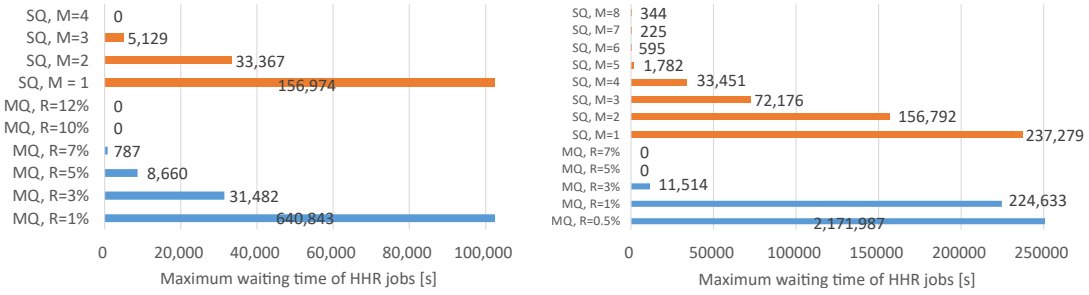
**Table 5.3:** Percentage of HRR jobs that experience waiting time in MQ and SQ systems. Lower percentages indicate better system responsiveness for interactive workloads.

(a) UniLu trace

| Case | Ratio | Case | Ratio |
|---|---|---|---|
| SQ, M=4 | 0% | MQ, R=12% | 0% |
| SQ, M=3 | 0.11% | MQ, R=10% | 0% |
| SQ, M=2 | 2.1% | MQ, R=7% | 0.23% |
| SQ, M=1 | 17% | MQ, R=5% | 0.85% |
| | | MQ, R=3% | 7.2% |
| | | MQ, R=1% | 72% |

(b) KIT trace

| Case | Ratio | Case | Ratio |
|---|---|---|---|
| SQ, M=8 | 0.042% | MQ, R=7% | 0% |
| SQ, M=7 | 0.042% | MQ, R=5% | 0% |
| SQ, M=6 | 0.042% | MQ, R=3% | 1.1% |
| SQ, M=5 | 0.12% | MQ, R=1% | 18% |
| SQ, M=4 | 0.93% | MQ, R=0.5% | 30% |
| SQ, M=3 | 30% | | |
| SQ, M=2 | 18% | | |
| SQ, M=1 | 41% | | |




(a) For UniLu trace

(b) For KIT trace

**Figure 5.4:** Maximum waiting time for HRR jobs when MQ and SQ systems. Since some cases exceed the upper bound of axis, the value for each case is included near the bar.

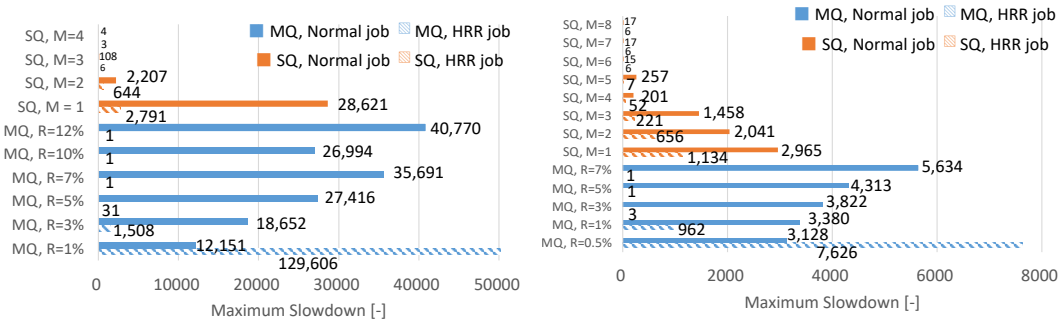
### 5.4.2 Evaluation of slowdown

Next, we evaluate the slowdown of jobs mainly for normal jobs. Here we use $S_{OSub}$, which has been revised in Section 5.3.2. Figure 5.5 (a) shows $S_{OSub}$ on the MQ and SQ systems with UniLu trace.

Slowdowns of normal jobs are largely different between the two systems. In the MQ systems, the values for normal jobs get worse as $R$ increases. On the other hand, the slowdown values of all HRR jobs are one with R = 10 [%], which corresponds to the fact that the waiting time is zero. This indicates that when the

MQ system are configured for efficient execution of HRR jobs, it is unfavorable for normal jobs, introducing a trade-off. When $R$ is 10% with UniLu, the slowdown of normal jobs is quite terrible. We also observe a similar tendency with KIT (Figure 5.5 (b)); all HRR jobs can be started without waiting when $R$ is 5% as shown in Figure 5.4 (b), however, the response of normal jobs is awful.

On the other hand, with oversubscribing scheduling, slowdown gets better both for normal and HRR with larger $M$. HRR jobs slowdown improves, however, unlike on the MQ systems, the slowdown does not reach one. This is due to performance degradation caused by oversubscribing. While we suffer from this overhead, we can conclude that normal HRR and normal jobs can coexist efficiently on the SQ systems.




(a) For UniLu trace

(b) For KIT trace

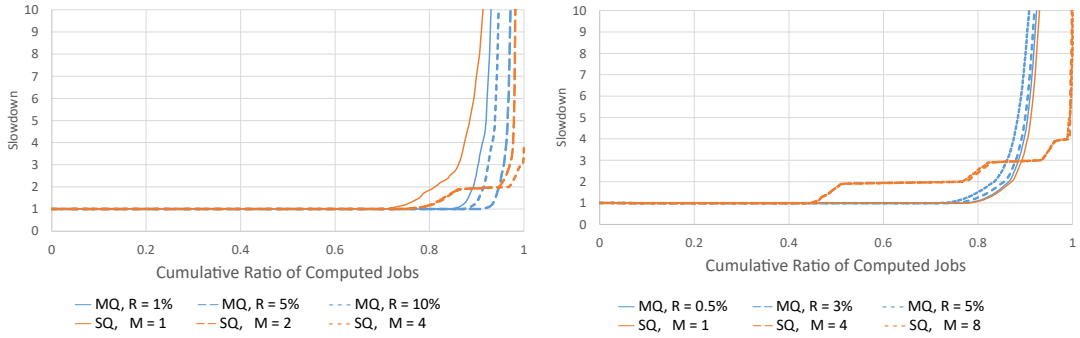
**Figure 5.5:** Maximum oversubscribing conscious slowdown for normal and HRR jobs for MQ and SQ systems. Filled bars represent normal jobs, striped bars HRR jobs. The label shows the value. Since some cases exceed the upper bound of axis, the value for each case is included near the bar.

### 5.4.3 Detailed evaluation of slowdown

In the previous section, we examined the maximum value of slowdown. In order to evaluate tendency in detail, Figure 5.6 shows distribution of slowdown values among jobs. The X-axis is the cumulative ratio of jobs and the Y-axis is the slowdown value. Here HRR and normal jobs are mixed.

The cumulative ratio on the SQ system reaches one faster than that of the MQ case, which means that the maximum slowdown can be suppressed by oversubscribing. Now we compare MQ system with $R = 5\%$ and SQ system with $M = 4$ with UniLu trace. While slowdown with oversubscribing is 4 at maximum, on the MQ system, the line is still at 96.2%. Also, 650 jobs (1.3%) have $S_{OSub} \geq 100$, though we cannot see it in the figure. The KIT trace shows similar results. On MQ system with $R = 3\%$, 3.2% jobs suffer from $S_{OSub} \geq 100$.

With the KIT data trace, while "OSub, M=8" reaches one at $S_{OSub} = 6$, other lines are still around 90%, which means around 10% jobs suffer from $S_{OSub} > 6$. This is explained by the that the KIT trace represents a more crowded system than the UniLu trace. Thus oversubscribing scheduling is preferable to improve responsiveness on such a crowded system with a proper configuration of $M$. Also oversubscribing tends to be fair for all the jobs in the aspect of slowdown.




(a) For UniLu trace

(b) For KIT trace

**Figure 5.6:** Distribution chart of Slowdown for all jobs when multiple-queue system and oversubscribing system.

### 5.4.4 Investigation of individual jobs

So far we have evaluated different scheduling methods statistically. Contrarily, this section picks up some individual jobs with specific characteristics from the UniLu data set shown in Table 5.4.

Job 2819 is a massively parallel job with 200 cores, and the slowdown is largely improved from 11.0 to 2.00 by introducing oversubscribing, which is an example of how oversubscribing works effectively for massively parallel jobs. Job 918 has a short execution time. Although the class is normal, we think that the responsiveness is important like HRR jobs. In this experiment, there are no jobs with a quite painful slowdown in the MQ system accidentally. However, we expect oversubscribing can reduce the waiting time for this type of job. Job 9121 is a job with a long execution time. We observe its slowdown is improved from 7.07 to 2.00.

### 5.4.5 Evaluation of the overall system efficiency

Finally, we show the overall system efficiency. Table 5.5 shows the 90 th, 95 th, and 100th percentiles of computed jobs. There is not a significant difference between cases. The increase in 100th percentile of the worst case ("OSub, M=2") is very

**Table 5.4:** Examples of improved jobs under oversubscribing. MQ system is with $R = 5\%$ and SQ is with $M = 4$. $T_{spec}$ is the specified time and $P$ is the parallelism. Rest variables are defined in Section 5.3.2

| Job ID | $T_a$ | Job Class | $T_{spec}$ | $P$ | $T_r$ | | $T_w$ | | $S_{OSub}$ | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | MQ | SQ | MQ | SQ | MQ | SQ |
| 2819 | 2,938 | normal | 36,000 | 200 | 2,938 | 5,876 | 29,289 | 0 | 11.0 | 2.00 |
| 918 | 119 | normal | 300 | 4 | 119 | 119 | 0 | 0 | 1.00 | 1.00 |
| 9121 | 20,920 | normal | 36,000 | 12 | 20,920 | 41,840 | 12,702 | 0 | 7.07 | 2.00 |

tiny, 2.5%. It is tolerable. As mentioned in Section 5.2.2, although oversubscribing scheduling achieves high responsiveness it may invite a degradation of system efficiency. However, these results suggest that oversubscribing scheduling, if ever, hardly affects in terms of system efficiency.

With the KIT trace, a similar consideration can be derived with from Table 5.6. The increase in 100th percentile of the worst case ("OSub, M=8") is 1.2% and negligible.

**Table 5.5:** 90th, 95th, and 100th percentiles of computed amounts (CPU time) required by all jobs for MQ and SQ systems with UniLu-Gaia-2014-1 workload trace. Oversubscribing results with $M > 4$ are the same with $M = 4$.

| | 90th percentile [hour] | 95th percentile [hour] | 100th percentile [hour] |
|---|---|---|---|
| SQ, M=4 | 1883 | 2015 | 2139 |
| SQ, M=3 | 1889 | 2015 | 2139 |
| SQ, M=2 | 1874 | 2016 | 2193 |
| SQ, M=1 | 1850 | 2015 | 2139 |
| MQ, R=12% | 1859 | 2015 | 2139 |
| MQ, R=10% | 1833 | 1989 | 2139 |
| MQ, R=7% | 1833 | 1989 | 2139 |
| MQ, R=5% | 1833 | 1989 | 2139 |
| MQ, R=3% | 1832 | 1989 | 2139 |
| MQ, R=1% | 1832 | 1989 | 2158 |

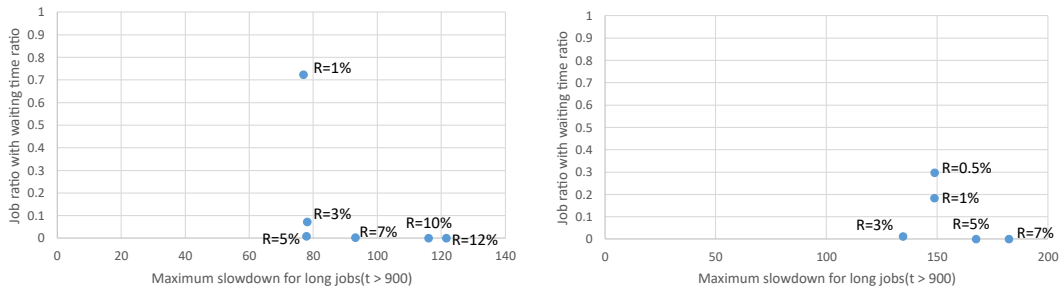
### 5.4.6 Optimal configuration for multiple queue systems

This section discusses the optimal configuration for multiple queue system. Figure 5.7 examines the maximum slowdown for long normal jobs versus job count ratio with waiting time. This shows the trade off tendency - as increase in $R$, the job count decreases and the slowdown increases, which implicates the parato space. The optimal configuration depends on the policy of the operators. In this time, with assumption that most prominent requirement is responsiveness for HRR jobs, we can think of optimal configuration considering maximum waiting time as shown

**Table 5.6:** 90th, 95th, and 100th percentiles of computed amounts (CPU time) required by all jobs for MQ and SQ systems with KIT-FH2-2016-1 workload trace.

| | 90th percentile [hour] | 95th percentile [hour] | 100th percentile [hour] |
|---|---|---|---|
| SQ, M=8 | 13566 | 13728 | 14147 |
| SQ, M=7 | 13568 | 13728 | 14101 |
| SQ, M=6 | 13569 | 13728 | 14108 |
| SQ, M=5 | 13568 | 13728 | 14102 |
| SQ, M=4 | 13568 | 13728 | 14109 |
| SQ, M=3 | 13566 | 13727 | 14072 |
| SQ, M=2 | 13566 | 13726 | 14047 |
| SQ, M=1 | 13566 | 13726 | 13982 |
| MQ, R=7% | 12838 | 13365 | 13982 |
| MQ, R=5% | 12837 | 13359 | 13982 |
| MQ, R=3% | 12856 | 13359 | 13982 |
| MQ, R=1% | 12835 | 13354 | 13982 |
| MQ, R=0.5% | 12834 | 13351 | 13982 |

in Figure 5.4, thus the configurations are $R = 10\%$ for UniLu trace and $R = 5\%$ for KIT trace, while $R = 1\%$ for UniLu trace and $R = 0.5 - 1\%$ for KIT trace are obviously misconfiguration.




(a) For UniLu trace

(b) For KIT trace

**Figure 5.7:** Maximum Oversubscribing conscious slowdown for normal long jobs and job count ratio with waiting time in multiple-queue system.

### 5.4.7 Summary of evaluation

In this section, we have evaluated oversubscribing scheduling system by comparing it with a multiple-queue system with separated queues, using two actual workload traces. We demonstrated that oversubscribing largely decreases response time, which is critical for HRR jobs. It could be achieved on the multiple-queue system, if it is configured with plentiful dedicated HRR nodes, however, it makes the slowdown

of normal jobs worse. We observe that oversubscribing improves turn around time, which is not seen in the multiple-queue system. Thus the oversubscribing scheduling works efficiently both for HRR jobs and normal jobs.

# 6

# Effect of Oversubscribing Scheduling for Interactive Jobs

## Contents



## 6.1 Introduction to considering interactive jobs

HPC systems have witnessed a significant increase in the usage of interactive jobs in recent years. As discussed in Section 1.1.2, with the growing importance of AI/ML

development, HPC resources have become indispensable for tasks such as model prototyping, hyperparameter optimization, and experimental validation. These development workflows typically require immediate resource access and frequent iterations, leading to a substantial increase in interactive job submissions.

Oversubscribing has demonstrated remarkable success in other computing domains, particularly in operating systems and cloud environments, as examined in Section 3.2.2. A key factor in this success has been their fundamental premise: computational workloads naturally exhibit intermittent resource utilization patterns, enabling efficient resource sharing through time-division multiplexing. Interactive jobs in HPC environments, with their characteristic pattern of alternating between computation and user interaction periods, theoretically align well with this premise.

However, while interactive jobs are expected to show intermittent resource usage patterns similar to those successfully managed in OS and cloud environments, their actual behavior patterns in HPC contexts remain insufficiently analyzed. Understanding these patterns is crucial for validating OSub's effectiveness in HPC environments, particularly as systems face increasing proportions of interactive workloads, as outlined in Section 3.5.

This chapter examines OSub's effectiveness in managing interactive workloads through two main approaches. First, we conduct a detailed analysis of actual interactive job behavior in production HPC environments, focusing on resource utilization patterns and user interaction characteristics. Second, we evaluate system performance under various interactive workload scenarios through both simulation studies and actual system measurements. This comprehensive evaluation aims to validate OSub's capability to efficiently support the growing demands of interactive computing in modern HPC environments while maintaining system efficiency.

