National Technical University of Athens
School of Electrical and Computer Engineering
Computer Science Division
Computing Systems Laboratory

Experiences in Deploying Ephemeral Slurm as a Slurm Job and Co-execution Analysis using ¼-socket CPU Allocation

Diploma Thesis

Floros Epameinondas

Supervisor: George Goumas, Professor NTUA

Athens, October 2025

National Technical University of Athens
School of Electrical and Computer Engineering
Computer Science Division
Computing Systems Laboratory

Experiences in Deploying Ephemeral Slurm as a Slurm Job and Co-execution Analysis using ¼-socket CPU Allocation

Diploma Thesis

Floros Epameinondas

Supervisor: George Goumas, Professor NTUA

Approved by the three-member examination committee on 30 October 2025

George Goumas	Nectarios Koziris	Dionisios Pnevmatikatos
Professor, NTUA	Professor, NTUA	Professor, NTUA

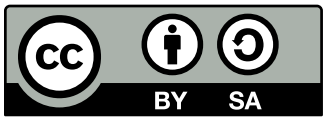
Athens, October 2025

National Technical University of Athens
School of Electrical and Computer Engineering
Computer Science Division
Computing Systems Laboratory

Υπεύθυνη Δήλωση Για Λογοκλοπή Και Για Κλοπή Πνευματικής Ιδιοκτησίας

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτή τη Διπλωματική εργασία είναι του συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών ή του Εθνικού Μετσόβιου Πολυτεχνείου.

Φλώρος Επαμεινώνδας, Αθήνα 2025
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π



Περίληψη

Στη σύγχρονη επιστήμη αλλά και στις πρακτικές εφαρμογές λογισμικού, είναι πολύ μεγάλη η συνεισφορά των συστημάτων υψηλής επίδοσης (High Performance Computing clusters). Εξαιτίας των μεγάλων απαιτήσεων των συστημάτων αυτών σε ενέργεια αλλά και με δεδομένη την εντεινόμενη ζήτηση για υπηρεσίες που μπορούν να παρέχουν μόνο τέτοια συστήματα, κρίνεται αναγκαία η βέλτιστη αξιοποίησή των πόρων τους και η βελτιστοποίηση των επιμέρους λειτουργιών τους.

Από τη σκοπιά του σχεδιαστή τέτοιων συστημάτων μέχρι αυτή του τελικού χρήστη, υπάρχει ανάγκη για εστίαση στην ευελιξία της διαμόρφωσης των συστημάτων διαχείρισης πόρων, εστιάζοντας στις περιοχές της επεκτασιμότητας και της προσαρμοστικότητας του συστήματος στα υπολογιστικά φορτία. Το Slurm, αποτελεί το πιο διαδεδομένο λογισμικό διαχείρισης πόρων συστημάτων υψηλής επίδοσης, και βασίζεται στο kernel του Linux για τη λειτουργία του. Οι βασικές λειτουργίες του συνοψίζονται στη διαχείριση δικαιωμάτων χρηστών επί των πόρων του συστήματος, στην παροχή ενός πλαισίου για την εκτέλεση των εργασιών που υποβάλλονται σε αυτό και, τέλος, στη διαμεσολάβηση για τη χρήση των πόρων του συστήματος.

Στόχος της παρούσας διπλωματικής εργασίάς είναι η ανάπτυξη ενός εργαλείου, χωρίς την ανάγκη για αυξημένα διαχειριστικά δικαιώματα ώστε να καταστεί εφικτή η δοκιμή επεκτάσεων και τροποποιήσεων για το σύστημα διαχείρισης πόρων του Slurm. Επιπλέον, πραγματοποιείται η πειραματική αξιολόγηση της συνεκτέλεσης εργασιών MPI σε συνθήκες μοιραζόμενων πόρων, ιδιαίτερα σε επίπεδο socket.

Η επιδίωξη από το εργαλείο είναι να επεκτείνει το Slurm, με τέτοιο τρόπο, ώστε να καταστεί δυνατή η αξιολόγηση επεκτάσεων και εναλλακτικών υλοποιήσεων των διάφορων λειτουργιών του. Επιπλέον, παρουσιάζεται μία πρακτική αξιολόγηση της ορθής λειτουργίας του εργαλείου και της ικανότητάς του να ενσωματωθεί, με τρόπο αξιόπιστο και ακριβή, σε παραγωγικά συστήματα υψηλών επιδόσεων με λειτουργικές διανομές Slurm.

Εν κατακλείδι, οι επιδόσεις των εργασιών MPI αξιολογούνται με χρήση ποσοτικών δεικτών για την εξακρίβωση της αποτελεσματικότητας και της βιωσιμότητας της συνεκτέλεσης σε συστήματα υψηλών επιδόσεων, ιδιαίτερα όταν πραγματοποιείται σε εκτεταμένη κλίμακα.

Λέξεις Κλειδιά:

HPC, Χρονοδρομολόγηση, Σύστημα Διαχείρισης Πόρων, SLURM, συντοποθέτηση,συν-τοποθέτση, συνεκτέλεση, συν-εκτέλεση, μετρήσεις επίδοσης, MPI, bash

Abstract

High Performance Computing (HPC) clusters have a major contribution in scientific and commercial software. Energy requirements of such systems and growing demand drive the need for constant optimisation in resource utilisation as well as expanding and improving individual components of HPC systems.

At present, the necessity for the optimal utilisation of resources of high performance computing systems is the trailblazer for change in current and future resource management systems. Starting from the perspective of the system designer up to the end user, it is essential to focus on the flexible aspect of configuring resource management systems, directing attention to the fields of extendability and adaptability of HPC systems to the multitude of workload types. Slurm, an open source software based on the Linux kernel, is currently holding the title of the most used and renowned resource management software for HPC systems. Slurm's basic functionality is summarised in three functions, namely access and privilege management, providing a framework for the submission and execution of workloads, and, finally mediation and resolution for the underlying system's resources.

The aims of this thesis to develop a tool in a user-space environment, without the need for elevated administrator rights, for testing extensions and modifications to the Slurm resource management system, as well as, investigate the co-execution effects among MPI workloads that share common hardware resources, specifically at socket level.

The tool's objective is to extend Slurm so that, in the environment of a real functioning HPC system, it will allow the evaluation of extensions and alternative implementations of Slurm's various components as well as extract insights and data for the operation of these components. In addition, an hands-on evaluation study of the tool is presented, both for the accuracy of its results and its reliability to integrate in production systems with existing Slurm installations.

Finally, the performance of various MPI workloads is assessed using quantitative metrics to evaluate the viability and efficiency of co-executing computational tasks in HPC environments. These results provide a foundation for understanding the sustainability and potential benefits of workload co-execution in large-scale systems.

Keywords:

HPC, Scheduling, Resource Management System, SLURM, co-location, colocation, co-execution, coexecution, benchmarking, MPI, bash

Acknowledgements

This thesis would not be possible without:

George Goumas, Professor NTUA, in whom I saw one of the most inspiring professors in my 15 years in academia. I will never forget that I registered on his OS course because it was obligatory and by the time the first lecture ended I had a cause and a course on my path as an electrical and computer engineer.

The support and guidance of **Nikos Triantafyllis**, who endured my many eccentricities with unwavering patience. In times of deep desperation regarding the thesis course he always had a positive attitude and a new idea to try, which kept me on track.

My companion in life, **Katerina**, to whom disproportionate burden has fallen. I thank her for the solidarity and love she has shown me through all these years. Tireless, supportive and kind, this feat would not be possible without her.

My friends **Thimios** and **Petros**, whose company I deeply appreciate and have greatly missed during these years. They are always a source of inspiration and reflection. Wherever our paths may lead, I know that in the end I can always depend on them.

Finally, to my family, **Charilaos**, **Marianthe** and **Io**, who encouraged me to pursue my ambition to study ECE and endured my unbearable temper. The values I hold dear are owed to them.