## 6.2 Understanding interactive jobs

### 6.2.1 Characteristics of interactive jobs

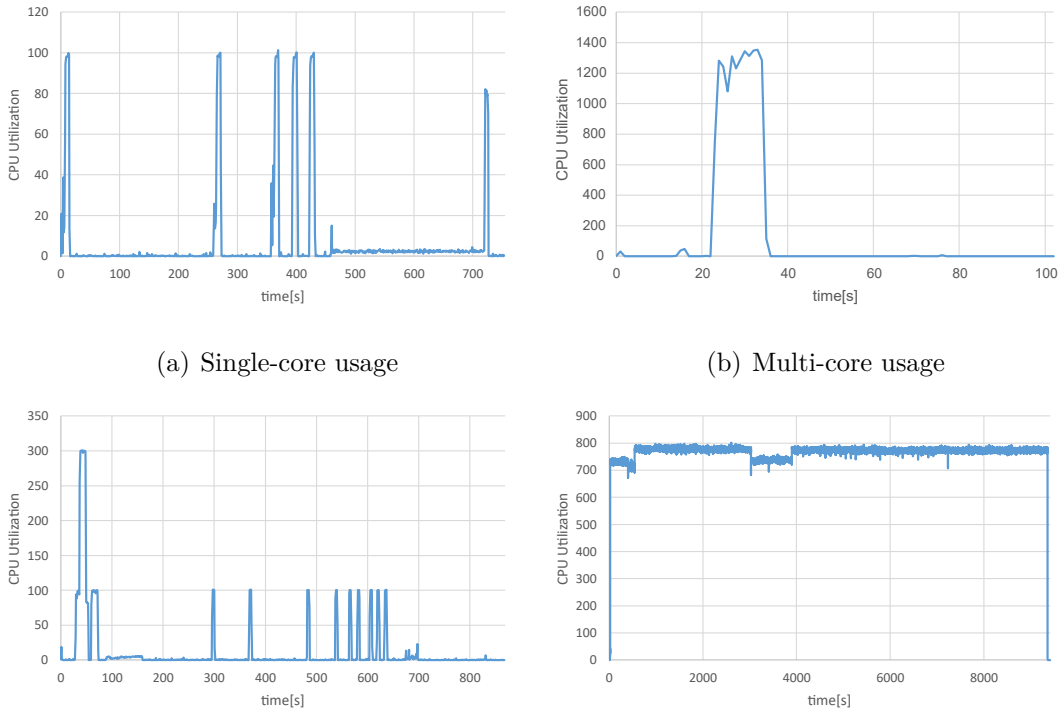
Interactive jobs include visualization, debugging of software, development loop of the AI/ML applications, and so on. Some usages can be via web browsers[11], [55], [105], [106]. We view their characteristics and requirements are different from those of batch jobs as follows.

(1) Interactive jobs should be started immediately when users request them. Also the frequency of requests can be more frequent on weekdays and less on weekends, which shows the burstiness.

(2) During the execution, the actual utilization of resources, including CPU cores, tends to fluctuate. This is because users often interact with resources by executing some processes, seeing their results, and determining what to do next. Figure 2.3 shows a simplified illustration of such behaviors.

(3) Traditionally, interactive jobs like visualization tend to occupy only small resources. This tendency is, however, changing as interactive AI/ML usage becomes popular, where processes invoked interactively may require multiple cores and nodes. Thus, fluctuation of actual resource utilization described in (2) becomes even larger.

### 6.2.2 Observation of interactive jobs




(a) Single-core usage

(b) Multi-core usage

(c) Single- and multi-core usage

(d) Batch-like usage

**Figure 6.1:** Instances of typical interactive job usages on TSUBAME3.0.

In order to confirm the characteristics described above, we analyze the behavior of actual interactive jobs. For this purpose, we collected time series data of CPU utilization during interactive jobs on the TSUBAME3.0 supercomputer from 9th September 2022 to 1st November 2022. On TSUBAME3.0, interactive jobs are explicitly marked and visible to system administrators and users through the job scheduler, allowing us to precisely identify and measure these jobs. In this system, each interactive job is allowed to use up to 14 logical cores on a node,

thus values of CPU utilization are from 0% to 1,400%. The sampling frequency of CPU utilization is five seconds. Excluding data with measurement failure, data of 193 jobs are collected.

Out of the collected data of 193 jobs, we pick up four jobs with different behaviors, and show their time series of CPU utilization in Figure 6.1. In Figures 6.1 (a), (b), and (c), we observe the characteristics (2) in the previous section; the CPU utilization fluctuates largely by repeating idle periods and busy periods. We consider that idle periods correspond to "users' thinking time". On the other hand, the utilization ratio in busy periods are different among jobs. While job (a) uses about 100% CPU (a single core), job (b) uses around 1400% (14 cores). The job (c) includes both single-core usages and multi-core usages. These observations support the characteristics (3); even interactive jobs require multiple cores for higher performance. Figure 6.1 (d) shows a different behavior from others; the CPU utilization is constantly kept high, although this job is invoked on an interactive node. We call such a case a batch-like usage.

In summary, while the actual interactive jobs basically obey characteristics described in the previous section, the utilization in busy periods or lengths of periods is largely different among jobs. Also, some interactive jobs behave like batch jobs.

### 6.2.3 Modeling interactive jobs

In Figure 6.1, we have seen the divergent behaviors of interactive jobs. In order to understand jobs more generally, this section introduces a model, whose purposes are two-fold. One is to observe the statistics of collected data. The other is to process the collected time series data on TSUBAME3.0 based on this model and then embed them in the input of the simulation as described in Section 6.3.

In our model, an interactive job includes repeated busy periods and idle periods during its execution time as illustrated in Figure 6.2. During idle periods, CPU utilization ratio is almost zero and we consider they correspond to users' thinking time. During busy periods, CPU utilization ratio is higher for some computational processes.

Since CPU cores may be slightly used to react to users' commands or for OS jitters, we use a threshold to distinguish the two periods, which is 20% in our experiment. If the CPU utilization during each sampling period of five seconds is higher than this value, the period is regarded as a part of a busy period.

This model can express batch-like usage like in Figure 6.1 (d), which is modeled as a job with a single long busy period. On the other hand, it does not consider the difference in CPU utilization among busy periods in a job. Thus it does not

distinguish multi-core usage and single-core usage in Figure 6.1 (c). In the future, the model could be more sophisticated to capture CPU utilization more in detail. This research, however, uses this simple model for our purposes.

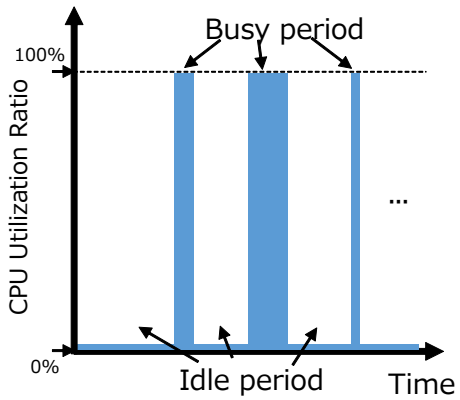



**Figure 6.2:** A simplified behavior of resource utilization during an interactive job. Busy periods and idle periods are repeated.

### 6.2.4 Statistics of measured jobs

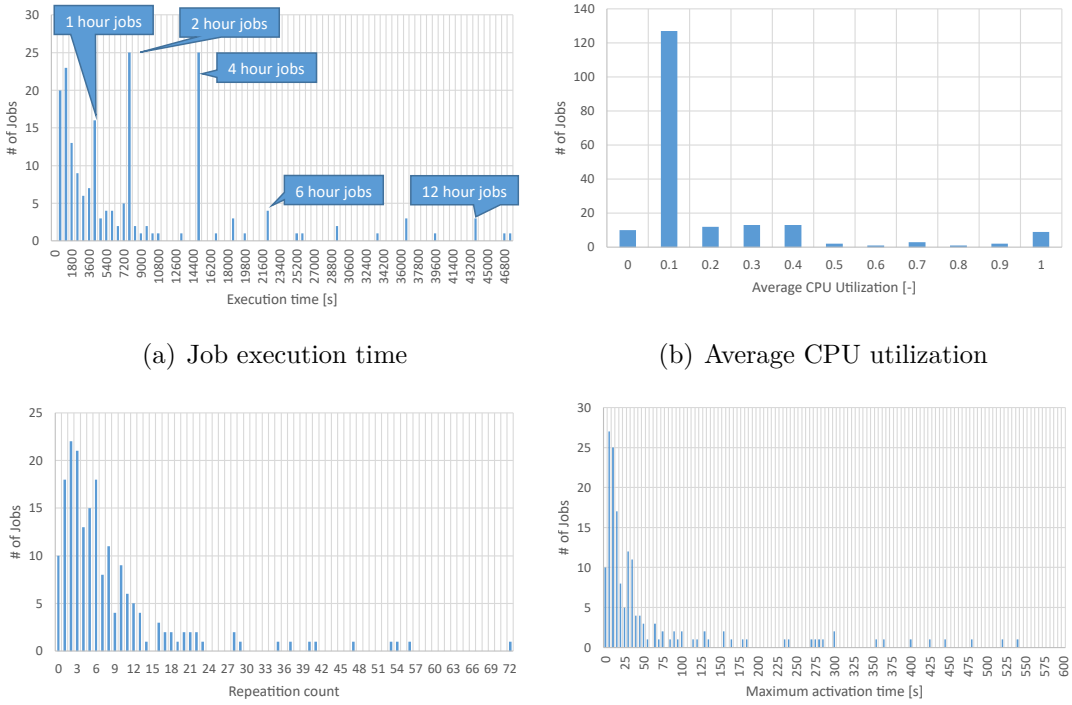
Based on our model, we have processed all collected data of 193 jobs and taken statistics as shown in Figure 6.3. In each histogram, the bar heights represent the number of jobs.

#### Job execution time

In Figure 6.3 (a), we see that short jobs less than 2 hours are popular. Also, there are peaks at 1, 2, 4, 6, and 12 hours and so on. We consider that this is because of users' requested time. While some users close the interactive sessions explicitly, others may leave them without closing, and the system forcibly terminates them. In total, the 90th percentile value is 5.2 hours.

#### Average CPU utilization

Figure 6.3 (b) corresponds to the average CPU utilization that counts both busy and idle periods in each job. In most jobs, it is quite small; for 71% of jobs, it is less than 10% (0.1 core). Also for 37% of jobs, it is less than 0.01 (it is too small and does not appear in the histogram). These observations reinforce our assertion that dedicating computing resources for interactive jobs degrade efficiency of system usage and the aggressive oversubscribing scheduling is important. On the other hand, we see several jobs with higher CPU utilization, which are considered batch-like jobs.




(a) Job execution time

(b) Average CPU utilization

(c) Repetition count ($\leq$ 99th percentile)

(d) Longest busy period ($\leq$ 85th percentile)

**Figure 6.3:** The primary statistics of interactive jobs collected on TSUBAME3.0. The sample size is 193 (jobs).

### Repetition count and longest busy period

Figure 6.3 (c) shows the statistics of repetition counts of busy periods. Jobs with less than 10 busy periods are common, while the maximum count is 94. Figure 6.3 (d) summarizes the longest busy periods. While the 54th percentile is 30 seconds, the longest busy period in all the data is exceptionally long as 46,715 seconds.

From these observations, we can consider typical behaviors of interactive jobs, each of which repeats busy periods about 10 times and each length is less than 30 seconds.

## 6.3 Simulation method with interactive jobs

The chapter aims to demonstrate the effectiveness of oversubscribing scheduling considering both interactive jobs and batch jobs. In order to carry out a simulation of scheduling methods with oversubscribing, we have developed a simulator of job scheduling, named *node conscious oversubscribing scheduler simulator* (NCS)[100], as introduced in previous section.

As the input of simulation, we use the standard work format (SWF), a well-known job trace format[101]. An SWF file consists of records of job information, each of which includes the submitted time point, the requested time, the (actual) execution time, the number of request cores, the request memory amount, and so on.

NCS takes an SWF file as input, and simulates the behavior of each job considering the overhead of oversubscribing. The original NCS, however, assumed the CPU cores are always kept busy during job execution; in other words, it considered only batch jobs, as mentioned in previous section. In order to simulate interactive jobs, there are two issues to be solved.

- The SWF file does not include information on the changes of actual CPU utilization during each job.
- The overhead of oversubscribing in the simulation needs to be revised to reflect the fluctuation of CPU utilization.

### 6.3.1 Data processing to embed CPU utilization

In our simulation, we use a workload trace named UniLu-Gaia-2014-1 [103] (UniLu hereafter) from Parallel Workloads Archive[102]. This data set contains three months data from the Gaia cluster at the University of Luxemburg. We choose it since the cluster consists of two queues, one of which is interactive queue (MQ configuration) and each job data has a flag of job type. As shown in Table 5.1, 1,762 jobs (3.4%) are interactive jobs out of 51,871 jobs.

Unfortunately, the SWF format file has no information on CPU utilization. Instead, we embed information collected on TSUBAME3.0 (Section 6.2.2) into data of interactive jobs in UniLu as follows. For each of UniLu interactive job ($J_U$), we see the execution time of it. And we pick one record from data set generated from TSUBAME interactive jobs ($J_T$), whose execution time is close to that of $J_U$. Then we embed data patterns of $J_T$ that consist of busy and idle periods into $J_U$. To avoid using the same pattern too frequently, we also use randomness in choosing $J_T$. With this method, the generated behaviors include various types of interactive jobs as shown in Figure 6.3. For batch jobs, we assume CPU cores are always fully utilized.

#### Generation of interactive jobs time-series

To conduct the accurate simulation of interactive jobs, we map the time-series results on our supercomputing system to SWF records. However, since the number of measured results is limited we can not make up for all records. This motivates us to generate time-series data which is reasonable as a realistic interactive job.

To generate the time-series, we apply the method that synthesizes the new data based on DTW distance[107], [108]. This method generates the data belonging to each cluster using the clustering results, thus we can obtain the data which has a similar property to the measured data.

### 6.3.2 Performance model of oversubscribing in simulation

NSC applys the two main performance model: (1) We consider slow down of each process depending on the number of coexisting processes on the same CPU core. (2) In a parallel job, the slowest process or threads in the job determines the progress of all the processes or threads in the job, as shown in Section 4.

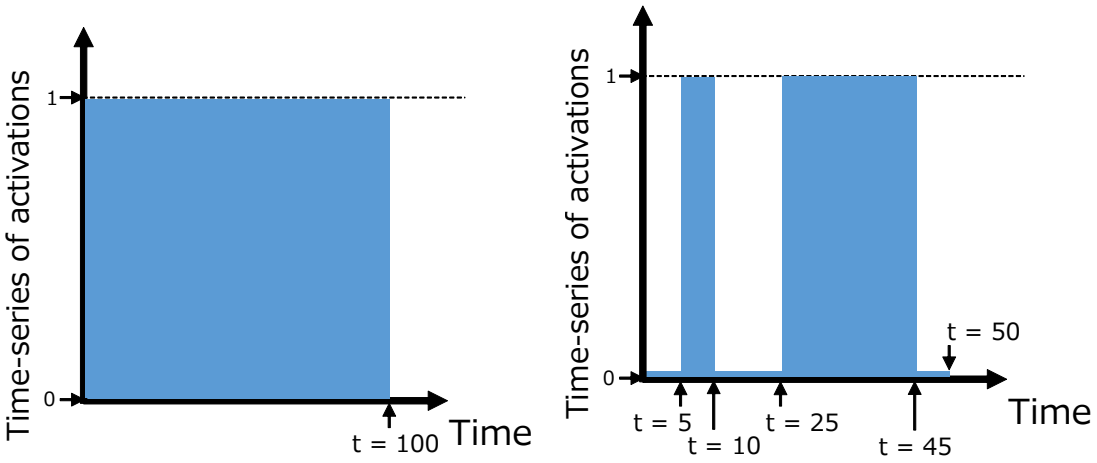
Here we mention the slow down of coexisting processes in (1) further. We use an assumption that the speed of each busy process is divided by $(m' \times C_o)$, where $m'$ is the number of coexisting busy processes and $C_o$ is a constant $= 1.2$. $C_o$ is introduced considering overhead of OSub, including switching cost, cache pollution, synchronization methods, and so on. The value is based on the results in Section 4. Although we have observed the actual behavior is more complex where the value would not be constant, this research adopts simplified model.

In addition to overhead of oversubscribing, in order to support interactive jobs, we additionally assume that idle time is constant if oversubscribing cause. This is why idle time corresponds thinking time or light-weight task like keyboard inputting. Thus we implemented the behavior of NCS simulation as follows. When a process is idle, it does not affect the progress of other coexisting processes. Also its progress is not affected by other coexisting processes. The latter rule is set since the lengths of idle periods are mainly determined by interactive users' behaviors, not by other jobs.

### 6.3.3 Treatment of interactive jobs under oversubscribing scheduling

This section shows the simulating flow when considering oversubscribing. For example, take two jobs. Figures 6.4 (a) and (b) show the time-series of two jobs. The one indicates constant behavior (batch job, Job A), and the other indicates intermittent behavior (interactive job, Job B). The batch job and interactive job

would end at $t = 100, 50[s]$ respectively if they are dedicated execution. The interactive job has two times activation at $t = 5, 25[s]$. The first activation time is 5 seconds and the second is 20 seconds. The sum is 25 seconds. The idle time between activation is 15 seconds. The epilogue time, which is the rest time after last busy period, is 5 seconds. The job's busy periods ratio is 25/50 * 100 = 50% in total. This is an exaggeration case and it must be smaller for most interactive jobs.




(a) Batch job (Job A) (b) Interactive job (Job B)

**Figure 6.4:** Time-series of behavior for batch and interactive jobs.

Figure 6.5 shows the CPU utilization (CPU speed for each job) when Jobs A and B are oversubscribed. Let us explain the progress of the jobs. Note the overhead ratio = 1.20.

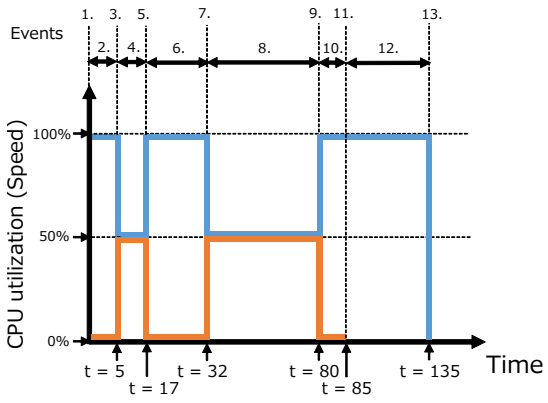



**Figure 6.5:** Time-series of the CPU utilization (CPU speed for each job) when the jobs are oversubscribed. Event numbers are on top.

1. $t = 0$: Both jobs are allocated same resource and start. Because job B is inactive, the speed of job A is still 100%.

2. $0 < t < 5$: Job B is at idle period. During this period, job A is at 100% speed. Note that we assume that idle time is constant regardless of the existence of coexisting jobs.

3. $t = 5$: Job B is activated. The utilization for job A and B is reduced to 50% (= 1/2).

4. $5 < t < 17$ : Job B's first activation. Due to the speed down, the activation time is extended from 5 seconds to 12 (= 5 / 0.5 * 1.2) seconds.

5. $t = 17$: Job B is deactivated. The speed for job A recovers to 100%.

6. $17 < t < 32$: Job B is inactive. Like first idle period, the idle time does not change regardless of the existence of coexisting jobs. Job A is at 100% speed.

7. $t = 32$: After 15 seconds of idle time, Job B activates again. The utilization for jobs A and B is reduced to 50%.

8. $32 < t < 80$: Job B's second activation. Due to the speed down, activation time is extended from 20 seconds to 48 (= 20 / 0.5 * 1.2) seconds.

9. $t = 80$: Job B is deactivated. The speed for job A recovers to 100%.

10. $80 < t < 85$: Job B is inactive. Like other idle periods, the idle time is constant regardless of the existence of coexisting jobs. Job A is at 100% speed.

11. $t = 85$: After 5 seconds idle time, Job B ends. Job A is calculating.

12. $85 < t < 135$: Job A only runs at 100% speed.

13. $t = 135$: Job A ends because the computation is completed.

The slowdown, turnaround time divided by original running time, for Job A is 135/100 = 1.35 times owing to the existence of job B. It looks a bit large but it came from the high CPU utilization of job B. As explained in Section 6.2.4, most interactive jobs actually have quite low CPU utilization. If the CPU utilization is 1%, the value is 1.01. It is quite tolerable. On the other hand, the slowdown for Job B is 85/50 = 1.7. It sounds large but we think the benefit is even greater because the responsiveness is improved. Without oversubscribing, Job B must wait until Job A ends.

### 6.3.4 Extension of SWF for interactive jobs simulation

SWF is the workload trace format in the supercomputing fields[101]. We extend this format with additional information to conduct our interactive jobs' model. The model requires three elements: prologue time (first idle period length), each activation and idle time, and epilogue time (last idle period length). Table 6.1 shows the part of the extended SWF format. For example, the job B in Section 6.3.3 can be expressed with this format as "5 5 2 5 20 15".

**Table 6.1:** Extension of SWF for interactive jobs simulation. The original SWF column length is 18 and the extension can start from column 19, but we make room for some columns.

| Start column number | # of elements | Name |
|---|---|---|
| 22. | 1 | Prologue time |
| 23. | 1 | Epilogue time |
| 24. | 1 | Repetition count $N$ |
| 25. | $N$ | Each activation time |
| $(25+N)$. | $N-1$ | Each idle time |

## 6.4 Simulated results of oversubscribing scheduling considering interactive jobs

This section shows results of the simulation that evaluate the effectiveness of the oversubscribing scheduling. The simulation used processed job data based on UniLu workload trace as in Section 6.3.1 The size of the simulated system is the same as UniLu Gaia in Table 5.1.

We compare the several scheduling methods and configurations described in Section 5.3.1. First, in the SQ configuration, a single queue accepts both job types. Here a system parameter, the maximum multiplicity $M$, is configured. SQ with $M = 1$ is the simplest system, which does not allow oversubscribing, and is considered to impose long waiting times on jobs. SQ with $M > 1$ uses our proposed oversubscribing scheduling. As the comparison targets, we evaluate the MQ configuration that prepares a queue for batch jobs and one for interactive jobs. The ratio of interactive nodes is configured by the ratio $R$. Also, we allow oversubscribing on the interactive queue, with the maximum multiplicity of $M_I$.

### 6.4.1 Experiment 1: Effectiveness of aggressive oversubscribing

This experiment evaluates performance of job scheduling policies. To see effects of oversubscribing, we compare SQ configurations with $M = 1, 2, 3, 4$. $M = 1$ does not use oversubscribing. Also, we evaluate MQ configurations with $R = 1\%, 2\%, 3\%$, that correspond to different numbers of interactive nodes. On the interactive nodes in MQ, $M_I$ is fixed at 4.

#### The responsiveness of interactive jobs

The purpose of the oversubscribing scheduling is to satisfy more interactive users' demand immediately, while restricting the number of coexisting processes. Table 6.2 shows the ratio of interactive jobs that suffer from waiting time. SQ configuration with $M = 1$ (no oversubscribing) imposes waiting time on 6.2% of jobs, while the rest (93.8% of jobs) are started immediately. With larger $M$ using OSub, the ratio is decreasing as expected, and when $M = 4$, all interactive jobs can be started immediately. We see MQ configuration also eliminates waiting times with $R \geq 2\%$. On the other hand, when $R = 1\%$, 8.3% experiences waiting time, which is larger than with "SQ, $M = 1$". In MQ, misconfiguration of the system ($R = 1$), as mentioned in Section 5.4.6, can have a large impact. Figure 6.6 (a) shows the longest waiting time among interactive jobs. We see as ratio in the Table 6.2 increases, the longest waiting time becomes worse.

**Table 6.2:** Ratio of interactive jobs that suffer from waiting time.

| Case | Ratio | Case | Raio |
|---|---|---|---|
| SQ, M=4 | 0% | MQ, R=3% | 0% |
| SQ, M=3 | 0.7% | MQ, R=2% | 0% |
| SQ, M=2 | 1.2% | MQ, R=1% | 8.3% |
| SQ, M=1 | 6.2% | | |

#### Slowdown of jobs

With oversubscribing, the executions of both interactive jobs and batch jobs get slower. Also the waiting time extends the turnaround time visible to the users. To investigate these effects, we evaluate the "slowdown" metric, as defined in Section 5.3.2.

Figure 6.6 (b) shows the largest slowdown among batch jobs and interactive jobs. With MQ, the largest slowdown of batch jobs is significantly larger than that of interactive jobs. Also, we see $MQ, R = 3\%$ rises 18,652x slowdown while it is

12,697x with $R = 2\%$, while the impact on interactive jobs is minor. Again we observe the sensitivity of configuration in MQ. In SQ, larger $M$ contributes to a smaller slowdown both for batch jobs and interactive jobs. This property is expected to be helpful even for batch job users since it offers a shorter turnaround time.

In order to observe the tendency among all jobs, Figure 6.6 (c) shows the distribution of slowdown values among jobs. The X-axis is the cumulative ratio of jobs and the Y-axis is the slowdown value. Here interactive and batch jobs are mixed. We see "SQ, M=1" and MQ configurations show similar curves; while the slowdown of around 92% of jobs is one, the curve rises sharply. It means minor parts of jobs suffer from a very large slowdown. With SQ ($M \geq 2$), around 30% of jobs suffer from a slowdown, however, the slowdown ratio is smaller than 3 for most of such jobs. When $M = 4$, the largest slowdown is 4.8 among both job types. From this observation, we can say that SQ with OSub scheduling can achieve fairer scheduling.

#### Efficiency of the system

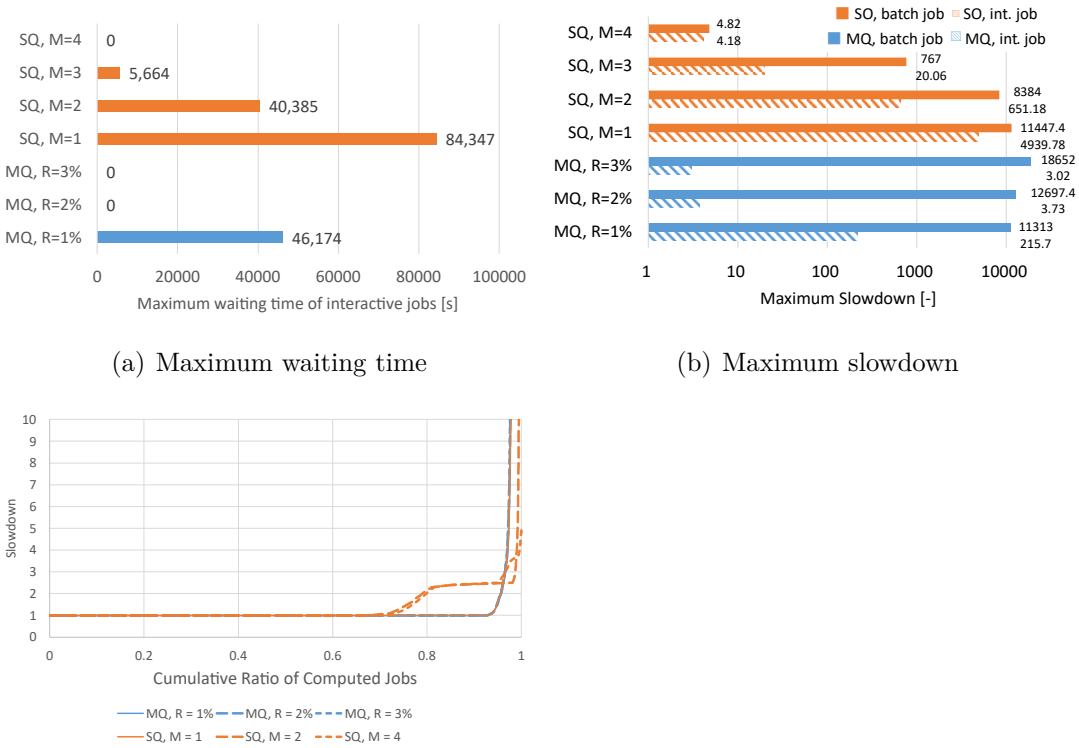
Finally, we show the overall system efficiency as defined in Section 5.3.2. Table 6.3 shows makespan, the time until all the jobs are completed. SQ with $M \geq 3$ exhibits around 2.5% longer makespan, however, we observed this is due to a few jobs scheduled lastly in the duration. When we observe interim progresses, the time until 90% or 95% of jobs are completed, the results are similar among all the configurations.

**Table 6.3:** System efficiencies in experiment 1. Makespan is the ending time of the last completed job. It also shows the times when 90% and 95% of jobs are completed.

| Case | Makespan [hour] | Finished time [hour] | |
|---|---|---|---|
| | | 90th | 95th |
| SQ, M=4 | 2194 | 1859 | 2004 |
| SQ, M=3 | 2192 | 1870 | 2016 |
| SQ, M=2 | 2154 | 1883 | 2013 |
| SQ, M=1 | 2138 | 1852 | 2015 |
| MQ, R = 3% | 2138 | 1851 | 2015 |
| MQ, R = 2% | 2138 | 1851 | 2015 |
| MQ, R = 1% | 2138 | 1850 | 2015 |

### 6.4.2 Experiment 2: Robustness to the interactive job dominant system

While the previous section used the fixed job data set, we consider that OSub scheduling will have more advantages when interactive jobs are more dominant in




(a) Maximum waiting time

(b) Maximum slowdown

(c) Distribution of Slowdown

**Figure 6.6:** The results of the experiment 1.

HPC workloads. In order to confirm this expectation, we conduct the evaluation with artificial data sets containing increasing proportions of interactive jobs.

To generate these data sets, we modified the original UniLu trace by converting selected batch jobs into interactive jobs. The selection targeted batch jobs that exhibited characteristics typical of interactive workloads - namely, those with smaller resource requirements (12 cores or fewer) and shorter execution times (less than 10,000 seconds). This approach is based on the observation that interactive jobs tend to be shorter in duration and require fewer computational resources compared to traditional batch workloads. By randomly selecting jobs meeting these criteria and progressively increasing the selection frequency, we generated multiple data sets with varying proportions of interactive jobs.

This approach resulted in six different data sets, as shown in Table 6.4. Data set 6, with the highest proportion, includes 39,473 (76%) interactive jobs.

With SQ configuration, we fixed the maximum multiplicity $M = 4$, allowing oversubscribing. On the other hand, through the preliminary measurement with MQ, we observed it cannot follow the change of interactive jobs ratio at all. To give an advantage to MQ, we conducted an intensive parameter survey to determine the

optimal system parameters, like discussed in Section 5.4.6, for each data set. The parameters include $R$ and $M_I$. Also we allow oversubscribing even on the batch queue, with the maximum multiplicity of $M_B$. The found optimal parameters are shown in Table 6.4. For data set 6, the system needs to prepare 18% of dedicated nodes in the system for the interactive jobs for better performance.

**Table 6.4:** The generated data set and optimal parameters in MQ configuration

| | 1. | 2. | 3. | 4. | 5. | 6. |
|---|---|---|---|---|---|---|
| # of interactive jobs | 1762 | 6500 | 15913 | 25336 | 34772 | 39473 |
| ratio | 0.034 | 0.125 | 0.307 | 0.488 | 0.670 | 0.761 |
| Well-configured values for MQ | | | | | | |
| $R$[%] | 2 | 2 | 6 | 12 | 18 | 18 |
| $M_{\mathrm{I}}$ | 4 | 4 | 4 | 4 | 4 | 4 |
| $M_{\mathrm{B}}$ | 4 | 4 | 5 | 8 | 8 | 8 |

Figure 6.7 shows the maximum slowdown for each case. Apparently, SQ ($M = 4$) achieves stable performance and the slowdown values are less than 5 for all the jobs. With MQ configuration, we observe interactive jobs have a modest slowdown successfully, owing to optimized $R$ for each data set. On the other hand, batch jobs suffer from a larger slowdown. These results are obtained even with the parameter survey. If the ratio of interactive jobs changed dynamically (such as the change between daytime and night), the performance with MQ would be much worse.

### Discussion on scaling with increased interactive workloads

Our evaluation showed OSub's effectiveness with interactive job ratios up to 76%. When the number of interactive jobs increases further, increasing the multiplicity parameter $M$ would be desirable to maintain the benefits of OSub - reducing waiting times and improving job throughput. However, memory capacity, not CPU sharing, becomes the key factor that limits such scaling of $M$.

There are two main factors that affect scalability. First, as shown in Section 6.2.4, interactive jobs use CPU resources intermittently. Most interactive jobs have average CPU utilization below 10%, which means CPU sharing remains effective even with many concurrent jobs. Second, while jobs share CPU time effectively due to their intermittent nature, each job requires its full memory allocation throughout its lifetime, regardless of whether it is active or idle.

This means that as the number of concurrent interactive jobs grows, memory becomes the primary constraint. While CPU cores can be shared efficiently through time-division multiplexing, memory requirements add up directly with each additional job. Therefore, the maximum number of concurrent jobs that can be supported depends mainly on the system's memory capacity.

For the multiplicity parameter $M$, higher values reduce waiting times and improve job throughput, but only until memory limits are reached. Our results showed good performance with $M = 4$ even at 76% interactive jobs, but the maximum practical value for $M$ will typically be determined by available memory rather than CPU constraints.