I am indebted to the struggles of countless people that managed to keep, even for this brief historical moment, Greek higher education public and -mostly- free. Without them, people like me would never get a chance on education.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation - Problem Statement

High Performance Computing (HPC) clusters underpin a wide range of scientific and commercial applications. Their growing energy footprint and increasing workload diversity drive a continual need to improve resource utilization and to extend system capabilities. Central to this effort is the design of resource management systems that can flexibly accommodate new scheduling policies, custom accounting modules and novel security mechanisms without compromising the stability of a production environment.

As HPC installations grow from dozens to thousands of nodes, static allocation policies and coarse-grained scheduling lead to increased queue wait times, resource fragmentation and suboptimal utilisation. Scaling intensifies contention for network and storage subsystems, making it harder to predict performance and balance load across the cluster. Without a mechanism to test alternative allocation strategies under realistic load, administrators and researchers cannot quantify potential gains or identify problematic cases in the algorithm design phase.

Demand for HPC services continues to rise across domains—from climate modeling to machine learning[1]—driving bursts of heterogeneous workloads that mix batch jobs, interactive analyses and long-running simulations. Traditional resource managers lack the flexibility to adapt their internal heuristics dynamically, forcing users and operators to rely on manual tuning, static profiles and policy-based management that quickly become obsolete as workload patterns shift.

Another layer of complexity is added to the problem from emerging hybrid systems that integrate quantum accelerators alongside classical compute[2]. Quantum tasks often require tight co-scheduling with classical pre- and post-processing, as well as specialized resource reservations. Current resource managers offer minimal support for quantum-classical workflows, and any extension must be validated in situ to ensure correct orchestration across both domains.

Slurm (Simple Linux Utility for Resource Management), an open-source resource manager for Linux-based clusters, dominates the current HPC landscape. Its core services—access control, workload submission and execution framework, and resource mediation—are well established, yet modifying or extending SLURM normally requires administrator privileges and changes to a live system. This poses a barrier to rapid prototyping and rigorous evaluation of new components.

The problem addressed in this thesis is the lack of a user-space environment in which Slurm extensions can be developed and tested safely on a functioning HPC installation. The

solution proposed here is a tool that intercepts and redirects Slurm's internal interfaces in user space, enabling the deployment and measurement of alternative implementations without root access or disruption to existing workflows. An experimental study evaluates both the fidelity of the tool's emulation and its capability to integrate with the existing Slurm infrastructure.

To add to that, a study on the effects of co-locating different workloads is conducted in advance, in order to probe into the different applications that more advanced scheduling algorithms, integrated into existing resource systems, could lead to.

1.2 Thesis Structure

The remainder of this thesis is organized as follows:

- **Chapter 2** reviews fundamental concepts in HPC resource management, surveys common bottlenecks and examines Slurm's architecture and security model.
- **Chapter 3** discusses co-location strategies and existing approaches to resource allocation optimisation in related work.
- **Chapter 4** presents the design of *Slurm-in-Slurm* (SiS), particularly detailing its conceptualisation on ARIS HPC and overall system architecture.
- **Chapter 5** describes the implementation of SiS in native user-space environment, and explains integration with a running SLURM installation.
- **Chapter 6** reports a proof-of-concept validation of SiS operating in a production environment and acting as an intermediary tool for Slurm jobs. Moreover, the chapter recites experimental results for quarter-socket MPI workloads. Finally, it summarizes the findings, discusses limitations and outlines directions for future research.
- **Chapter 7** provides an extended overview of the thesis in Greek.

Chapter 2

HPC Resource Management

2.1 Bottlenecks in HPC Research and Utilisation

Despite continued progress in processor and system architecture, several critical bottlenecks persist in HPC systems. These limitations affect scalability, efficiency, and usability across scientific and industrial domains.

Understanding and addressing the existing bottlenecks of HPC is critical to improving application efficiency, energy proportionality, and scalability. Current HPC research and practice have been focusing on constraints related to memory access, interconnects, storage, energy consumption, heterogeneity, and parallel scalability. These bottlenecks are not isolated but often interact, compounding performance limitations across system components. Nevertheless, HPC research itself is limited by the availability of production HPC systems for research purposes regarding the systems themselves. Generally, HPC systems retain high levels of utilisation during all times due to the importance and time span of the several research and commercial applications that need them for operational reasons.

A brief introduction the the several bottlenecks of HPC are presented in this section.

While computation capabilities have been improving constantly during past years, memory bandwidth and latency remain fundamental constraints[3]. Modern processors often stall waiting for data due to limited throughput [4] and inefficiencies in the memory hierarchy. From a design viewpoint, memory incurs 20% of the total system cost for modern HPC systems[5], thus, proper management and resource allocation of the memory subsystem is crucial for contemporary HPC systems. Data movement across memory levels and between compute nodes incurs substantial energy and time costs [6]. These factors dominate performance and energy consumption in data-intensive workloads, especially at large scales.

Related to the issue of memory access is the technology and capabilities of interconnects. Most of the Top 500 HPC systems [7] are a collection of interconnected commercially available CPUs and/or GPUs, thus, the science and technology progressing interconnects is of outmost importance. Data movement across the processing units dominates the overall time that an application executing in an HPC system needs to reach completion (memory-bound calculations), and this kind of application profile is far more frequent than computation time profiles, in which computation dominates the total completion time (computation-bound calculations)[8]. As of the writing of this dissertation thesis, apart from the less frequent proprietary interconnects, the interconnects which are more common in the Top 500 systems are:

- *Infiniband* - the industry standard currently holding a 54.2% share of HPC interconnects.
- Gigabit Ethernet - A well known technology, oftentimes used alongside other intercon-

nects with higher throughput.

- HPE Slingshot - 7 out of the top 10 HPC systems in the Top 500 list use the HPE Slingshot interconnect.
- Intel Omnipath

The performance of HPC systems is increasingly constrained by a significant data storage bottleneck, creating an "I/O wall" where computational speed far outpaces the storage subsystem's ability to deliver and persist data. This disparity forces powerful processors into idle states while waiting for data, drastically reducing overall application efficiency and scientific throughput. This long-standing issue has been exacerbated in the exascale era[1], where massive parallelism and data-intensive simulations generate I/O requests at unprecedented rates[9]. Traditional parallel file systems, while scalable, often struggle to handle the highly concurrent I/O patterns, which oftentimes exhibit bursts of I/O requests, of modern scientific applications. Despite recent advances in regulating I/O bursts [10], managing the complex data path across the memory and storage hierarchy remains a central challenge in designing next-generation HPC systems.

The escalating energy demand of HPC is a well-documented trend[11], but has only been pursued as a driver for the design and operation of HPC systems in very recent years[12]. The aggregate performance of the Top 500 HPC systems has grown exponentially, far outpacing improvements in energy efficiency. This has led to individual systems consuming tens of megawatts of power, with the total energy footprint of the HPC industry reaching hundreds of terawatt-hours on an annual basis.

The architectural paradigm of heterogeneity in HPC, which integrates diverse processing units like CPUs and GPUs, introduces substantial performance bottlenecks despite its promise for accelerating computation. A primary challenge is the data movement overhead required to transfer information between the separate memory spaces of CPUs and accelerators, often constrained by the limited bandwidth of interconnects, which can leave powerful processing units idle and starved for data [13]. This hardware reality is compounded by significant programming complexity, as developers must often master and combine discrete programming models (e.g., OpenMP for CPUs and CUDA for GPUs) within a single application. Furthermore, efficiently balancing the computational load between architecturally distinct processors is a non-trivial scheduling problem, where a naive partitioning of work can lead to severe underutilisation of resources as faster units are forced to wait for slower ones to complete their tasks [14]. Successfully mitigating these interconnected bottlenecks in data transfer, programming, and load balancing, remains a critical focus of research to fully harness the power of modern heterogeneous HPC.

As a final remark on bottlenecks faced by HPC systems stands a basic observation that came to be known as Amdahl's Law[15], which dictates that the maximum achievable speedup of any parallel application is ultimately capped by the fraction of its code that is inherently sequential and cannot be parallelized. Amdahl's Law ultimately states that linear performance gains as processor counts increase is not achievable. In practice, as systems scale to thousands or millions of cores, this theoretical limit is compounded by the rapidly growing cost of communication and synchronization.