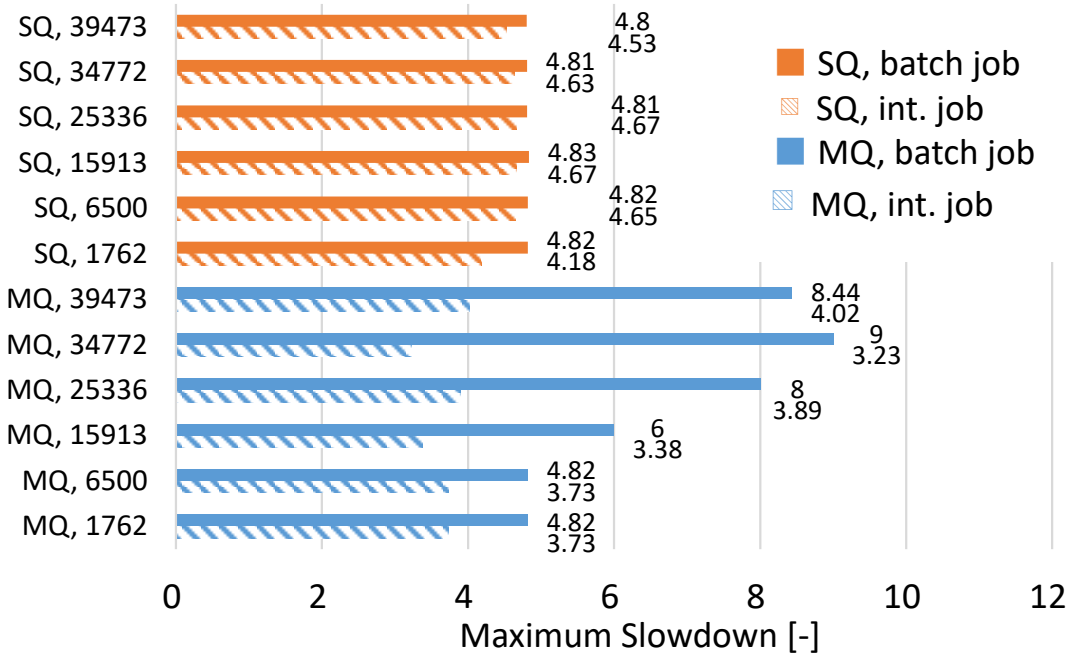



**Figure 6.7:** The results of experiment 2; maximum slowdown with different trace data sets.

# 7

# Physical Demonstration of Oversubscribing Scheduling

## Contents

## 7.1 Introduction to physical system experiment

As discussed in Chapter 1, oversubscribing (OSub) adoption in HPC environments has been limited despite its potential benefits, primarily due to concerns about performance predictability and fairness. HPC centers, which must maintain consistent performance for large-scale scientific computations, have been particularly hesitant to adopt resource sharing approaches that could introduce performance variability.

Our research has addressed these concerns through systematic evaluation, as outlined in Chapter 3. The previous chapters have presented comprehensive simulation-based validation using real workload traces, demonstrating OSub's effectiveness in reducing waiting times while maintaining acceptable performance trade-offs. However, simulation-based validation faces inherent limitations. While our simulation incorporates performance degradation models based on detailed analysis in Chapter 4, it assumes constant overhead regardless of job combinations, which contradicts our findings about varying interference patterns between different types of applications. Furthermore, simulations operate under idealized conditions that neglect system-level overheads such as scheduling delays and job switching costs.

Physical system validation becomes crucial for two fundamental reasons. First, it provides empirical validation of the benefits observed in simulation studies - particularly the reduction in waiting times and acceptable performance trade-offs under actual system conditions. Second, and perhaps more importantly, it demonstrates the practical implementability of OSub within existing HPC infrastructure. This validation addresses a fundamental question of whether OSub can be effectively realized in real system environments while maintaining the predicted performance characteristics and operational benefits.

However, conducting meaningful physical system experiments presents several technical challenges. Production workload traces typically span months or years and involve thousands of compute nodes, making direct replay impractical in experimental environments. Additionally, these traces contain only timing information without specifying the actual computational content of jobs. This chapter presents our approach to physical system validation, demonstrating OSub effectiveness through controlled experiments that maintain connection to real HPC workload patterns while addressing these practical constraints of experimental environments.

## 7.2 Previous physical system experiments: methodology and workload reproduction

In general, the main objective of experimenting on real physical systems is to evaluate features considered difficult to assess in simulators. For example, some research has reported on the latency of production job schedulers [13], [109], [110]. Other studies have conducted experiments on real physical systems to validate simulator results [111]. In such research, the target is to evaluate the scheduling algorithm or efficiency of the job scheduler. It is often sufficient to mold the job in terms of runtime and parallelism without actual calculation and submit it to the job scheduler, without the need to run the application. Consequently, it is common to use `sleep` for the runtime inside the job, referred to as a sleep job.

Conversely, oversubscribing jobs affect each other's performance, making it insufficient to use sleep jobs. Running actual applications inside jobs becomes necessary. Additionally, not using sleep jobs introduces another challenge: the reproducibility of each job's runtime. While it is easy for sleep jobs to reproduce runtimes written in the submission history, it is challenging for real applications. Our research uses runtime-reproducing real applications to accurately evaluate the effects of oversubscribing.

Furthermore, job submissions causing system load are typically done by replaying workload traces. These traces are considered system-dependent, and when applied to other systems, modification is needed while maintaining the original trace's features. One of the most significant features is the arrival pattern of job submissions. Lublin et al. reported that dividing the day into 48 30-minute slots and maintaining the submission patterns for each slot is crucial [112]. Therefore, we also created and utilized traces that preserve these essential characteristics for our experiments.

## 7.3 Overview of the physical system experiment

### 7.3.1 Scenario of the system experiment

Our experiments are designed based on observations of activities in current supercomputing systems. Generally, batch jobs run continuously throughout the day, regardless of the time. In contrast, interactive jobs are concentrated during daytime hours. This pattern likely corresponds to typical human activity periods.

Our objective is to observe system behavior during periods of human activity. Figure 7.1 illustrates our experimental period (or timeline). It consists of three phases, described as follows:

- Warm-up: Period outside of human activity (before the first interactive job submission)
- Active phase: Period during which interactive jobs are running (human activity time)
- Cool-down: Period after the last interactive job ends

We focus on the responsiveness of interactive jobs during the human activity period. Responsiveness is evaluated through waiting time, which is the duration between job submission and job start. We assess performance degradation, particularly for batch jobs. This assessment uses metrics such as TAT or slowdown, which reflect both waiting time and running time. Furthermore, throughout all periods, we evaluate potential side effects on the efficiency of system processing.

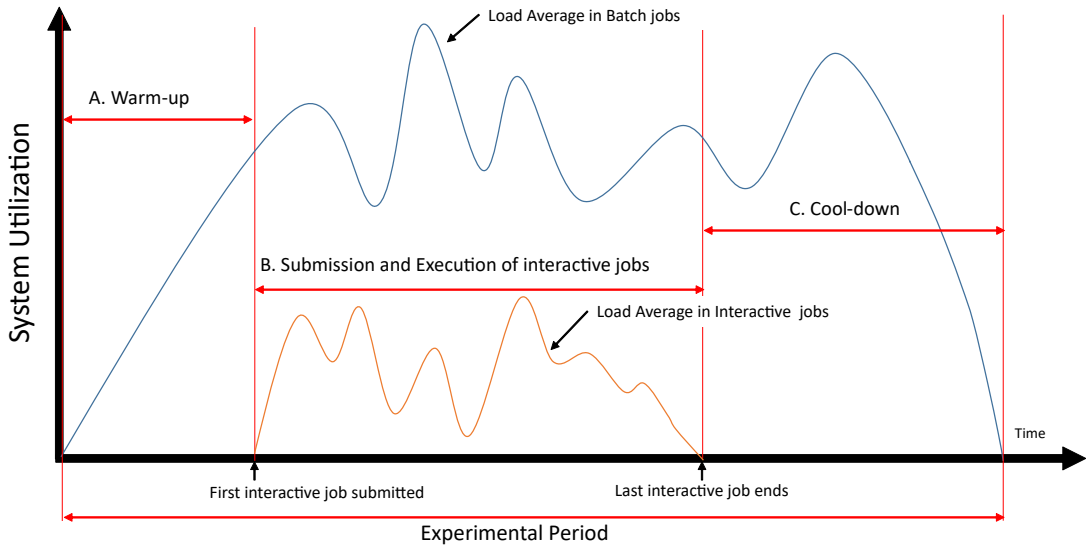



**Figure 7.1:** Experimental period and system utilization patterns, showing the relationship between batch and interactive workloads.

### 7.3.2 How to guarantee reproducibility of the system experiment

Maintaining the reproducibility of experimental input is crucial, as our experiments are conducted with various system configurations and scheduling policies. As mentioned in Section 7.2, many experiments on HPC systems can be conducted by replaying workload trace files, ensuring input reproducibility. However, in our scenario where actual applications run inside jobs, reproducibility becomes

more complex. First, a job must execute a pre-determined application in every experiment. Second, the application should consistently reproduce the runtime specified in the workload trace under dedicated conditions. Therefore, we need to maintain reproducibility in two aspects: (1) job submission patterns and (2) job applications and their runtimes.

To address point (1), we replay the workload trace using the standard workload format (SWF) [101]. Each record represents a job, containing information such as arrival time, running time, and required processor cores.

To address point (2), we develop specialized applications for both job types and run them inside the jobs. These applications can execute for a given runtime. We associate these applications with each job, corresponding to each record in the SWF.

### 7.3.3 Emulation of interactive jobs

Interactive jobs can be characterized by alternating busy and idle periods, as illustrated in Figure 6.2. In our experiment, these jobs are submitted as batch jobs. Inside each job, a specialized application operates according to pre-determined busy and idle periods. It is important to note that our experiment does not involve actual human interaction with the interactive jobs.

The time series data of busy/idle periods allocated to each job are derived from user history in our supercomputing system. We provide a detailed description of the application in Section 7.4.4.

## 7.4 Experimental setup of the physical system experiment

### 7.4.1 Summary of experimental environment

Figure 7.2 provides an overview of our experimental environment from a software perspective. We conduct experiments by replaying recorded jobs. Using *the job submitter program*, we submit jobs to the *Slurm* job scheduler, following job submission patterns recorded in the *workload trace file* in standard workload format (SWF). Inside interactive jobs, *the emulating interactive behavior application* (EIBA) executes, while *the runtime reproducing batch application* (RRBA) runs inside batch jobs. To maintain performance in oversubscribing systems, we utilize *Broadcast Synchronization* (BS), a runtime gang scheduler for oversubscribing jobs, shown as "gang scheduler" and "client agent" in the figure.

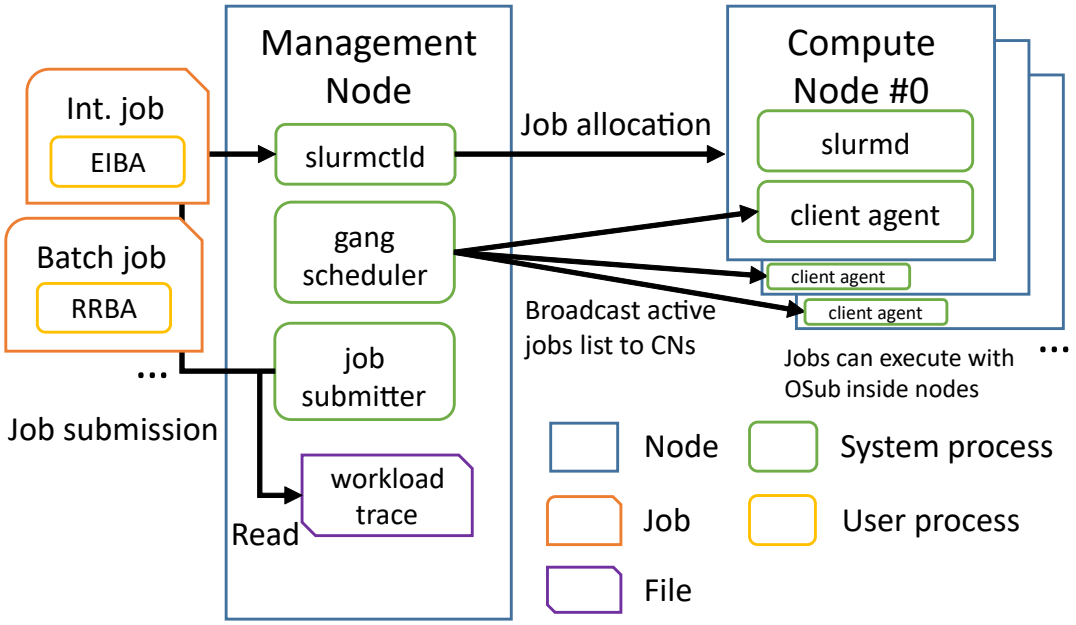



**Figure 7.2:** Software architecture diagram of the experimental system, showing components and their interactions.

We conduct multiple trials with various system configurations, as discribed in Sections 5.3.1 and 6.4. First, we apply the Multiple Queues solution (MQ). This approach prepares dedicated interactive nodes, with a portion of nodes assigned to process only interactive jobs and the others processing only batch jobs. In interactive nodes, oversubscribing is enabled. Although multiple jobs' busy periods coinciding could cause degradation of running time, this scenario is considered unlikely to occur. We assign a maximum multiplicity in interactive nodes $M_i$ of four, which represents a relatively conservative solution in terms of running time compared to the Single Queue solution. We evaluate the MQ by changing the number of interactive nodes $N_i$ and batch nodes $N_b$, while keeping the total number of nodes constant. Second, we adopt the Single Queue solution (SQ). In this solution, both types of jobs can execute on all nodes, with oversubscribing enabled on all nodes. This approach would likely reduce the waiting time for both type of jobs, while most jobs might experience degradation in running time. Thus, SQ can be considered a more aggressive approach than MQ. We evaluate the SQ by changing the maximum multiplicity $M$. Scheduling occurs at the CPU core level, with distribution spread across nodes. The system selects nodes with higher vacancy rather than packing jobs onto fewer nodes. This approach prevents any single node from experiencing excessive oversubscribing. Additionally, for the SQ solution, we conduct evaluations both with and without Broadcast Synchronization (BS), the latter referred to as the Naive approach.

### 7.4.2 Hardware configuration

While our introduction highlighted the importance of AI workloads that typically utilize GPUs, we conducted this initial study of oversubscribing on a CPU-only cluster. This choice allows us to establish fundamental principles of resource sharing and scheduling that can be later extended to GPU-enabled systems. Additionally, CPU resources remain essential for many interactive computing tasks, including code development and data preparation phases of AI workflows.

We constructed an experimental HPC cluster using three symmetrical physical nodes, each equipped with dual AMD EPYC 7513 processors. These physical nodes were divided to create a virtual cluster consisting of 12 virtual nodes. To match the experimental workload trace requirements, each virtual node was configured with 12 CPU cores. The AMD EPYC 7513 processor has a specific locality design. Its 32 physical cores are organized into four Core Complex Dies (CCDs), with each CCD connected to two of the processor's eight memory channels. For each virtual 12-core node, we allocated two CCDs while maintaining dedicated memory channel assignments. In this configuration, we used all eight cores from one CCD and four cores from another CCD, leaving the remaining four cores of the second CCD unused. This means that jobs running on the same node inevitably share memory channels, which can affect performance even when using dedicated CPU cores. Table 7.1 presents the specifications of our cluster configuration. While our experimental system's scale of 12 nodes is modest compared to production supercomputers, this size is sufficient for our research objectives. The key interactions between oversubscribed jobs - including parallel jobs spanning up to 5 nodes - can be effectively studied at this scale. The system size allows us to observe realistic resource contention and scheduling behaviors, particularly as we have carefully scaled our workload to maintain realistic job interaction patterns, as shown in Section 7.4.3.

**Table 7.1:** Hardware specifications of the experimental cluster. Each virtual node consists of CPU cores from two CCDs to maintain dedicated memory channel assignments.

| | |
|---|---|
| Number of nodes | 12 |
| CPU | AMD EPYC 7513 |
| CPU cores per node | 12 (8 cores from CCD1 + 4 cores from CCD2) |
| Memory per node | 64 GiB, DDR4-3200 |
| Memory channels per node | 4 (2 channels per CCD) |

### 7.4.3 Experimental workload trace

We use a real workload trace from the Gaia cluster at the University of Luxembourg (UniLu) [103] for our experiment to ensure realistic conditions. The trace records job types, including batch and interactive jobs. Using this trace, we construct the workload according to the submission patterns mentioned in Section 7.3.1. However, several modifications and assumptions are necessary due to environmental differences and insufficient information in the SWF, as follows:

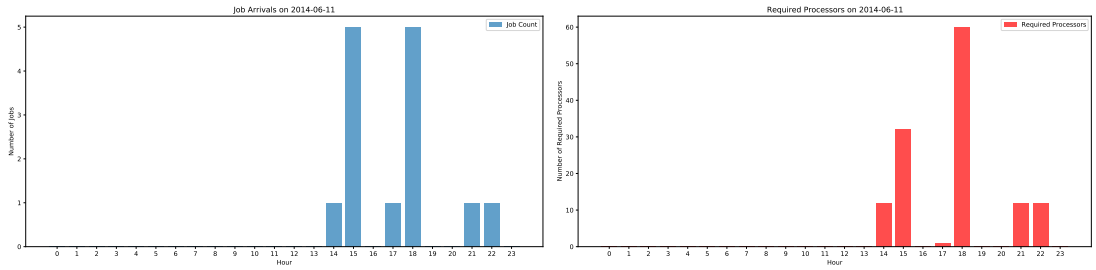
1. The workload trace covers 3 months. We need to extract the duration that aligns with our scenario.