[1] Exascale computing refers to computing systems capable of calculating at least 1018 IEEE 754 double precision (64-bit) operations (multiplications and/or additions) per second[9].

2.2 SLURM Resource Manager

To try and tackle some of the aforementioned open challenges for HPC, especially those of heterogeneity, interconnections and energy efficiency, resource management systems came into existence, as they play a key role in how a supercomputer's resources are allocated to user applications. A resource management system does not only help distribute workloads of different types and durations, but also offers common interfaces across different types of machines and their configurations, simplifying access and alleviating concerns about portability issues. It provides mechanisms through which computing systems can be accessed by various categories of users (including those outside the institution hosting the cluster, via specialized collaborative environments) while accurately recording resource usage and applying appropriate user billing.

Slurm is an open source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters. Slurm requires no kernel modifications for its operation and is relatively self-contained. As a cluster workload manager, Slurm has three key functions. First, it allocates exclusive and/or non-exclusive access to resources (compute nodes) to users for some duration of time so they can perform work. Second, it provides a framework for starting, executing, and monitoring work (normally a parallel job) on the set of allocated nodes. Finally, it arbitrates contention for resources by managing a queue of pending work, whilst, having the ability to incorporate user plugins for several of its subsystems.[16]

Slurm's origins date back to 2001 where a small team of software engineers lead by Morris Jette in Lawrence Livermore National Laboratory started researching advanced scheduling subsystems for large scale computers. Since then, Slurm's development has progressed impressively, with 200 collaborators contributing on the project as well as multiple institutions, including SchedMD LLC (the company responsible for Slurm's development, support and training), Linux NetworX, HewlettPackard, Groupe Bull, Cray, Barcelona Supercomputing Center, Oak Ridge National Laboratory, Los Alamos National Laboratory, Intel, Nvidia and more.[17]

As a general rule, in HPC clusters, resource management systems are essential software components. Their role centers on three key tasks: distributing resources, scheduling jobs, and tracking their execution. Resource allocation involves assigning the necessary hardware—whether just a small segment of the machine or the full system—to user workloads, depending on the specific requirements. A generic view of the operations upheld by resource managers can be observed on Fig. 2.1

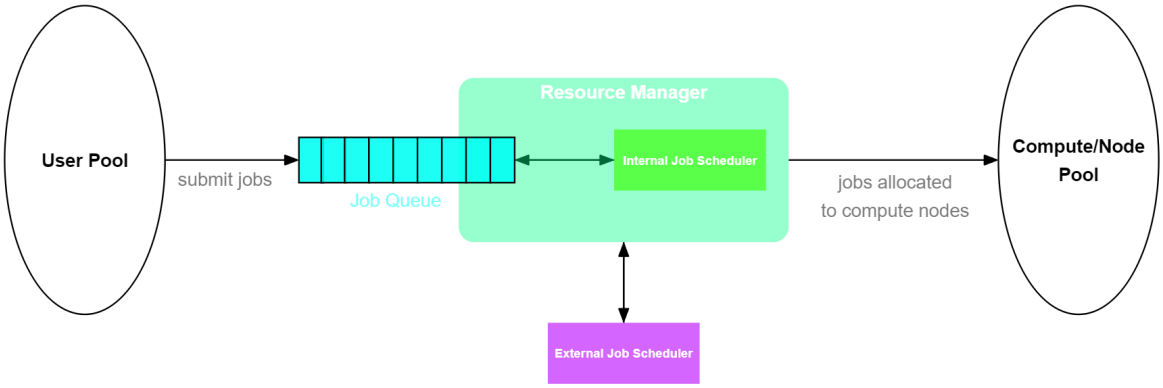


Figure 2.1: A high-level view of the operation of a generic resource manager

Slurm functions in a similar way, providing many more capabilities that will be presented in brief in the next sections. Slurm has a centralized manager, `slurmctld`, to monitor resources and work. There may also be a backup manager to assume those responsibilities in the event of failure. Each compute server (node) has a `slurmd` daemon, which can be compared to a remote shell: it waits for work, executes that work, returns status, and waits for more work. The `slurmd` daemons provide fault-tolerant hierarchical communications. There is an optional `slurmdbd` (Slurm DataBase Daemon) which can be used to record accounting information for multiple Slurm-managed clusters in a single database. There is an optional `slurmrestd` (Slurm REST API Daemon) which can be used to interact with Slurm through its REST API. The described layout of the Slurm concept operation can be seen in Fig. 2.2

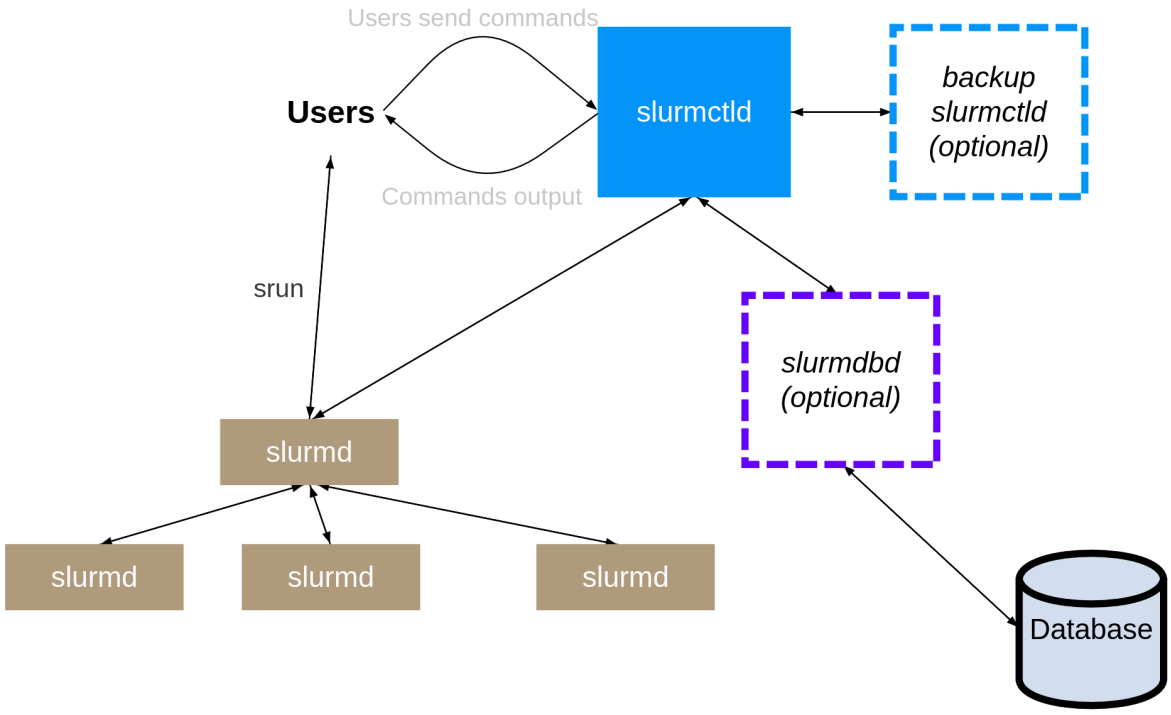


Figure 2.2: Slurm resource manager concept operation

A description of the basic functional units - daemons that consist Slurm and are in charge of its operation is given in subsections 2.2.4 and 2.2.5.

2.2.1 Resource Allocation

In order to manage the complexity of large-scale HPC environments, Slurm is designed with a logical division into distinct entities. This separation of concerns allows Slurm to efficiently coordinate job submission, scheduling, and execution across thousands of compute nodes while maintaining scalability and fault tolerance. By organizing its functionality into well-defined components, Slurm ensures that each entity can operate independently yet cooperatively, enabling both flexibility for system administrators and transparency for end-users. This logical structuring is essential for maintaining performance, reliability, and extensibility in modern HPC systems.