2. Our experimental real physical system is 8% the size of Gaia. This means the original workload is too large for our system to process. We need to reduce the workload size accordingly.

3. The SWF does not record the content (application) of the jobs. We make assumptions about appropriate applications for both batch and interactive jobs.

To address issue (1), we first decide to analyze the workload on a daily basis, since daily patterns show how users typically use the system. To select which day to analyze, we observe the usage patterns of interactive jobs on weekdays. From this observation, we identify and select a busy day that shows typical interactive job activity.

Figure 7.3 illustrates the submission pattern of interactive jobs on the selected day. We interpret the peak hours as 2 PM to 10 PM. We consider this period as human active time, as explained in Section 7.3.1.

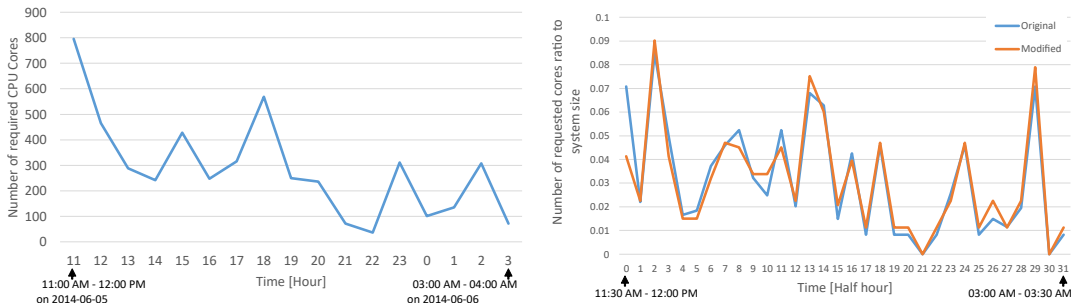



(a) Job count (b) Number of required cores

**Figure 7.3:** Interactive job submission pattern extracted from the UniLu trace. Interactive user activity increases from 2 PM. Although not shown in the figure, there are no interactive jobs between the end of this period and 8 AM the following day.

For batch jobs, we construct the submission pattern to cover the peak hours of interactive job activity. We analyze the workload trace on a daily basis and select a typical submission pattern as our foundation, as illustrated in Figure 7.4 (a). The selected period extends from 11:00 AM to 3:30 AM the following day. To address issue (2), we reduce the workload by randomly selecting jobs and decreasing their size. Listing 2 illustrates these procedures using pseudo-code.

Additionally, we adjust the ratio of serial jobs and the ratio of jobs with parallelism in powers of two to match those of the original trace for batch jobs, following the approach described in [112]. Figure 7.4 (b) illustrates the modified submission pattern. We can confirm that it maintains the original time series characteristics.




(a) Number of required cores of batch jobs in original trace

(b) Number of required cores of batch jobs in our experimental trace

**Figure 7.4:** Batch job submission patterns: comparison between original UniLu trace and modified trace for the experimental system.

Finally, to address issue (3), we assign applications to the jobs. For batch jobs, we randomly allocate the benchmarks `LU, BT, MG, and SP`. We assign EIBA to all interactive jobs; however, the busy/idle periods emulated within each job vary. We use the busy/idle period time series measured in our supercomputing system and randomly allocate these periods to the interactive jobs, as mentioned in Section 6.2.

Table 7.2 presents information about the modified traces.

**Table 7.2:** Summary of modified workload trace characteristics, showing job counts, parallelism, runtime, and busy period metrics for both interactive and batch jobs.

| | Job Count | Maximum parallelism | Maximum runtime[s] | Average runtime[s] | Maximum total busy periods time[s] | Average total busy periods time[s] |
|---|---|---|---|---|---|---|
| Interactive job | 14 | 12 | 43209 | 8566 | 7299 | 1123 |
| Batch job | 68 | 25 | 14410 | 5968 | 14410 | 5968 |

```
Begin
    For each slot in SLOTS_PER_DAY:    // SLOTS_PER_DAY = 48
        // Calculate target core count for this slot
        original_core_count = original_trace[slot]
        normalized_core_count = original_core_count /
           original_system_size
        target_core_count = normalized_core_count *
           target_system_size

        // Define target range, TARGET_RANGE_PERCENTAGE = 0.1
        target_range_min = target_core_count * (1 -
           TARGET_RANGE_PERCENTAGE)
        target_range_max = target_core_count * (1 +
           TARGET_RANGE_PERCENTAGE)

        adjusted_slot_jobs = empty list
        original_slot_jobs = GetJobsForSlot(original_trace, slot)

        Repeat:
            // Step 1: Randomly select jobs from original trace
            adjusted_slot_jobs = RandomlySelectJobs(
               original_slot_jobs)

            // Step 2: Add jobs if below target range
            While SumCores(adjusted_slot_jobs) < target_range_min:
                Add RandomlySelectJob(original_slot_jobs) to
                   adjusted_slot_jobs

            // Step 3: Remove jobs if above target range
            While SumCores(adjusted_slot_jobs) > target_range_max:
                Remove RandomlySelectJob(adjusted_slot_jobs) from
                   adjusted_slot_jobs

            // Step 4: Check if within target range
            If target_range_min <= SumCores(adjusted_slot_jobs) <=
                target_range_max:
                Break repeat

            // Step 5: If not within range, halve job sizes and
               retry
            For each job in original_slot_jobs:
                job.core_count = job.core_count / 2
        End Repeat
        Add adjusted_slot_jobs to adjusted_trace
    End For
    Return adjusted_trace
End
```

**Listing 2:** Pseudocode for workload trace generation, showing the procedure for scaling down the original trace while maintaining its temporal characteristics and resource usage patterns.

### 7.4.4 Software configuration

#### Job scheduler Slurm

The well-known job scheduler Slurm supports oversubscribing, allowing multiple jobs in permitted partitions to use the same resources. A configuration parameter sets the maximum multiplicity for nodes enabled for oversubscribing. We can use this parameter to limit the upper bound of oversubscribing multiplicity, as mentioned in Section 7.4.1.

Slurm provides not only oversubscribing but also a preemption mechanism to mitigate performance degradation caused by oversubscribing. This feature can repeatedly switch overlapped jobs between suspended and resumed states. However, we do not use these functions due to their significant overhead. We revisit this topic in Section 7.4.4.

As described in Section 7.4.1, scheduling occurs at the CPU core level with a distributed allocation. This approach is implemented using the configuration parameter `SelectTypeParameters=CR_Core,CR_LLN`.

#### Job submitter

The job submitter interprets the SWF file, molds jobs from the records, and submits them to the job scheduler at the appropriate times. In addition to the SWF file, the job submitter requires the replay start time (in UNIX time format) and supplementary application information as inputs. This supplementary information includes the arguments for EIBA and the outermost loop length of RRBA for each job.

The job submitter is implemented using Slurm's C API, which provides direct access to the job scheduling functions through lightweight system calls. This implementation choice ensures minimal overhead during job submission, allowing the submitter to maintain precise timing control.

#### Emulating Interactive Behavior Application (EIBA)

Emulating interactive behavior application replays pre-determined busy and idle periods. Table 7.3 shows the data format of busy/idle periods that the application receives.

We assume that oversubscribing does not affect the duration of idle periods. To support this assumption, different timing methods are used for busy and idle periods. When measuring the duration of a busy period, the application monitors

**Table 7.3:** Input argument format for EIBA (Emulating Interactive Behavior Application), specifying the sequence and duration of busy/idle periods.

| Argument Position | Description | Type | Notes |
|---|---|---|---|
| 1 | Number of periods (n) | Integer | Required, n $\geq$ 1 |
| 2 | Duration of first idle period | Integer | Required |
| 3 | Duration of first busy period | Integer | |
| 4 | Duration of second idle period | Integer | |
| 5 | Duration of second busy period | Integer | |
| ⋮ | ⋮ | ⋮ | |
| n+1 | Duration of last period (idle or busy) | Integer | |

consumed CPU time, whereas for idle periods, it monitors wall time. If the job is oversubscribed during a busy period, the duration of that busy period is extended.

Additionally, the application supports multi-threaded execution with synchronization. When replaying a busy period, it continues until all threads consume the specified CPU time. Consequently, even if one thread executes with oversubscribing and experiences reduced performance, it determines the duration of the busy period. Listing 3 shows the implementation of the multi-threaded part. Lines 7 and 8 express the repetition of busy/idle periods.

`timespecs` is an array representing the durations of busy/idle periods, and `timespecs.size()` corresponds to $n$ in Table 7.3. Inside the `for` statement, the called function depends on whether the `index` is even or odd. If `i` is even, it calls the function that sleeps for `timespecs[i]`, while if it is odd, it calls the function that executes for `timespecs[i]`. The synchronization is expressed in the iteration part of the `for` loop. A thread waits here until all threads consume the `timespecs[i]` of CPU time.

```
1  void CPUBusyIdleCycle::startup_thread(std::size_t
       thread_index)
2  {
3    threads_.emplace_back([this](std::size_t thread_index) {
4      const std::chrono::steady_clock::time_point
         start_wall_time{std::chrono::steady_clock::now()};
5      const std::timespec start_cpu_time{thread_cpu_time_now()
         };
6      barrier_->arrive_and_wait();
7      for (std::size_t i{0}; index < timespecs.size(); barrier_
         ->arrive_and_wait(), ++i)
8        (this->*thread_do_actions_[i & 0b1])(timespecs[i]);
9      ...
10   }, thread_index);
```

```
}
```

**Listing 3:** Core implementation of the Emulating Interactive Behavior Application (EIBA), showing the multi-threaded execution logic for alternating busy and idle periods with thread synchronization.

### Runtime Reproducing Batch Application (RRBA)

We assign the NAS Parallel Benchmark Suite (NPB) as the batch job applications. For replay, we need to reproduce the running time recorded in the SWF (the target running time). However, the original versions of NPB applications do not provide a straightforward way to adjust computational amounts and running time. Therefore, our modifications involve two steps: first, modifying the NPB applications to allow changes in computational amounts and running time, and second, reproducing the target running time.

NPB applications are primarily constructed of loop parts. Since the loop variables represent time and space, we focus on the time loop, expecting that we can linearly change the running time while maintaining the memory footprint by adjusting the time loop length.

First, we select applications from the NPB suite that include time integration for modification. From a programming perspective, the main loop is constructed with time as the outermost variable. We modify these applications to dynamically specify the outermost loop length (ITR). The modified applications include `LU, BT, MG, and SP`.

First, we extract the application with a part of time integral for modification. In programming view, main loop is constructed where outermost variable is time. We modify the applications to dynamically specify the outermost loop length (ITR). Such applications are `LU, BT, MG, and SP`.

Second, we need to determine the appropriate ITR to reproduce the target running time in advance. To achieve this, we develop an running time predictor model. We conduct sampling of the applications, taking an application-parallelism pair and executing the application on a dedicated node with various ITR values. We then model the linear regression expression for each pair. For example, for the pair (app = LU.B, parallelism = 4), we execute with $ITR = 1000, 4000, 8000$ and generate the linear regression model from the results. We validate the model by ensuring that all batch jobs in the experimental workload replicate the target running time within $\pm 8\%$ precision.

### Broadcast Synchronization

Broadcast Synchronization (BS) is a runtime gang scheduler for oversubscribed jobs in oversubscribing systems. As described in Section 2.2.2, this approach originates from previous research [113]. The scheduler, shown as "gang scheduler" in Figure 7.2, accesses the Slurm scheduling information to perform gang scheduling and broadcast the active job ID list to all compute nodes. Client agents on the compute nodes receive this list and switch processes of oversubscribed jobs accordingly. The active jobs operate in synchronization, coordinating both intranode and internode processes within a job.

While Slurm offers a similar function (SP) as discussed in Section 7.4.4, BS demonstrates superior performance. Table 7.4 presents results from a basic oversubscribing scenario. Our gang scheduling BS consistently delivers stable and enhanced performance, whereas the existing SP exhibits unstable and inferior performance.

**Table 7.4:** Comparison of application performance under different resource sharing approaches (Naive, SP, and BS), using NPB LU benchmark with 12 parallel processes. Performance degradation is calculated relative to ideal sharing (50% of baseline speed).

| Metric | Trial | Single speed | Speed with sharing (Naive) | Speed with sharing (SP) | Speed with sharing (BS) |
|---|---|---|---|---|---|
| | 1 | 18.18 | 2.17 + 2.22 | 12.99 + 5.49 | 8.40 + 8.33 |
| | 2 | 17.86 | 2.09 + 2.04 | 13.16 + 5.49 | 8.00 + 8.06 |
| Speed | 3 | 17.54 | 1.93 + 1.96 | 7.19 + 6.76 | 8.20 + 8.33 |
| | 4 | 17.24 | 2.10 + 2.03 | 12.99 + 5.65 | 8.33 + 8.26 |
| | 5 | 17.54 | 1.74 + 1.73 | 7.19 + 6.76 | 8.13 + 8.06 |
| Median Speed | - | 17.24 | 2.10 + 2.03 | 12.99 + 5.49 | 8.20 + 8.33 |
| Performance degradation | - | - | 76.1% | 7.2% | 4.1% |

We anticipate that this approach can provide fair scheduling for jobs, and we introduce the metric *usage interval distribution* to assess this fairness. The usage interval is defined as follows: when a job uses compute resources during two successive timeslices, the usage interval is one. When a job uses compute resources after being idle for one timeslice, the interval is two. This value increases with the number of idle timeslices. By counting these intervals, we obtain the usage interval distribution for each job. In busy workloads, the mode of the usage interval distribution should be the maximum multiplicity of oversubscribing, as this represents the fairest scheduling. Our software can output scheduling information for each timeslice, allowing us to derive the usage interval distribution through analysis.

## 7.5 Evaluation of oversubscribing scheduling on physical systems

All jobs in our experiments completed successfully without encountering memory constraints or unexpected issues.

### 7.5.1 Waiting time of all jobs

Figure 7.5 illustrates the maximum waiting time for jobs of both types (interactive and batch). Waiting times of one or two seconds are considered negligible; we can assume jobs with such short waiting times start immediately. Table 7.5 presents the ratio of jobs that experience waiting time.

In MQ solutions, as the number of interactive nodes increases, the maximum waiting time for interactive jobs decreases while that for batch jobs increases, indicating a trade-off between interactive and batch job performance, as shown in Sections 5.4 and 6.4.

In SQ solutions, we observe that oversubscribing reduces maximum waiting times. There is no significant difference between SQ ($M = 2$, Naive) and SQ ($M = 2$, BS) because the systems can accommodate the required resources with an oversubscribing multiplicity of 2.

Additionally, the experimental workload trace used in this study was relatively light. With an oversubscribing multiplicity of two ($M = 2$), all jobs commenced immediately. It is expected that results for oversubscribing multiplicities of three and greater ($M \geq 3$) would be identical to those for $M = 2$ due to Slurm's distribution policy as shown in 7.4.4.

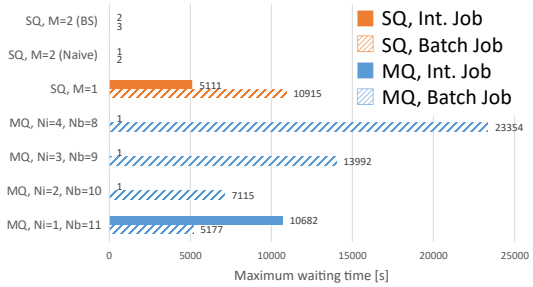



**Figure 7.5:** Maximum waiting times for interactive and batch jobs in MQ and SQ systems.

**Table 7.5:** Percentage of interactive jobs that experience more than five seconds of waiting time in MQ and SQ systems. Lower percentages indicate better system responsiveness for interactive workloads.

| Case | Ratio | Case | Raio |
|---|---|---|---|
| SQ, M=2 (BS) | 0% | MQ, Ni=4, Nb=8 | 0% |
| SQ, M=2 (naive) | 0% | MQ, Ni=3, Nb=9 | 0% |
| SQ, M=1 | 43% | MQ, Ni=2, Nb=10 | 0% |
| | | MQ, Ni=1, Nb=11 | 29% |

### 7.5.2 Efficiency of processing of workload

Figure 7.6 presents a cumulative distribution function of finished jobs, as defined in Section 5.3.2. The horizontal axis indicates the time elapsed since the first job submission, while the vertical axis represents the ratio of finished jobs at each point in time. The values combine both types of jobs. There are minor differences between cases, except for MQ ($N_i = 4, N_b = 8$). These results confirm that the side effects of oversubscribing are tolerable. Table 7.6 provides detailed values, including the 90th and 95th percentiles and makespan.

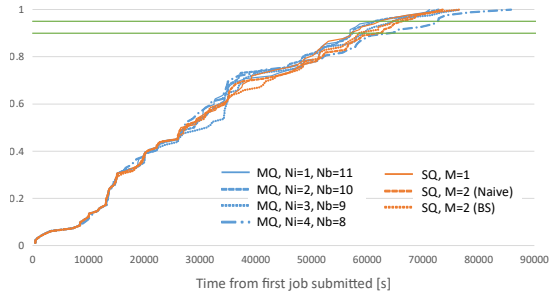



**Figure 7.6:** Cumulative distribution function of job completion over time in MQ and SQ systems.

**Table 7.6:** Job completion time metrics (in seconds) showing the 90th percentile, 95th percentile, and 100th percentile (makespan) in MQ and SQ systems.

| | MQ, Ni=1, Nb=11 | MQ, Ni=2, Nb=10 | MQ, Ni=3, Nb=9 | MQ, Ni=4, Nb=8 | SQ, M=1 | SQ, M=2 (Naive) | SQ, M=2 (BS) |
|---|---|---|---|---|---|---|---|
| 90th percentile [s] | 56747 | 57220 | 59268 | 64169 | 56894 | 62123 | 58407 |
| 95th percentile [s] | 61301 | 61171 | 64630 | 72474 | 62153 | 66053 | 64355 |
| 100th percentile [s] (makespan) | 70960 | 72569 | 76465 | 85573 | 73180 | 73323 | 76278 |

### 7.5.3 Performance analysis of individual jobs

Figure 7.7 illustrates the maximum slowdown for both types of jobs. In MQ solutions, the slowdown primarily reflects the waiting time because oversubscribing among jobs is considered unlikely, as mentioned in Section 7.4.1. Consequently, the trade-off in slowdown between interactive and batch jobs is due to waiting time, as shown in Figure 7.5. For interactive jobs with $N_i \geq 2$, the slowdown is approximately one, indicating no collision occurs between jobs' busy periods.

In SQ solutions, the slowdown primarily reflects running time degradation ($M \geq 2$) since waiting times are nearly eliminated, as shown in Figure 7.5. For batch jobs, while SQ ($M = 2$, Naive) exhibits significant performance degradation due to inefficient resource sharing, the use of coordinated scheduling in SQ ($M =$

2, BS) successfully mitigates this degradation. However, even with BS, some jobs still exceed the ideal slowdown value ($2.00 + \alpha$). For interactive jobs, the maximum slowdown in all SQ configurations is determined by a single job (jobid = 5030). As detailed in Figure 7.8, this job contains long busy periods relative to idle periods, exhibiting resource utilization patterns similar to batch jobs in terms of resource utilization.

Figure 7.9 presents the slowdown distribution for cases MQ ($N_i = 2, N_{10}$) and SQ ($M = 2$, BS). The distributions exhibit markedly different patterns, reflecting their distinct scheduling approaches. While MQ shows a more concentrated distribution due to its dedicated resource allocation strategy, SQ displays a wider spread owing to its resource-sharing nature. To better understand the source of these performance differences, we analyze the degradation ratio, which compares actual running time to dedicated running time (Figure 7.10). In MQ systems, despite CPU core dedication, some applications experience performance degradation due to shared memory channel resources as discribed in Section 7.4.1. Notably, the jobs showing significant degradation ratios are not necessarily the same ones exhibiting high slowdown values. In SQ systems, the degradation ratio becomes equivalent to slowdown since waiting times are effectively eliminated. Seven jobs show degradation ratios exceeding 2.00, which warrants closer examination. Table 7.7 presents the usage interval distribution for these jobs. The distribution reveals that Broadcast Synchronization achieves fair resource allocation, with usage intervals of one and two dominating the distribution. This suggests that the observed performance degradation originates from application-specific characteristics rather than scheduling inefficiencies.

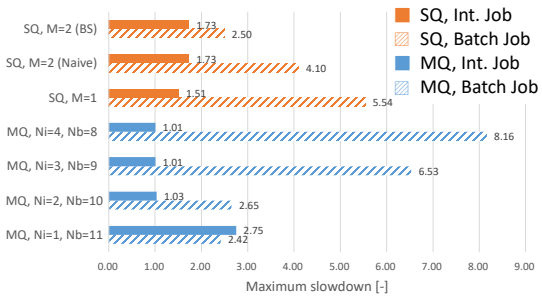



**Figure 7.7:** Maximum slowdown for interactive and batch jobs in MQ and SQ systems. For SQ solutions, slowdown primarily reflects running time degradation ($M \geq 2$) since waiting times are effectively eliminated.

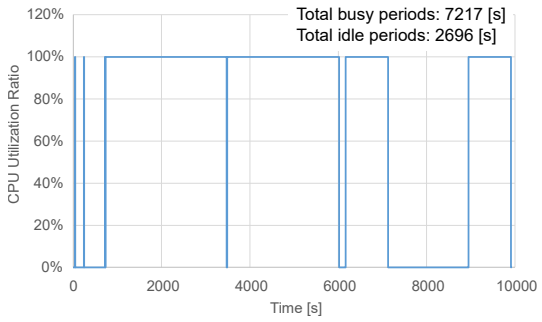
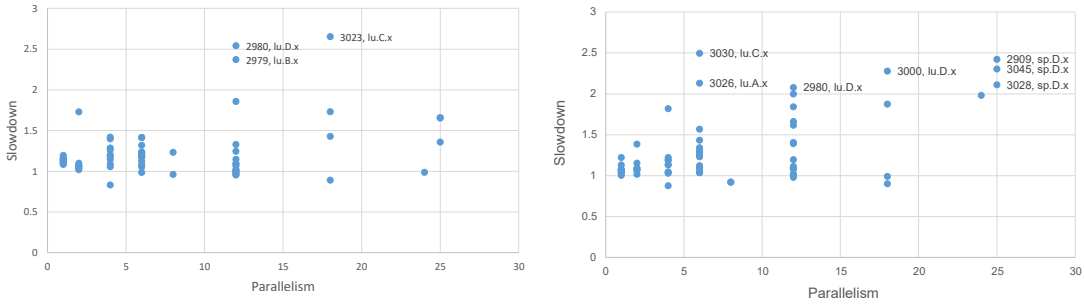


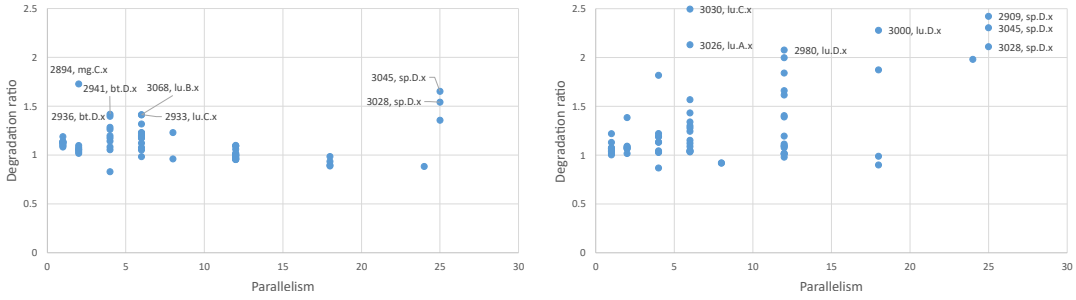

**Figure 7.8:** Resource utilization pattern of interactive job (jobid = 5030) executed by EIBA, following the format specified in Table 7.3. Total busy time: 7,217 s, idle time: 2,696 s.




(a) Slowdown at MQ, Ni = 2, Nb = 10

(b) Slowdown at SQ, M = 2, BS

**Figure 7.9:** Slowdown distribution in MQ and SQ systems. Jobs experiencing significant slowdown are labeled with their job ID and application type.




(a) Performance degradation ratio at MQ, Ni = 2, Nb = 10

(b) Performance degradation ratio at SQ, M = 2, BS

**Figure 7.10:** Distribution of performance degradation ratios in MQ and SQ systems. Jobs experiencing significant degradation are labeled with their job ID and application type.

**Table 7.7:** Detailed analysis of jobs experiencing significant performance degradation (degradation ratio > 2.0) in the SQ (M=2, BS) configuration. Usage intervals show the distribution of resource allocation patterns, where interval=1 indicates consecutive timeslice usage and larger intervals indicate gaps between resource allocations.

| JobId | App | Degradation ratio | Usage Interval Distribution | | | |
|---|---|---|---|---|---|---|
| | | | usage interval = 1 | usage interval = 2 | usage interval = 3 | usage interval = 4 |
| 2909 | sp.D, 25 | 2.42 | 959 | 16679 | 6 | 0 |
| 2980 | lu.D, 12 | 2.08 | 0 | 3730 | 0 | 0 |
| 3000 | lu.D, 18 | 2.28 | 0 | 16144 | 5 | 2 |
| 3026 | lu.A, 6 | 2.13 | 0 | 2837 | 2 | 0 |
| 3028 | sp.D, 25 | 2.11 | 5217 | 12347 | 4 | 0 |
| 3030 | lu.C, 6 | 2.50 | 0 | 3320 | 2 | 0 |
| 3045 | sp.D, 25 | 2.31 | 4931 | 9135 | 0 | 1 |

### 7.5.4 Discussion on system configuration

When considering the optimal configuration, the most critical factor is the response time for interactive jobs; other performance aspects should be considered only after meeting responsiveness requirements. Therefore, the optimal configuration is MQ ($N_i = 2, N_b = 10$), while MQ ($N_i = 3, N_b = 9$) is considered the next optimal (tolerable) option. MQ ($N_i = 1, N_b = 11$) fails to meet the responsiveness requirements, while MQ ($N_i = 4, N_b = 8$) exhibits greater degradation of batch jobs.

In our experimental conditions, SQ ($M = 2$, BS) satisfies both responsiveness and performance criteria without requiring additional configuration. The success in maintaining fair resource allocation in complex resource sharing scenarios derived from production traces (as evidenced by the usage interval distributions) demonstrates that gang scheduling remains effective even with multiple concurrent jobs.

The choice between these configurations ultimately depends on specific institutional requirements, but our results suggest that SQ with BS provides a promising default configuration, particularly when administrative simplicity is valued. While our experiments are limited to a specific workload pattern, they demonstrate that oversubscribing with appropriate gang scheduling can achieve performance comparable to carefully tuned multiple queue configurations.

### 7.5.5 Discussion on the system experiment results compared to simulation results

This section discusses the precision of simulation through comparison between physical system experiment and simulation. First of all, we mention the models in simulation - performance model, scheduling cost model, and so on. The models are just extraction of real physical environment, which may cause the different results:

- Simplified performance model in simulation: The model primarily focuses on the multiplicity factor, assuming that performance degradation scales with the number of co-located jobs. While the actual implementation involves time-division scheduling at different system layers as described in Section 3.2, our simulation abstracts these complex scheduling mechanisms into a simple performance impact factor. Additional effects such as overhead of oversubscribing, which reflects the context switching, cache effect, or network performance losses, is represented by a constant value, 1.20.

- Simplified scheduling cost model in simulation: Overhead of scheduling cost time, mainly when the scheduler calculates the priority and chooses the appropriate compute resources slot, is zero seconds.

- Fixed execution time for jobs: Execution time may vary even when time reproducing applications are used and under dedicated condition in physical system experiments, which invites the different scheduling results.

The second one, the scheduling cost is light weight in the experiment as shown in Figure 7.5. In the experiments with $M = 2$, nearly zero waiting time for all jobs implicates the tiny scheduling cost. To discuss third point, we show the Table 7.8 examining scheduling place results during initial experiment stage, even not occurred oversubscribing. Job 2849 used different places between simulation and physical experiment because there is no occupancy on node 1. Predecessor job on node 1 is job 2829, the finished times are 6,955 for simulation and 7,020 for physical experiment. It is a little bit of difference but it invite the different scheduling place since the former is before incoming job 2849 and the latter is after job 2849. In this way, the scheduling place may change, which implicates the coexisting jobs are different and even whether a job experiences the oversubscribing or not. From the discussion, it is not meaningful to compare each jobs' results and observation of statistical values in traces.

Table 7.9 shows the primary statistical metrics as mentioned in Table 7.5, Figure 7.5, Figure 7.6, and Figure 7.7 for SQ M=2. Oversubscribing can provide the immediate commence of interactive jobs, eliminate the waiting time, and achieve alleviated slowdown. The close agreement between simulated maximum slowdown (2.40 for batch jobs) and physical results (2.50 for batch jobs with BS) validates our simplified performance model's ability to capture the essential characteristics of resource sharing, despite its abstractions. Figure 7.11 and right side of Table 7.9 shows a system efficiency. It shows tiny difference between simulation and physical experiment.

**Table 7.8:** Job scheduling results with simulated and physical metrics. The notation "0(0,1)" in "place" column means that the job uses 0th and 1st cores on 0th node. For example, job 2844 used cores 0-8 on nodes 7 and 8.

| Job ID | Arrival time | Described execution time | Finished time (Sim) | Finished time (Phys) | Place(Sim) | Place(Phys) | Same place |
|---|---|---|---|---|---|---|---|
| 2814 | 0 | 1800 | 1800 | 1928 | 0(0,1) | 0(0,1) | T |
| 2815 | 173 | 2724 | 2897 | 3150 | 1(0,1) | 1(0,1) | T |
| 2816 | 234 | 113 | 347 | 349 | 2(0,1) | 2(0,1) | T |
| 2817 | 280 | 13 | 293 | 295 | 3(0,1) | 3(0,1) | T |
| 2818 | 473 | 527 | 1000 | 1047 | 2(0,1) | 2(0,1) | T |
| 2819 | 571 | 14410 | 14981 | 16097 | 3(0,1,2,3,4,5,6,7,8,9,10,11) | 3(0,1,2,3,4,5,6,7,8,9,10,11) | T |
| 2823 | 2250 | 14410 | 16660 | 18304 | 0(0,1,2,3,4,5,6,7,8,9,10,11) | 0(0,1,2,3,4,5,6,7,8,9,10,11) | T |
| 2829 | 3444 | 3511 | 6955 | 7020 | 1(0,1,2,3,4,5,6,7,8,9,10,11) | 1(0,1,2,3,4,5,6,7,8,9,10,11) | T |
| 2835 | 3772 | 14410 | 18182 | 17899 | 2(0,1,2,3,4,5,6,7,8,9,10,11) | 2(0,1,2,3,4,5,6,7,8,9,10,11) | T |
| 2838 | 4138 | 14410 | 18548 | 19907 | 4(0,1,2,3,4,5,6,7,8,9,10,11) | 4(0,1,2,3,4,5,6,7,8,9,10,11) | T |
| 2839 | 4140 | 14410 | 18550 | 19913 | 5(0,1,2,3,4,5,6,7,8,9,10,11) | 5(0,1,2,3,4,5,6,7,8,9,10,11) | T |
| 2842 | 5211 | 2703 | 7914 | 8273 | 6(0,1,2,3) | 6(0,1,2,3) | T |
| 2844 | 5679 | 2938 | 8617 | 8325 | 7(0,1,2,3,4,5,6,7,8)<br>8(0,1,2,3,4,5,6,7,8) | 7(0,1,2,3,4,5,6,7,8)<br>8(0,1,2,3,4,5,6,7,8) | T |
| 2849 | 6966 | 3072 | 10038 | 9791 | 1(0,1,2,3,4,5,6,7) | 9(0,1,2,3,4,5,6,7) | F |
| 2854 | 8942 | 154 | 9096 | 9077 | 6(0,1,2,3) | 1(0,1,2,3) | F |

**Table 7.9:** Primary metrics in simulation results in SQ (M=2) on simulation

| Percentage of interactive jobs that start immediately [%] | Maximum waiting time of all jobs [s] | Maximum slowdowns [-] | | Job completion time metrics (percentiles) [s] | | |
|---|---|---|---|---|---|---|
| | | For batch jobs | For interactive jobs | 90th | 95th | 100th |
| 100% | 0 | 2.40 | 2.02 | 58578 | 67223 | 74769 |


**Figure 7.11:** Cumulative distribution function of job completion over time in simulation and physical experiment. The configuration in physical experiment is SQ, M = 2 (BS).