One of the fundamental entities in the system is the *node*. Each node must belong to at least one *partition*, which groups resources into a logical structure. Partitions are logical compartments and do not represent physical divisions. These partitions are not required to be mutually exclusive; in fact, they often overlap. Typically, partitions are used to organize nodes that share similar characteristics, such as comparable hardware configurations or software environments.

To give these structures purpose, Slurm also defines the concepts of the *job* and the *job step*. A job represents an allocation of resources to a user for a specific period of time, while a job step refers to a collection of tasks (which are potentially parallel) that execute within the scope of a job. Each partition can be thought of as a separate job queue, governed by constraints such as maximum job count, time limits, or user access policies and more.

Jobs are scheduled onto nodes within a partition based on priority until the available resources of that partition are exhausted. Once resources are assigned to a job, the user gains full control to launch one or more job steps. These steps can be configured in various ways: for instance, a single step may utilize the entire allocation, or multiple steps may run concurrently, each consuming a subset of the resources assigned to the job.

2.2.2 Scheduling

In high-performance computing environments, jobs pending execution are organized into queues, which govern the sequence in which tasks are dispatched by the resource management system. While many scheduling decisions adhere to the First-In, First-Out (FIFO) policy—reflecting the classical queuing model in computer science, more advanced schedulers incorporate sophisticated strategies aimed at maximizing overall resource efficiency. Typically, systems implement multiple specialized queues, each tailored to a distinct workload profile and associated with specific scheduling constraints. For example, separate queues may exist for interactive jobs, while others may enforce strict limits on wall-clock time, memory consumption, or the number of allocated compute nodes. Although, these more sophisticated methods are usually the subject of rigorous research and investigation, most production systems still employ a mix of simple policies for scheduling jobs.

By default, Slurm adopts comparatively simple scheduling algorithms, consistent with its foundational design principles of simplicity and efficiency, although more recent versions of Slurm have integrated more complex algorithms, alas, the description of which is not in the scope of this dissertation. The scheduling mechanism is event-driven, activated by specific system events such as job completion, job submission, or modifications to the system

configuration. In this mode, the scheduler evaluates only a fixed number of jobs —by default, the first 100 entries in the priority queue— thereby balancing responsiveness with low overhead. Nevertheless, Slurm also provides the capability to consider the entirety of the queued workload. This extended scheduling approach, while more comprehensive, incurs higher computational costs and is therefore executed less frequently, according to a configurable scheduling interval (sched_interval).

For completeness and without immersion into the details of Slurm plugins, it is noted that Slurm supports the integration of scheduling extensions to enrich and customize its default scheduling functionality. These extensions are configured via the `SchedulerType` parameter within the `slurm.conf` configuration file. Although administrators may modify the active scheduling plugin to adapt the system's behavior to specific workload or policy requirements, any such modification necessitates a restart of the `slurmctld` daemon for the changes to take effect.

2.2.3 Security

The security model of Slurm is deliberately minimalistic. All authenticated cluster users are permitted to submit jobs and to cancel only those they have submitted, while also being able to inspect the system's configuration and runtime status. Administrative operations, such as altering configuration parameters, canceling arbitrary jobs, or performing other privileged actions, are restricted to users with elevated rights. By default, this category of privileged users includes only the root account and the Slurm users explicitly defined in the configuration file as such. In scenarios where configuration modifications must be delegated to additional users, privilege escalation mechanisms such as Set-UID programs (e.g., setuid, setgid) are required to grant finely scoped administrative permissions.

Traditionally, node-to-node authentication was handled through the use of reserved ports in combination with Set-UID programs. Daemons would validate incoming requests by verifying that the source port was within a privileged range accessible exclusively to the root user, thereby treating such connections as implicitly trustworthy. This approach, however, was constrained by the scarcity of reserved ports and further weakened by the inherent security risks of Set-UID binaries. To overcome these limitations, Slurm introduces an alternative authentication mechanism that avoids reliance on such legacy methods. In fact, this alteration in the security policy of Slurm was the enabler for the SiS tool developed for the scope of this dissertation! The logic of the contemporary authentication method imposes that each inter-node message in a Slurm cluster is tagged with a unique identifier, which encodes the sender's user ID (uid) and group ID (gid). Once verified by the recipient, these credentials establish the sender's authenticity and authorization level.

As far as job authentication is concerned, Slurm abides by the following mechanism: When the controller allocates resources to a user, it generates a unique identifier for each job step submitted by that user. This identifier encapsulates several elements: the user's UID, the corresponding job ID, the step ID, the set of allocated resources, and a validity period. To ensure security, the identifier is cryptographically signed using the controller's private key. This mechanism authorizes the user to access the designated resources without requiring additional validation queries from the `slurmd` to the `slurmctld`. Instead, the `slurmd` verifies the authenticity of the identifier by checking the controller's digital signature with its public key and confirming that the embedded details match the request. In this way, public key cryptography underpins secure and efficient resource access control within the system.

Finally, Slurm provides mechanisms for partition-level access control. One such mechanism

is the `RootOnly` flag, which, when activated, limits the submission of jobs and resource allocation requests exclusively to privileged users—those whose effective UID grants administrative rights. Users with such privileges are authorized to submit jobs on behalf of other accounts. Furthermore, Slurm supports partition access restrictions based on group membership through the `AllowGroups` directive in the configuration file, enabling administrators to enforce fine-grained policies that align computational resource usage with organizational or project-specific boundaries.

2.2.4 Slurm Controller (slurmctld)

In Slurm, most of the system's state information is managed by the `slurmctld` daemon. This controller is multithreaded and employs separate locking mechanisms for read and write operations to ensure scalability. Upon startup, it loads the system configuration from the configuration file along with any previously stored state data. To support fault tolerance, the controller periodically saves its full state to disk, or immediately if configuration changes occur. The `slurmctld` can operate in either master or standby mode, depending on whether a backup controller (fail-over twin) is present. Unlike the `slurmd` daemon, it does not require root privileges. Instead, Slurm recommends assigning a dedicated system user—defined in the configuration file under `SlurmUser`—to run `slurmctld`. The key internal components of the `slurmctld` are:

- *Node Manager*: The Node Manager is responsible for monitoring the operational status of all nodes within the cluster. This is accomplished either through periodic polling of the downstream (See: section 2.2.5) daemons or by processing asynchronous state updates they provide. Before a node is deemed suitable for task allocation, the Node Manager verifies that it conforms to the expected configuration parameters.
- *Partition Manager*: Groups nodes into the previously described sets referred to as partitions. It assigns nodes to jobs based on their status and configuration. Job start requests originate from the Job Manager. It also modifies the configuration of nodes and partitions according to commands issued from previleged users.
- *Job Manager*: The Job Manager handles job submission requests, maintaining them in a priority queue until resources become available. It is triggered either on a scheduled basis or in response to state transitions that may enable the initiation of pending jobs, the completion of existing workloads, the arrival of new submissions, or the activation of a node or partition. When such conditions arise, the Job Manager selects an appropriate job from the queue for each available partition and allocates the necessary resources. Following allocation, it ensures that all required execution details are communicated to the designated nodes.

2.2.5 Slurm Daemon (slurmd)

The `slurmd` daemons, are multithreaded programs that are attached on the compute nodes of the clusters and are generally in charge of relaying and monitoring the command execution from the `slurmctld` (See: section 2.2.4). Its responsibilities include, but are not limited to, parsing the system configuration from the Slurm configuration file, registering its active status with the controller, awaiting job assignments, executing these jobs, returning execution results, and remaining on standby for subsequent tasks. Because it initiates processes intermediatiating

commands from users, execution under root privileges is required. Furthermore, slurmd communicates asynchronously with the slurmctld, exchanging information about both the status of active jobs and the condition of the hosting node. Its state management is deliberately minimal, maintaining only the data strictly necessary for the jobs currently in execution.

2.3 Other Resource Managers for HPC

At the time of writing of this dissertation more HPC resource managers are available for system administrations, in the scope of this research only a brief presentation of those using open source software will take place.