# 8 Conclusion

## Contents



This chapter concludes the dissertation by summarizing the research contributions and discussing their implications for HPC system design and operation. First, we review the background and objectives of this research, focusing on the challenges posed by increasingly diverse workloads in modern HPC environments. Next, we examine our methodological approach and its novelty in addressing these challenges. We then present our key findings and contributions, demonstrating how they advance both theoretical understanding and practical implementation of oversubscribing in HPC environments. The chapter concludes with a discussion of future research directions and remaining challenges in this field.

## 8.1 Research background and objectives

The evolution of High-Performance Computing (HPC) workloads, particularly the increasing prevalence of AI/ML applications, has fundamentally changed the demands placed on modern supercomputing systems. As discussed in Chapter 1, while these systems have historically excelled at handling traditional batch-oriented scientific computations, they now face unprecedented challenges in supporting diverse computational patterns, especially interactive workflows essential for AI/ML development.

This diversification of workloads has created a fundamental tension in HPC system operation. Traditional scheduling approaches, optimized for batch processing, often result in significant waiting times that impede productive interactive work. While various solutions have been attempted, such as dedicated interactive nodes or preemption mechanisms, these approaches often lead to either resource inefficiency or operational complexity. Furthermore, the increasing proportion of interactive workloads, driven by AI/ML development needs, has made these limitations increasingly problematic.

The primary objective of this research was to validate oversubscribing as a comprehensive solution to these challenges in HPC environments. Specifically, we aimed to demonstrate that OSub could simultaneously address user productivity needs through immediate resource access and system efficiency requirements through tolerable resource utilization, while maintaining operational simplicity.

## 8.2 Methodological approach and novelty

Our research approached these challenges through a novel, comprehensive evaluation framework that emphasized both theoretical rigor and practical applicability. As outlined in Chapter 3, we adopted a straightforward implementation strategy centered on a single control parameter - the multiplicity factor ($M$) - while maintaining flexibility in time-sharing implementation through support for gang scheduling and configurable time quanta.

The novelty of our approach lay in its multi-dimensional evaluation methodology that addressed multiple aspects of OSub implementation:

- Rigorous analysis of inter-job resource interference through hardware counter analysis and performance modeling
- Comprehensive workload system evaluation incorporating both user experience metrics and operational concerns

- Assessment of operational feasibility including management complexity and deployment strategies

- Physical system validation through actual implementation and experimentation in a real HPC environment

Unlike previous research that often relied on simplified workload models or focused solely on individual application performance, our framework incorporated:

- Detailed analysis of inter-job resource interference using hardware performance counters

- System-wide evaluation using real production workload traces

- Explicit consideration of interactive job characteristics based on actual usage patterns

- Empirical validation through physical system implementation and controlled experiments

This comprehensive approach, combining both simulation studies and physical system validation, enabled us to bridge the gap between theoretical potential and practical implementation, providing insights crucial for HPC centers considering OSub adoption. The framework's effectiveness was demonstrated throughout our research, with simulation studies evaluating OSub performance across varying interactive workload ratios, while physical system experiments provided crucial validation of its core operational characteristics in real production environments.

## 8.3 Key research findings and contributions

Our research has provided three major contributions that significantly advance understanding of oversubscribing in HPC environments, each supported by systematic empirical evidence.

### 8.3.1 Demonstration of users/operators benefits

First, we demonstrated that oversubscribing can simultaneously benefit both users and system operators while maintaining acceptable performance trade-offs. Through detailed performance analysis in Chapter 4, we showed that performance degradation under OSub remains predictable and manageable, with detailed characterization of both parallel and sequential application behavior under resource sharing conditions. Our hardware counter analysis revealed key patterns in resource interference, enabling development of predictive models for performance impacts.

The system-level simulations presented in Chapter 5 validated these benefits at scale, demonstrating:

- Significant reduction in job waiting times while maintaining system throughput
- Performance impacts remain within tolerable bounds without complex parameter tuning
- Acceptable performance trade-offs even under aggressive oversubscribing configurations

These findings were further validated through physical system experiments in Chapter 7, which demonstrated:

- Consistent achievement of predicted performance characteristics in real operational settings
- Practical viability of immediate job execution through resource sharing
- Successful maintenance of system efficiency under various workload conditions

### 8.3.2 Empirical assessment of OSub effects with data from production environments

Second, our research provided the first comprehensive characterization of OSub behavior in real HPC environments. The detailed performance analysis in Chapter 4 revealed actual interference patterns between co-located jobs, while Chapter 5's system-level evaluation demonstrated OSub's effectiveness using real workload traces.

This empirical foundation was significantly strengthened through the physical system validation presented in Chapter 7, which provided:

- Direct evidence of OSub's practical implementability in production environments

- Verification of performance predictions through controlled experiments
- Demonstration of operational stability under real workload conditions

Particularly significant was Chapter 6's analysis of interactive job characteristics, which revealed:

- Typical interactive jobs comprise around 10 busy periods
- Most busy periods last less than 30 seconds
- 71% of jobs have busy period ratios below 0.1

These findings, validated through both simulation studies and physical system experiments, provided crucial empirical support for OSub's viability in production environments.

### 8.3.3 Validation in interactive-job-dominant environments

Third, Chapter 6 demonstrated OSub's effectiveness in environments where interactive jobs constitute the majority of workloads. Our analysis showed that:

- OSub maintained stable performance for both batch and interactive jobs without parameter tuning
- Performance remained predictable even with interactive job ratios up to 76
- Traditional multiple-queue approaches required complex optimization as interactive job ratios increased

This validation in interactive-job-dominant scenarios is particularly significant given the growing importance of AI/ML workflows in modern HPC environments.

## 8.4 Future challenges and prospects

While this research has demonstrated the effectiveness of OSub in HPC systems and paved the way for its practical implementation, several challenges remain for broader adoption. These challenges can be categorized into two distinct areas: first, developmental challenges that extend the approaches demonstrated in this research, and second, new research directions that build upon our findings.

### 8.4.1 Developmental challenges

The most critical and immediate challenge for advancing OSub adoption lies in its validation within larger-scale systems. While our research has provided compelling evidence in small-scale environments, comprehensive testing in large production facilities is essential to establish OSub's viability as a general solution for modern HPC operations. This expanded validation would demonstrate scalability across different system architectures and generate long-term operational data crucial for both strategic planning and daily operations. System operators would gain the empirical evidence needed to justify OSub deployment in major facilities, while research teams would receive concrete assurance about performance consistency for their large-scale computations, particularly critical for extensive AI training workloads and long-running scientific simulations. This validation represents the fundamental next step in establishing OSub as a standard resource management approach in the HPC community.

The second developmental challenge involves comprehensive memory capacity simulation and management. As discussed in Section 6.4.2, our analysis revealed that memory capacity, rather than CPU sharing, becomes the primary limiting factor when scaling to higher interactive workload ratios. While our current research demonstrated that CPU sharing remains effective due to the intermittent nature of interactive jobs (with most showing less than 10% average CPU utilization), memory demands accumulate directly as concurrent jobs increase. This fundamental constraint requires detailed investigation starting with comprehensive measurement of actual memory usage patterns in interactive jobs - specifically, analyzing peak memory requirements and temporal variations in memory consumption throughout job lifetimes. The development of memory usage prediction models, combined with real-time monitoring capabilities, would allow schedulers to make more informed decisions about concurrent job limits.

The third challenge involves developing sophisticated resource utilization optimization mechanisms. The current research has demonstrated basic resource sharing effectiveness, but achieving optimal efficiency requires advanced monitoring and dynamic control systems. Automated parameter tuning mechanisms would continuously optimize resource allocation based on workload characteristics, significantly reducing administrative overhead while maintaining performance objectives. Real-time visibility into resource utilization patterns would transform how computational workflows are planned and executed, enabling both system operators and researchers to make more informed decisions about resource allocation and job scheduling.

The fourth developmental challenge focuses on integrating performance prediction capabilities with scheduling systems. This integration would enable proactive resource management decisions that prevent performance degradation while maximizing system utilization. The scheduling system could make intelligent decisions about job co-location based on predicted interference patterns, particularly valuable for managing diverse workloads including GPU computations and emerging parallel programming models. This enhanced predictability would benefit the entire user community through more efficient resource allocation, while providing system operators with automated tools for maintaining service quality across varying workload conditions.

### 8.4.2 New research directions

Beyond the immediate developmental challenges, this research opens several promising new directions that could fundamentally transform HPC resource management. These directions extend beyond mere improvements to existing approaches, potentially revolutionizing how we think about shared computing infrastructure.

The first promising direction lies in establishing performance-cost tradeoff models that draw inspiration from cloud computing while respecting HPC's unique characteristics. This research could develop sophisticated frameworks where resource allocation policies consider both performance guarantees and operational costs. Such models would enable flexible resource sharing policies where users could choose different levels of performance guarantees based on their specific needs - from guaranteed dedicated resources for critical production runs to shared resources for development work. The resulting system would maintain the high-performance capabilities essential for scientific computing while introducing beneficial flexibility in resource allocation and pricing strategies.

A second innovative direction involves comprehensive workflow optimization. While current OSub implementations focus on individual job management, future research should address the optimization of entire computational workflows. Intelligent resource sharing across dependent jobs could significantly reduce end-to-end completion times for complex scientific workflows. This approach would particularly benefit AI/ML development cycles, where workflows typically involve multiple phases from data preprocessing through model training to inference deployment. By understanding these workflow dependencies and characteristics, scheduling systems could make sophisticated decisions about resource allocation that optimize the entire development cycle rather than individual components.

The third direction explores the integration of power-aware computing with resource sharing strategies. As energy costs and environmental concerns become increasingly critical in HPC operations, future research should investigate how OSub can contribute to power efficiency without compromising computational capability. Dynamic resource sharing policies could consider power consumption patterns alongside traditional performance metrics, potentially achieving significant energy savings through intelligent workload consolidation. This research direction would address both environmental sustainability goals and operational cost reduction, particularly crucial for large-scale computing facilities.

# References


[1] K. Miyamoto, S. Yamazaki, F. Uchida, K. Fujisawa, and N. Yoshida, "Quantum algorithm for the vlasov simulation of the large-scale structure formation with massive neutrinos," *Phys. Rev. Res.*, vol. 6, p. 013 200, 1 2024. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevResearch.6.013200.

[2] S. Tanaka, S. Tokutomi, R. Hatada, K. Okuwaki, K. Akisawa, K. Fukuzawa, Y. Komeiji, Y. Okiyama, and Y. Mochizuki, "Dynamic cooperativity of ligand–residue interactions evaluated with the fragment molecular orbital method," *The Journal of Physical Chemistry B*, vol. 125, no. 24, pp. 6501–6512, 2021, PMID: 34124906. eprint: https://doi.org/10.1021/acs.jpcb.1c03043. [Online]. Available: https://doi.org/10.1021/acs.jpcb.1c03043.

[3] Y. Tamaki and S. Kawai, "Wall-resolved les of near-stall airfoil flow at re c = $10^7$ using the supercomputer fugaku," in *AIAA SCITECH 2022 Forum*. eprint: https://arc.aiaa.org/doi/pdf/10.2514/6.2022-1060. [Online]. Available: https://arc.aiaa.org/doi/abs/10.2514/6.2022-1060.

[4] Y. Otsuka, Y. Tamari, K. Fujita, and T. Ichimura, "Large-scale 3d seismic response analysis considering soil liquefaction in an urban area using the supercomputer "fugaku"," *Japanese Geotechnical Society Special Publication*, vol. 10, no. 46, pp. 1735–1740, 2024.

[5] S. Y. Shigeki Kaneko Naoto Mitsume, "Large-scale 3d thermal transfer analysis with 1d model of piped cooling water," *Digital Engineering and Digital Twin*, vol. 2, no. 1, pp. 33–48, 2024. [Online]. Available: http://www.techscience.com/dedt/v2n1/55358.

[6] T. Miyoshi, A. Amemiya, S. Otsuka, Y. Maejima, J. Taylor, T. Honda, H. Tomita, S. Nishizawa, K. Sueki, T. Yamaura, Y. Ishikawa, S. Satoh, T. Ushio, K. Koike, and A. Uno, "Big data assimilation: Real-time 30-second-refresh heavy rain forecast using fugaku during tokyo olympics and paralympics," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23, Denver, CO, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3581784.3627047.

[7] T. Ichimura, K. Fujita, R. Kusakabe, K. Koyama, S. Murakami, Y. Kikuchi, T. Hori, M. Hori, H. Inoue, T. Nose, T. Kawashima, and M. Lalith, "Extreme scale earthquake simulation with uncertainty quantification," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2022, pp. 1–11.

[8] S. of AI Report, *State of ai report compute index*, Accessed: 2024-12-18, 2024. [Online]. Available: https://www.stateof.ai/compute.

[9] TOP500.org, *TOP500 Supercomputer Sites*, `https://top500.org/`, Accessed: 2024-05-17, 2024.

[10] H. W. Meuer, E. Strohmaier, J. Dongarra, and H. D. Simon, *The TOP500: History, Trends, and Future Directions in High Performance Computing*, 1st. Chapman & Hall/CRC, 2014.

[11] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, *et al.*, "Jupyter Notebooks – a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, IOS Press, 2016, pp. 87–90.

[12] A. Reuther, T. Currie, J. Kepner, H. Kim, A. McCabe, P. Michaleas, and N. Travinin, "Technology requirements for supporting on-demand interactive grid computing," in *2005 Users Group Conference (DOD-UGC'05)*, 2005, pp. 320–327.

[13] A. Reuther, J. Kepner, C. Byun, S. Samsi, W. Arcand, D. Bestor, *et al.*, "Interactive supercomputing on 40,000 cores for machine learning and data analysis," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, 2018, pp. 1–6.

[14] S. L. Harrell, J. Kitson, R. Bird, S. J. Pennycook, J. Sewall, D. Jacobsen, D. N. Asanza, A. Hsu, H. C. Carrillo, H. Kim, and R. Robey, "Effective performance portability," in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2018, pp. 24–36.

[15] S. Kamil, J. Shalf, and E. Strohmaier, "Power efficiency in high performance computing," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–8.

[16] D. H. Ahn, J. Garlick, M. Grondona, D. Lipari, B. Springmeyer, and M. Schulz, "Flux: A next-generation resource management framework for large hpc centers," in *2014 43rd International Conference on Parallel Processing Workshops*, 2014, pp. 9–17.

[17] A. Reuther, C. Byun, W. Arcand, D. Bestor, B. Bergeron, M. Hubbell, M. Jones, P. Michaleas, A. Prout, A. Rosa, and J. Kepner, "Scalable system scheduling for hpc and big data," *Journal of Parallel and Distributed Computing*, vol. 111, pp. 76–92, 2018. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0743731517301983`.

[18] J. Bonneau, C. Herley, P. C. v. Oorschot, and F. Stajano, "The quest to replace passwords: A framework for comparative evaluation of web authentication schemes," in *2012 IEEE Symposium on Security and Privacy*, 2012, pp. 553–567.

[19] C. Christmann, E. Hebisch, and A. Weisbecker, "Oversubscription of computational resources on multicore desktop systems," in *Multicore Software Engineering, Performance, and Tools*, V. Pankratius and M. Philippsen, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 18–29.

[20] S. A. Baset, L. Wang, and C. Tang, "Towards an understanding of oversubscription in cloud," in *2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 12)*, San Jose, CA: USENIX Association, 2012. [Online]. Available: `https://www.usenix.org/conference/hot-ice12/workshop-program/presentation/baset`.

[21] S. Hofmeyr, C. Iancu, J. Colmenares, E. Roman, and B. Austin, "Time-sharing redux for large-scale hpc systems," in *IEEE HPCC/SmartCity/DSS 2016*, 2016, pp. 301–308.

[22] C. Byun, W. Arcand, D. Bestor, B. Bergeron, V. Gadepally, M. Houle, M. Hubbell, M. Jones, A. Klein, P. Michaleas, L. Milechin, J. Mullen, A. Prout, A. Reuther, A. Rosa, S. Samsi, C. Yee, and J. Kepner, "Node-based job scheduling for large scale simulations of short running jobs," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 2021, pp. 1–7.

[23] R. A. Tau Leng, J. Hsieh, V. Mashayekhi, and R. Rooholamini, "An empirical study of hyper-threading in high performance computing clusters," *Linux HPC Revolution*, vol. 45, 2002.

[24] S. Minami, T. Endo, and A. Nomura, "Measurement and modeling of performance of HPC applications towards overcommitting scheduling systems," in *Workshop on Job Scheduling Strategies for Parallel Processing*, Springer, 2021, pp. 59–79.

[25] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The Nas parallel benchmarks," *The International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[26] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, and V. Venkatakrishnan, "The nas parallel benchmarks," NASA Ames Research Center, Moffett Field, CA, USA, Tech. Rep. RNR-94-007, 1994, Available at `https://www.nas.nasa.gov/publications/npb.html`.

[27] S. Matsuoka, "Fugaku and a64fx: The first exascale supercomputer and its innovative arm cpu," in *2021 Symposium on VLSI Circuits*, 2021, pp. 1–3.

[28] S. Yamamura, Y. Akizuki, H. Sekiguchi, T. Maruyama, T. Sano, H. Miyazaki, and T. Yoshida, "A64fx: 52-core processor designed for the 442petaflops supercomputer fugaku," in *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 65, 2022, pp. 352–354.

[29] M. Sato, Y. Ishikawa, H. Tomita, Y. Kodama, T. Odajima, M. Tsuji, H. Yashiro, M. Aoki, N. Shida, I. Miyoshi, K. Hirai, A. Furuya, A. Asato, K. Morita, and T. Shimizu, "Co-design for a64fx manycore processor and "fugaku"," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–15.

[30] S. Matsuoka, T. Endo, A. Nukada, S. Miura, A. Nomura, H. Sato, H. Jitsumoto, and A. Drozd, "Overview of TSUBAME3.0, green cloud supercomputer for convergence of HPC, AI and big-data," *TSUBAME e-Science Journal*, vol. 16, pp. 2–9, 2017.

[31] K. Nakajima, T. Furumura, F. Boillod-Cerneux, E. Di Napoli, E. Suarez, T. Arakawa, S. Sumimoto, and H. Yashiro, "Innovative computational science by integration of simulation/data/learning on heterogeneous supercomputers," in *2024 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*, 2024, pp. 154–155.

[32] Y. Ajima, "High-dimensional interconnect technology for the K computer and the supercomputer fugaku," *Fujitsu Technical Review*, 2020.

[33] P. MacArthur, Q. Liu, R. D. Russell, F. Mizero, M. Veeraraghavan, and J. M. Dennis, "An integrated tutorial on infiniband, verbs, and mpi," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2894–2926, 2017.

[34] J. J. Dongarra, "The linpack benchmark: An explanation," in *Supercomputing*, E. N. Houstis, T. S. Papatheodorou, and C. D. Polychronopoulos, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 456–474.

[35] W.-c. Feng and K. Cameron, "The green500 list: Encouraging sustainable supercomputing," *Computer*, vol. 40, no. 12, pp. 50–55, 2007.

[36] A. B. Yoo, M. A. Jette, and G. Mark, "Slurm: Simple linux utility for resource management," in *Workshop on job scheduling strategies for parallel processing*, Springer, 2003, pp. 44–60.

[37] R. L. Henderson, "Job scheduling under the portable batch system," in *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 279–294.

[38] G. Borges, M. David, J. A. T. Gomes, J. López, P. Rey, Á. Simón, C. Fernández, D. Kant, and K. Sephton, "Sun grid engine , a new scheduler for egee middleware," 2007. [Online]. Available: https://api.semanticscholar.org/CorpusID:8013055.

[39] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA: USENIX Association, 2011. [Online]. Available: https://www.usenix.org/conference/nsdi11/dominant-resource-fairness-fair-allocation-multiple-resource-types.

[40] D. Jackson, Q. Snell, and M. Clement, "Core algorithms of the maui scheduler," in *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 87–102.

[41] R. Priedhorsky and T. Randles, "Charliecloud: Unprivileged containers for user-defined software stacks in hpc," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17, Denver, Colorado: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3126908.3126925.

[42] O. Arndt, B. Freisleben, T. Kielmann, and F. Thilo, "A comparative study of online scheduling algorithms for networks of workstations," *Cluster Computing*, vol. 3, no. 2, pp. 95–112, 2000. [Online]. Available: https://doi.org/10.1023/A:1019024019093.

[43] D. A. Lifka, "The ANL/IBM SP scheduling system," in *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 295–303.

[44] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan, "Characterization of backfilling strategies for parallel job scheduling," in *Proceedings. International Conference on Parallel Processing Workshop*, 2002, pp. 514–519.

[45] B. G. Lawson and E. Smirni, "Multiple-queue backfilling scheduling with priorities and reservations for parallel systems," *SIGMETRICS Perform. Eval. Rev.*, vol. 29, no. 4, pp. 40–47, 2002. [Online]. Available: https://doi.org/10.1145/512840.512846.

[46] D. Talby and D. Feitelson, "Supporting priorities and improving utilization of the ibm sp scheduler using slack-based backfilling," in *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999*, 1999, pp. 513–517.

[47] R. Kettimuthu, V. Subramani, S. Srinivasan, T. Gopalsamy, D. Panda, and P. Sadayappan, "Selective preemption strategies for parallel job scheduling," *International Journal of High Performance Computing and Networking*, vol. 3, no. 2-3, pp. 122–152, 2005. eprint: https://www.inderscienceonline.com/doi/pdf/10.1504/IJHPCN.2005.008032. [Online]. Available: https://www.inderscienceonline.com/doi/abs/10.1504/IJHPCN.2005.008032.

[48] W. Smith, I. Foster, and V. Taylor, "Scheduling with advanced reservations," in *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, 2000, pp. 127–132.

[49] Y. Georgiou, E. Jeannot, G. Mercier, and A. Villiermet, "Topology-aware job mapping," *The International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 14–27, 2018.

[50] D. G. Feitelson and L. Rudolph, "Gang scheduling performance benefits for fine-grain synchronization," *Journal of Parallel and Distributed Computing*, vol. 16, no. 4, pp. 306–318, 1992. [Online]. Available: https://www.sciencedirect.com/science/article/pii/074373159290014E.

[51] S.-H. Chiang, A. Arpaci-Dusseau, and M. K. Vernon, "The impact of more accurate requested runtimes on production job scheduling performance," in *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 103–127.

[52] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes," in *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2009, pp. 427–436.

[53] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong, "Theory and practice in parallel job scheduling," in *Workshop on Job Scheduling Strategies for Parallel Processing*, Springer, 1997, pp. 1–34.

[54] U. Ayachit, *The ParaView Guide: A Parallel Visualization Application*. Clifton Park, NY, USA: Kitware, Inc., 2015.

[55] E. Bisong, "Google colaboratory," in *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners*. Berkeley, CA: Apress, 2019, pp. 59–64. [Online]. Available: https://doi.org/10.1007/978-1-4842-4470-8_7.

[56] R. Keryell, R. Reyes, and L. Howes, "Khronos SYCL for OpenCL: a tutorial," in *Proceedings of the 3rd International Workshop on OpenCL*, ser. IWOCL '15, Palo Alto, California: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2791321.2791345.

[57] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[58] L. Dagum and R. Menon, "Openmp: An industry standard api for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[59] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. Cambridge, MA, USA: MIT Press, 1994.

[60] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0167819196000245.

[61] L. Dalcin, R. R. Paz, P. A. Kler, and M. S. Cosimo, "Mpi for python: Performance improvements and mpi-2 extensions," *Journal of Parallel and Distributed Computing*, vol. 68, no. 5, pp. 655–662, 2008.

[62] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for?" *Queue*, vol. 6, no. 2, pp. 40–53, 2008. [Online]. Available: https://doi.org/10.1145/1365490.1365500.

[63] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation," *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012.

[64] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A LLVM-based python JIT compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM '15, New York, NY, USA: ACM, 2015. [Online]. Available: https://numba.pydata.org/.

[65] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," in *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter*, 1995, pp. 19–25.

[66] I. Corporation, *Intel MPI Benchmarks User Guide*, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-mpi-benchmarks.html.

[67] T. O. S. University, *Osu micro-benchmarks (omb)*, The Ohio State University, 2021. [Online]. Available: https://mvapich.cse.ohio-state.edu/benchmarks/.

[68] L. LLNL and A. S. Center, *Ior: Parallel i/o benchmark*, Lawrence Livermore National Laboratory and Los Alamos National Laboratory, 2021. [Online]. Available: https://ior.readthedocs.io/.

[69] W. C. Skamarock, J. B. Klemp, J. Dudhia, D. O. Gill, D. M. Barker, W. Wang, and J. G. Powers, "A Description of the Advanced Research WRF Model Version 4," *National Center for Atmospheric Research (NCAR) Technical Note*, 2019.

[70] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton, "LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Computer Physics Communications*, vol. 271, p. 108 171, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010465521002836.

[71] P. Giannozzi, S. Baroni, N. Bonini, *et al.*, "QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials," *Journal of Physics: Condensed Matter*, vol. 21, no. 39, p. 395 502, 2009.

[72] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–16.

[73] A. Sergeev and M. Del Balso, "Horovod: Fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[74] M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess, and E. Lindahl, "GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers," *SoftwareX*, vol. 1-2, pp. 19–25, 2015.

[75] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009. [Online]. Available: https://doi.org/10.1145/1498765.1498785.

[76] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 157–173.

[77] H. Yabuuchi, D. Taniwaki, and S. Omura, "Low-latency job scheduling with preemption for the development of deep learning," in *2019 USENIX Conference on Operational Machine Learning (OpML 19)*, 2019, pp. 27–30.

[78] W. Chen, X. Zhou, and J. Rao, "Preemptive and low latency datacenter scheduling via lightweight containers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 12, pp. 2749–2762, 2020.

[79] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Nimble page management for tiered memory systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19, Providence, RI, USA: Association for Computing Machinery, 2019, pp. 331–345. [Online]. Available: https://doi.org/10.1145/3297858.3304024.

[80] C. Li, A. Zigerelli, J. Yang, Y. Zhang, S. Ma, and Y. Guo, "A dynamic and proactive gpu preemption mechanism using checkpointing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 1, pp. 75–87, 2020.

[81] J. Kehne, J. Metter, and F. Bellosa, "Gpuswap: Enabling oversubscription of gpu memory through transparent swapping," *SIGPLAN Not.*, vol. 50, no. 7, pp. 65–77, 2015. [Online]. Available: https://doi.org/10.1145/2817817.2731192.

[82] Y. Fan, Z. Lan, P. Rich, W. Allcock, and M. E. Papka, "Hybrid workload scheduling on hpc systems," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022, pp. 470–480.

[83] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," *Journal of Physics: Conference Series*, vol. 46, no. 1, p. 494, 2006. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/46/1/067.

[84] G. Bronevetsky, D. Marques, K. Pingali, P. Szwed, and M. Schulz, "Application-level checkpointing for shared memory programs," ser. ASPLOS XI, Boston, MA, USA: Association for Computing Machinery, 2004, pp. 235–247. [Online]. Available: https://doi.org/10.1145/1024393.1024421.

[85] S. R. Alam, R. F. Barrett, J. A. Kuehn, P. C. Roth, and J. S. Vetter, "Characterization of scientific workloads on systems with multi-core processors," in *2006 IEEE International Symposium on Workload Characterization*, 2006, pp. 225–236.

[86] H. Shan, K. Antypas, and J. Shalf, "Characterizing and predicting the i/o performance of hpc applications using a parameterized synthetic benchmark," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008, pp. 1–12.

[87] J. K. Ousterhout *et al.*, "Scheduling techniques for concurrent systems.," in *ICDCS*, vol. 82, 1982, pp. 22–30.

[88] F. Corbato, M. Merwin-Daggett, and R. Daley, "Ctss-the compatible time-sharing system," *IEEE Annals of the History of Computing*, vol. 14, no. 1, pp. 31–54, 1992.

[89] V. Vyssotsky and F. J. Corbató, " Introduction and Overview of the Multics System," *IEEE Annals of the History of Computing*, vol. 14, no. 02, pp. 12–13, 1992. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/MAHC.1992.10016.

[90] J. E. Moreira, H. Franke, W. Chan, L. L. Fong, M. A. Jette, and A. Yoo, "A gang-scheduling system for asci blue-pacific," in *Proceedings of the 7th International Conference on High-Performance Computing and Networking*, ser. HPCN Europe '99, Berlin, Heidelberg: Springer-Verlag, 1999, pp. 831–840.

[91] C. Iancu, S. Hofmeyr, F. Blagojević, and Y. Zheng, "Oversubscription on multicore processors," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010, pp. 1–11.

[92] D. G. Feitelson and M. A. Jettee, "Improved utilization and responsiveness with gang scheduling," in *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 238–261.

[93] A. B. Yoo and M. A. Jette, "An efficient and scalable coscheduling technique for large symmetric multiprocessor clusters," in *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 21–40.

[94] P. Jacquet, T. Ledoux, and R. Rouvoy, " SlackVM: Packing Virtual Machines in Oversubscribed Cloud Infrastructures," in *2024 IEEE International Conference on Cluster Computing (CLUSTER)*, Los Alamitos, CA, USA: IEEE Computer Society, 2024, pp. 190–201. [Online]. Available: `https://doi.ieeecomputersociety.org/10.1109/CLUSTER59578.2024.00024`.

[95] J. Yin, X. Lu, H. Chen, X. Zhao, and N. N. Xiong, "System resource utilization analysis and prediction for cloud based applications under bursty workloads," *Information Sciences*, vol. 279, pp. 338–357, 2014. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0020025514004204`.

[96] A. Iosup, N. Yigitbasi, and D. Epema, "On the performance variability of production cloud services," in *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2011, pp. 104–113.

[97] N. Purohit, P. Srivastava, V. Tripathi, and N. Mohd, "Spot pricing in cloud computing: A comprehensive survey of mechanisms, strategies, and future directions," in *Data Science and Network Engineering*, S. Namasudra, M. C. Trivedi, R. G. Crespo, and P. Lorenz, Eds., Singapore: Springer Nature Singapore, 2024, pp. 331–346.

[98] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 157–173.

[99] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, *et al.*, "Score-P: A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir," in *Tools for High Performance Computing 2011*, Springer, 2012, pp. 79–91.

[100] S. Minami, *Node conscious ovesubscribing scheduler simulator*. [Online]. Available: `https://github.com/iwturnedaiw/NodeConsciousScheduler`.

[101] S. J. Chapin, W. Cirne, D. G. Feitelson, J. P. Jones, S. T. Leutenegger, U. Schwiegelshohn, W. Smith, and D. Talby, "Benchmarks and standards for the evaluation of parallel job schedulers," in *IPPS/SPDP '99/JSSPP '99*, 1999, pp. 67–90.

[102] D. G. Feitelson, D. Tsafrir, and D. Krakov, "Experience with using the parallel workloads archive," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2967–2982, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731514001154.

[103] *The university of Luxemburg Gaia cluster log*, 2014. [Online]. Available: https://www.cs.huji.ac.il/labs/parallel/workload/l_unilu_gaia/index.html.

[104] *The KIT ForHLR II log*, 2018. [Online]. Available: https://www.cs.huji.ac.il/labs/parallel/workload/l_kit_fh2/index.html.

[105] M. Hardt, X. Chen, X. Cheng, *et al.*, "Amazon sagemaker clarify: Machine learning bias detection and explainability in the cloud," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, ser. KDD '21, Virtual Event, Singapore: Association for Computing Machinery, 2021, pp. 2974–2983. [Online]. Available: https://doi.org/10.1145/3447548.3467177.

[106] Microsoft, *Azure machine learning*, https://azure.microsoft.com/services/machine-learning/, 2015.

[107] D. Schultz and B. Jain, "Nonsmooth analysis and subgradient methods for averaging in dynamic time warping spaces," *Pattern Recognition*, vol. 74, pp. 340–358, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0031320317303163.

[108] O. Serbetci, *Synthetic time series generation for training simple multi-layer-perceptron classifier*, https://oguzserbetci.github.io/generate-time-series/.

[109] C. Byun, W. Arcand, D. Bestor, B. Bergeron, V. Gadepally, M. Houle, M. Hubbell, M. Jones, A. Klein, P. Michaleas, L. Milechin, J. Mullen, A. Prout, A. Reuther, A. Rosa, S. Samsi, C. Yee, and J. Kepner, "Node-based job scheduling for large scale simulations of short running jobs," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 2021, pp. 1–7.

[110] A. Reuther, C. Byun, W. Arcand, D. Bestor, B. Bergeron, M. Hubbell, M. Jones, P. Michaleas, A. Prout, A. Rosa, and J. Kepner, "Scheduler technologies in support of high performance data analysis," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, 2016, pp. 1–6.

[111] N. A. Simakov, R. L. Deleon, Y. Lin, P. S. Hoffmann, and W. R. Mathias, "Developing accurate slurm simulator," in *Practice and Experience in Advanced Research Computing 2022: Revolutionary: Computing, Connections, You*, ser. PEARC '22, Boston, MA, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3491418.3535178.

[112] U. Lublin and D. G. Feitelson, "The workload on parallel supercomputers: Modeling the characteristics of rigid jobs," *Journal of Parallel and Distributed Computing*, vol. 63, no. 11, pp. 1105–1122, 2003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731503001084.

[113] H. Ohtsuji, E. Hayashi, R. Kinoshita, M. Miwa, and E. Yoshida, "Scalable Fine-Grained Gang Scheduling for HPC Systems with Unreliable Broadcast Synchronization Mechanisms," in *The International Conference for High Performance Computing, Networking, Storage, and Analysis (Poster)*, 2023.