2.3.1 Flux

An interesting case of HPC resource manager is Flux. Flux is a resource and job management framework developed to address the limitations of traditional high-performance computing (HPC) schedulers in handling heterogeneous hardware, nested workloads, and ensemble-based workflows. Unlike other schedulers such as Slurm or PBS, Flux adopts a hierarchical model that enables the instantiation of nested schedulers within existing allocations. This approach allows for fine-grained scheduling decisions at multiple levels of the system, thereby reducing contention and improving utilization. Its architecture is built around lightweight components, including a key-value store and message broker, that support decentralized decision-making and enhance scalability on large-scale systems[18].

A key contribution of Flux lies in its ability to unify diverse scheduling requirements under a common framework while supporting portability across platforms. Flux's main advantage is that it can operate autonomously under already deployed resource managers in production systems.

2.3.2 PBS(Pro)

PBS (Portable Batch System Professional) is a mature, full-featured workload management and job-scheduling system originally developed under NASA's Portable Batch System (PBS) initiative, designed to manage compute-intensive workloads across diverse environments including parallel clusters and computational grids. As advanced by [19], PBS Pro supports core scheduling features such as advanced reservations, peer scheduling, and cycle harvesting, allowing coordinated resource utilization from idle desktops to supercomputer clusters, while offering fine-grained control over compute and data workflows in Grid computing contexts. Furthermore, PBSPro integrates mechanisms for resource abstraction, authentication, accounting, and data staging, thereby facilitating secure and programmable workload distribution across heterogeneous systems.

Moreover, PBSPro provides a configurable resource-definition framework that allows administrators to declare standard and custom (generic) resources with defined types, scopes, and behaviors, which jobs may request at submission. This enables advanced scheduling logic for specialised resources such as GPUs, memory, node attributes, or licenses. While this flexibility is beneficial, the enforcement of custom-resource constraints may require additional administrative tooling or scripting to ensure proper tracking and isolation[20].

2.3.3 RadicalPilot

RADICAL-Pilot (RP), is Python-based lightweight resource acquisition and scheduling system, not to be confused with a traditional resource manager. RP is used for executing heterogeneous tasks with maximum concurrency and at scale. RP can concurrently execute up to 10^5 heterogeneous tasks, including single/multi core/GPU and MPI/OpenMP. Tasks can be stand-alone executables or Python functions and both types of task can be concurrently executed[21].

RP presents itself as a Pilot system, meaning that it performs a distinction between resource acquisition and using those resources to execute application tasks. RP acquires resources by submitting a job to an HPC platform, and it can directly schedule and launch computational tasks on those resources. Thus, tasks are directly scheduled on the acquired resources, not via the batch system of the HPC platform. RP supports concurrently using single/multiple pilots on single/multiple high performance computing (HPC) platforms[21].

Similar projects are Balsam[22] and Parsl[23].

Chapter 3

Background and Related Work

3.1 Co-execution

In HPC environments, resource allocation policies typically assign jobs from different users to distinct nodes, ensuring both spatial and temporal separation. Although this strategy minimizes performance interference across workloads, it incurs significant drawbacks, notably reduced system throughput and lower energy efficiency. A new approach subjects jobs to occupy part of the nodes in order to make it possible to co-locate other jobs that will improve the overall usage of the node[24].

Studies[25, 26] demonstrate that overall system performance can be enhanced when co-located jobs exhibit heterogeneous resource requirements, such as pairing memory-intensive workloads with compute-intensive ones.

Co-location, from a physical viewpoint, can be seen as process-to-core mapping technique for HPC clusters and can come in two different forms, **striped** and **spread**.

On the contrary, how and which jobs should be co-located in each node is part of bleeding-edge research[27–30], centered around many innovative and advanced methods. For the most part, the resource manager can deploy applications based on the logic followed by the scheduler and based on the scheduler's policy it can be elected that an application can be spread across many nodes and/or allocate it in nodes that other applications are being executed.

Distributing applications across nodes enables them to leverage a greater pool of hardware resources, given that resource contention among different applications is being minimised at the same time, such as additional last-level cache or accelerators like GPUs. Furthermore, intra-node data transfer is optimized because fewer application instances are executed per node, thereby reducing communication overhead—particularly relevant in the context of MPI applications. Node sharing is especially beneficial when co-located applications exhibit complementary resource usage patterns. For example, a memory-bound application, whose scalability is constrained by bandwidth, can achieve shorter execution times when executed alongside a compute-intensive workload, as overall communication demands are diminished. From the system perspective, adopting a strategy of first spreading and then striping applications mitigates resource fragmentation, thereby improving utilization of nodes that would otherwise remain underused[31].

Conversely, heterogeneous applications often exhibit diverse communication patterns, which can lead to performance degradation under certain co-location scenarios. Such slowdowns may result from synchronization overhead across nodes or from cases where co-located workloads ultimately manifest similar behavior. Naturally, this that running two independent jobs of the same application concurrently is inadvisable: memory-bound applications

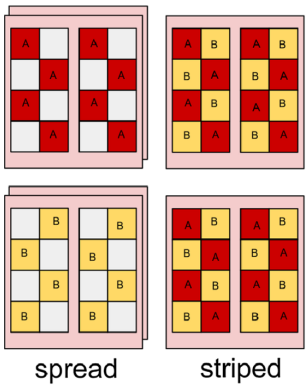


Figure 3.1: Methods for scheduling two 16 process distributed applications A and B on a supercomputer with two 8-core processor sockets. With *spread*, A and B are still isolated but run on two compute nodes each, with half the cores empty. In *striped*, A and B share all sockets of two compute nodes.[24]

suffer from increased contention, while compute-bound applications would achieve higher efficiency if paired with memory-intensive counterparts[32, 33]. An additional challenge arises in practical systems with pricing models that charge users based on the number of cores allocated multiplied by execution time. Under such schemes, users may incur higher costs when scheduling decisions inadvertently slow their applications, raising concerns of fairness. Although various mitigation strategies have been proposed, these are beyond the scope of our present analysis.

The impact of node sharing on application performance can be either beneficial or detrimental, depending largely on workload characteristics. The degree of homogeneity or heterogeneity among co-located jobs significantly influences co-execution efficiency, with neighboring workloads often serving as the primary factor behind performance degradation or optimisation[34]. Consequently, co-scheduling, meaning the implementation of co-location in a dynamic, long-running scenario, emerges as a highly complex problem, as addressing these challenges requires deeper insights into the behavior and requirements of submitted applications. Nonetheless, it may be argued that moderate increases in execution time are acceptable when offset by the advantage of reduced queue waiting times[35].

3.2 Simulating Resource Allocation

On the subject of designing and using state-of-the-art schedulers, especially ones that can manage co-location of workloads on different processors, the Computing Systems Laboratory of the Computer Science Division of National Technical University of Athens has developed a tool for modeling and evaluation scheduling algorithms[36].

Despite concrete data on the benefits of co-location, which are already presented throughout this chapter, there remains a critical gap in understanding how co-scheduling schemes would perform when jobs dynamically enter and exit the system. The complexity of this problem is compounded by a shortage of dedicated tools and deployments in production HPC environments that allow researchers to rapidly and accurately develop, test, and evaluate co-scheduling algorithms without requiring extensive environment configuration. Existing domain-specific and toolkit-based simulators often lack direct support for co-scheduling under space-sharing or are difficult to configure and extend for such purposes. In that ecosystem, simulation emerged as an essential initial step to explore the intricate behaviors of co-scheduling

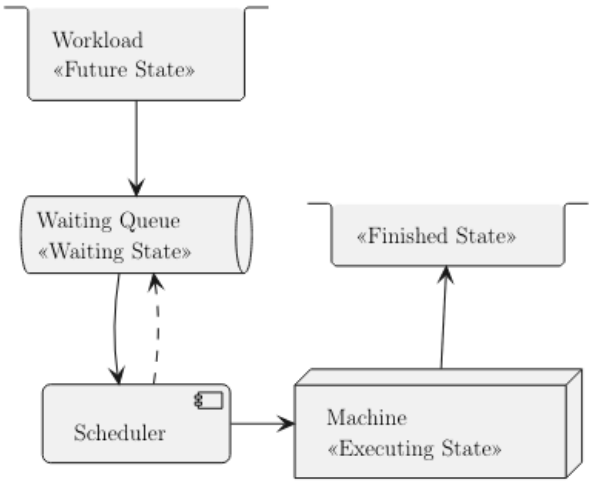


Figure 3.2: High-level design logic of the ELiSE framework[31]

and assess key metrics like system throughput, turnaround times, and resource fragmentation. In response to this pressing need, the Efficient Lightweight Scheduling Estimator (ELiSE), a Python-based framework, specifically designed for the rapid development and comprehensive evaluation of scheduling algorithms with co-location capabilities.

ELiSE's primary contribution lies in its ability to enable researchers to quickly and easily prototype and test (co-)scheduling algorithms for Message Passing Interface (MPI) applications on HPC systems.

Concerning design principles and supported features for modeling and evaluation, it is noted that ELiSE is built upon a set of design principles that prioritize focused assessment and ease of use. Crucially, it is not a full system simulator; instead, its core mission is to evaluate different scheduling policies, with a distinct emphasis on co-scheduling.

This design choice allows ELiSE to avoid the complexities of full system simulation, which often necessitate thorough configuration and extensive tuning (a cumbersome approach that SiS development had to sustain). By focusing on the behavior of co-scheduling scenarios and leveraging pre-computed or externally derived execution results (e.g., from real experiments, partial experiments, or other models), ELiSE facilitates the rapid development and testing of algorithms. For modeling HPC environments and workloads, ELiSE requires several inputs:

- *HPC Cluster Description*: Users provide a simple description of the target HPC cluster, specifying the number of nodes, CPUs per node, and cores per CPU. This flexibility allows for the modeling of diverse system architectures.

- *Dynamic HPC Job Load*: A dynamic load of HPC jobs is provided as input. This involves defining the desired number of applications, an "application seed" (a set of applications with their compact allocation execution times), and a co-execution matrix (heatmap) detailing pairwise speedups between applications.

- *Workload Generation*: ELiSE supports the generation of various types of static or dynamic workloads. Applications can be randomly selected from the seed, with user-defined frequencies, or via a specific list. Furthermore, the arrival time intervals of applications can be defined as constant, random, Poisson, or Weibull, enabling realistic workload modeling.

- *Co-execution Heatmap*: A heatmap of pairwise speedups, which quantifies the performance change when two specific applications are co-located, is a fundamental input. This data is crucial for ELiSE's ability to model the effects of co-scheduling.

A standout feature of ELiSE for scheduler modeling and evaluation is its robust support for both in-built and custom-made (co-)scheduling algorithms. The framework provides a clear and straightforward mechanism for extending to new schedulers through a class hierarchy. An abstract Scheduler class defines essential operations such as host allocation conditions, job allocation, co-location, deployment, and reordering the waiting queue. The abstract deploy and backfill methods must be implemented for any new scheduling algorithm to be usable within ELiSE, providing a standardized interface for algorithm development. Current implemented schedulers include FCFS, FCFS with EASY backfill, FCFS with conservative Backfill, an FCFS co-scheduler with EASY Backfill, and a "smart" co-scheduler called Filler.

This extensibility is vital for researchers to test novel algorithmic designs. ELiSE's adaptation for co-scheduling is a core component of its utility. It dynamically adjusts the remaining execution time of jobs based on their co-located neighbors.

The framework accounts for scenarios where an application's neighbors might change during its execution or where it might have different neighbors across multiple allocated nodes. ELiSE operates on the reasonable assumption that the execution time of an application co-located with multiple neighbors is determined by the slowest case among its pair-wise interactions, calculated from the provided heatmap. This ensures that the simulation accurately reflects the performance impact of dynamic co-locations. The remaining execution time is recalculated at each simulation step using a formula that leverages the minimum pairwise speedup from the heatmap, ensuring responsiveness to changing co-location scenarios. While currently focused on pairwise co-locations, ELiSE includes mechanisms to support more advanced multi-job co-locations in future iterations.

For evaluating scheduling outcomes, ELiSE provides a rich palette of output and visualization tools. It generates detailed logs of workload execution, including job arrival, start, and end times. Crucially for analysis, it offers various visualizations, such as Gantt charts, system throughput plots, queue size dynamics, system utilization over time, and application speedups presented in boxplots, a subset of which is presented in Fig. 3.3.

These visual aids are indispensable for researchers to understand performance metrics and assess the behavior of different scheduling strategies. The evaluation metrics considered by ELiSE include makespan (total time to complete all jobs), job turnaround time (submission to completion), job speedup (ratio of compact execution time to co-scheduled execution time), and system utilization (percentage of system in use until the waiting queue is empty).

On validation and initial observations on co-scheduling behavior ELiSE has been rigorously validated to ensure its simulation results are closely aligned with real-world performance. Comparisons against an extended OAR RJMS on a small Grid5000-Grvingt cluster demonstrated minimal deviations for both simple FCFS scheduling and co-scheduling scenarios. Through preliminary experiments using ELiSE, important initial observations have been made regarding co-scheduling behavior, guiding future algorithmic design, based on:

- *Impact of Process Count Diversity*: ELiSE revealed that the diversity in the process count of jobs significantly impacts the benefits of co-scheduling. This highlights the need for more sophisticated co-scheduling strategies that can effectively manage diverse job sizes. The "Filler Co-Scheduler" was implemented in ELiSE to crudely address this by filling unutilized resources, demonstrating ELiSE's capability to test mitigating strategies.

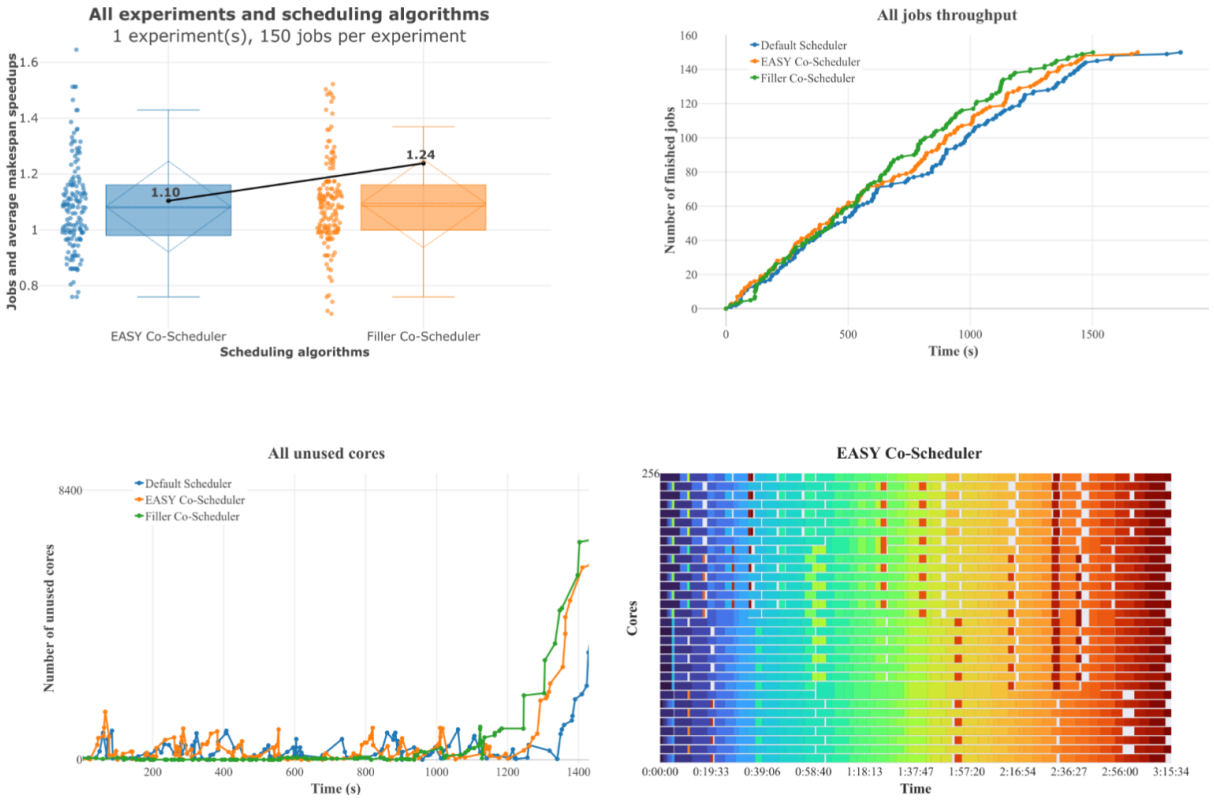


Figure 3.3: ELiSE ships with a set of built-in visualization features to help determine various scheduling factors[31]

- *Correlation between Mean Job Speedup and Makespan Improvement*: Experiments showed a strong correlation between the mean pairwise speedup of jobs in an application pool and the makespan improvement achieved through co-scheduling.
- *Correlation between Mean Job Speedup and Makespan Improvement*: This indicates that leveraging applications with high co-execution speedups is a critical factor for maximizing system throughput.
- *Trade-offs in Optimization Strategies*: ELiSE illustrated the inherent trade-off between system throughput (makespan improvement) and user satisfaction (percentage of jobs experiencing slowdowns relative to compact execution). The variability observed across different job shuffles, even for the same application pool, underscores the complexity and the necessity for advanced co-scheduling algorithms that can balance these competing metrics.

These insights are invaluable for informing the design of algorithms that aim for both high system performance and equitable user experience.

ELiSE demonstrates practical simulation performance, capable of simulating small and medium-scale scenarios very quickly, and even very large cases within tolerable times. For instance, it can simulate 10,000 jobs on a system with 2.5 million cores for 14 hours of makespan in approximately one hour of simulation time on a desktop machine.

This efficient performance ensures its applicability for extensive research and evaluation.

In conclusion, ELiSE provides a vital framework for addressing the challenges of co-scheduling in modern HPC systems. By offering a simple yet accurate platform for the rapid development, testing, and comprehensive evaluation of scheduling algorithms, ELiSE fills a

critical void in research tools. Its ability to model diverse system architectures and workloads, adapt to dynamic co-location scenarios, and provide rich visualization of performance metrics makes it an indispensable asset for understanding and optimizing HPC resource management.

The initial observations derived from ELiSE already highlight key considerations for algorithmic design, emphasizing its role in guiding the ultimate implementation and deployment of co-scheduling in production environments.

The aim of the tool developed in the scope of the present dissertation, SiS, is to build upon ELiSE's legacy and validate the results provided by ELiSE or other simulation tools. SiS will make available to the HPC community a tool to further their research on working production systems, enriching their insights on (co-)scheduling and optimal resource allocation.

Furthermore, ELiSE's preliminary studies on the behavior of workloads under co-location scenarios has been a source of inspiration of the development and experimentation with quarter-socket allocations of workloads on HPC systems.

Chapter 4

Design and Architecture of Slurm-in-Slurm (SiS)

4.1 ARIS

ARIS is the name of the Greek supercomputer, deployed and operated by GRNET S.A. (National Infrastructures for Research and Technology S.A.) in Athens. The system was ranked 468 in the Top500 list on June 2015, although, with 2025 data it is considered to be a legacy system with more than 10 years of contribution on scientific and research projects throughout all disciplines.

SiS development environment was the ARIS HPC system. SiS was developed while ARIS was in production and all experiments were conducted while the system was live, without interfering with its normal operation.

The following chapter has been based on information from the ARIS documentation page[37], as well as system information gathered by the author during the development of the SiS tool.

The ARIS system general information can be observed in the following Table 7.1:

Table 4.1: ARIS' System Specifications

Architecture	x86-64
Operating System	Redhat/Centos 6.7
Interconnect	
Technology	Infiniband FDR
Topology	Fat tree
Bandwidth [Gb/s]	56
Storage	
Type	IBM GPFS
Size [PByte]	2
Bandwidth [GB/s]	6
System Software	
Batch system	SLURM
System Management	xCat IBM
Monitoring	Nagios, Ganglia

Five hundred and thirty three (533) computational nodes in five (5) logical entities form ARIS. An overview is show in the following Table 4.2:

Table 4.2: ARIS' node Specifications

Node Type	Count	Accelerator	Memory	Cores
THIN nodes	426	not included	64 GB	20@2.8 GHz (two sockets)
GPU nodes	44	dual tesla k40m	64 GB	20@2.6 GHz + 2 x K40
PHI nodes	18	dual xeon phi 7120p	64 GB	20@2.6 GHz + 2 x 7120p
FAT nodes	44	not included	512 GB	40@2.4 GHz (four sockets)
ML node	1	8 volta v100	512 GB	40@2.2 GHz (two sockets)

All the nodes are connected via Infiniband network organized in a fat tree topology as in Fig. 7.5.

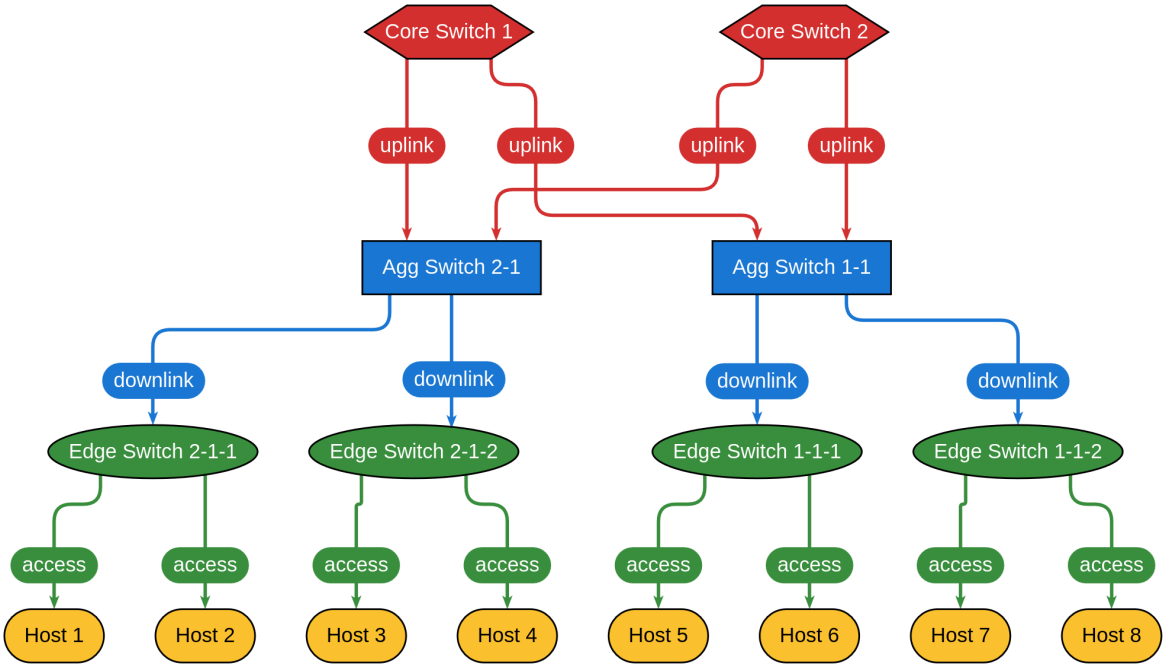


Figure 4.1: Abstract Fat Tree topology with 8 nodes. ARIS nodes follow the fat tree layout.

The nodes share 2 petabytes GPFS storage and access to the system is provided by two login nodes using SSH protocol connection.

A technical explanation of each node type is given in the next sections.

4.1.1 THIN Nodes

The 426 thin compute nodes (thin node island) deliver a theoretical peak performance (Rpeak) of 190.85 TFlops and a sustained Linpack benchmark performance (Rmax) of 179.73 TFlops. The thin node island is particularly well-suited for highly scalable applications that employ MPI or hybrid MPI/OpenMP programming models.

In Slurm terms, the system administrators of ARIS have organised the THIN nodes in the *compute* partition.

Table 4.3: THIN nodes technical information

Architecture	x86-64
System	IBM NeXtScale nx360 M4
Total number of nodes	426
Total number of cores	8520
Total amount of RAM [TByte]	27
Total Linpack Performance [TFlop/s]	180
Components	
Processor Type	Ivy Bridge - Intel Xeon E5-2680v2
Nominal Frequency [GHz]	2.8
Processors per Node	2
Cores per Processor	10
Cores per Node	20
Hyperthreading	OFF
Memory	
Memory per Node [GByte]	64

SiS was developed by leveraging the compute partition.

4.2 Overall SiS Architecture

As noted previously, SiS aspires to be a lightweight tool for testing functionalities on production Slurm systems. To that end, SiS is deployed through the common job submission mechanism on Slurm as a normal job, without requiring the user to have elevated rights, apart from being able to submit a job on the cluster. To get a grasp of the operation of SiS, Fig. 7.6 provides a useful visual aid:

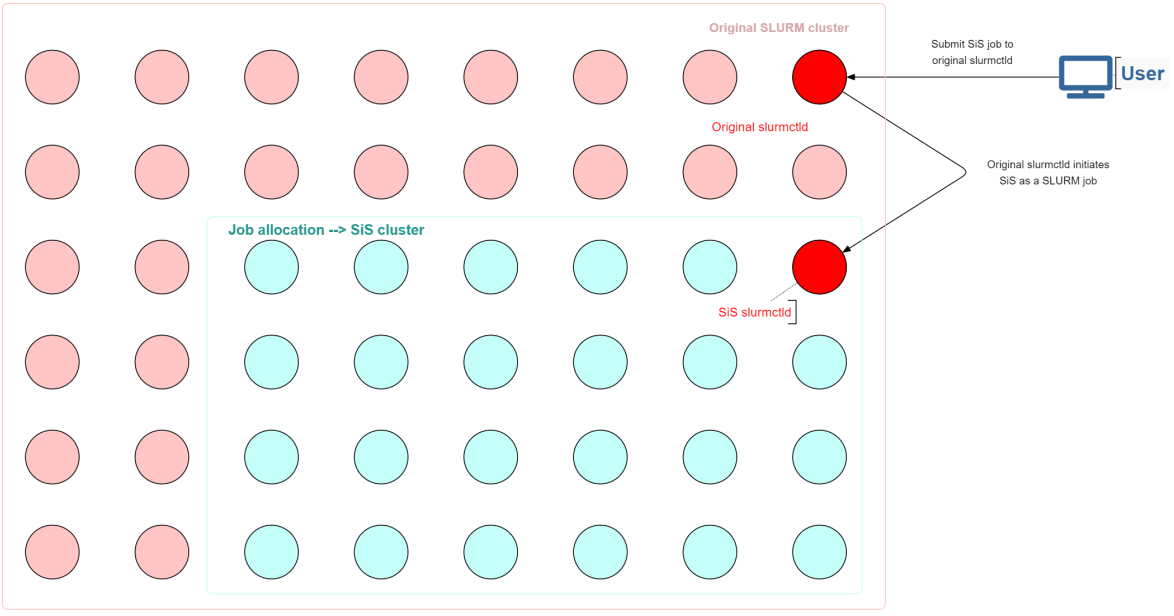


Figure 4.2: A simplified view of the workings of SiS within a Slurm cluster

A typical execution lifecycle of SiS involves the following steps:

1. The user submits the job to the external Slurm through the `sbatch` utility. The job script is the actual code wrapped with the SiS code to deploy and initiate the SiS cluster.
2. The job gets submitted to the underlying system's Slurm `slurmctld` which, among other operations allocates the `nodelist` for the specific job.
3. The allocation for the job constitutes the SiS cluster. A `slurmctld` acting as the controller of the SiS cluster initiates operation.
4. The SiS `slurmctld` establishes communication with the SiS `slurmd`s and delegates the execution to them.
5. When the submitted script finishes execution normally, or by means of the user's commands or by having exhausted the allocated time for the job completion, it exits gracefully terminating the SiS daemons as well.

With respect to the final observation concerning the lifecycle of the SiS framework, it becomes evident that the system is capable of executing the complete set of Slurm commands, provided that the relevant plugin or functionality has been properly integrated during the installation phase. In practical terms, this implies that SiS does not impose any restrictions on the standard usage of Slurm commands. Instead, it allows them to be invoked seamlessly through the SiS executables once the system is operational. Such invocations may occur either within an `SBATCH` script, thereby enabling batch job submission and management, or directly via the system's command-line interface (CLI). This characteristic highlights the flexibility and compatibility of SiS with the broader Slurm ecosystem, as it ensures that users can continue to rely on familiar job submission and management practices, while benefiting from the additional abstractions and capabilities introduced by SiS.

Since SiS is executed as a job within the Slurm workload manager, it is appropriate to examine the internal mechanisms governing the execution of Slurm jobs.

A *job* in Slurm represents a unit of work that aims to complete one or more *tasks*. Tasks may be specified at the job level using the options `--ntasks` or `--ntasks-per-node`, or alternatively at the step level through the `--ntasks` option. The number of CPUs allocated per task is controlled via the `--cpus-per-task` parameter.

Each task is executed on a single compute node within the context of a *job step*. A job is divided in job steps, starting the enumeration from job step 0. Each such step corresponds to an `srun` command. A task utilizes one or more CPU cores and requires a non-zero amount of memory. By default, Slurm assigns 5 Gigabytes of memory if no explicit request is made, although users may request smaller allocations. Notably, requesting less memory may increase the likelihood of a job being scheduled, as fewer resources are required.

When multiple tasks are defined within a job, each task consumes a subset of the resources allocated to the overall job. These tasks may be distributed across a single compute node or multiple nodes. However, the resources consumed by an individual task cannot exceed the resources available on a single node.

Once jobs are submitted and accepted by Slurm, they are assigned a unique *job identification number* (`jobid` or `job_id` for backwards compatibility). This identifier serves as the primary reference when interacting with Slurm utilities such as `scontrol` or when parsing output generated by programs such as `sacct`. Depending on the context, different forms of identification numbers may be displayed or required.

In practice, documentation and correspondence often refer simply to the *job* or *jobid*, with the implicit expectation that the user will infer the appropriate identifier to provide in a given situation.

The structure of Slurm job identifiers follows specific conventions. Underscores (_) are used to delimit elements of a job array from the associated job identifier, while periods .) separate the job identifier from a step identifier.

On average, a job will consist of two or more *steps*, each representing a distinct unit of execution within the job. The most common types of steps are outlined below:

- **External step** — This step represents the connection between the submission node and the leading node in the allocated node list. It typically succeeds regardless of whether the overall job completes successfully, failure to do so is a strong indication that a problem with the configuration or operation of the resource manager is has occured.
- **Batch step** — Created when jobs are submitted with `sbatch`. The exit code of the batch script directly impacts the final `STATE` of this step.
- **Interactive step** — Created when jobs are launched with `srun` outside of a batch job context.
- **Normal step** — A batch job can include multiple normal steps, which appear in the accounting output as `<jobid>.<step_id>`. Interactive jobs, in particular, do not contain normal steps.

When examining accounting information using commands such as `sacct`, one will observe that each job is associated with multiple entries, including an entry for the overall job itself. Each entry is reported with both a `STATE` code (e.g., `COMPLETED`, `FAILED`, `CANCELLED`) and an `EXITCODE` (e.g., `0:0`). It is therefore important to distinguish whether the reported state information refers to the overall job or to one of its component steps.

Having established the fundamental concepts of Slurm jobs, it is important to emphasize that the SiS framework is executed as a *batch job*. As previously discussed, the primary design objective of SiS is to function as a lightweight user-space tool that does not require elevated privileges for deployment or execution. By leveraging Slurm's batch job mechanism, SiS integrates seamlessly into the job scheduling environment while incurring only minimal overhead. This design choice ensures that the tool remains both efficient and non-intrusive, thereby aligning with the broader principle of scalability in high-performance computing systems.

The execution of SiS as a batch job within Slurm provides several important advantages that enhance both its usability and efficiency in high-performance computing environments. These advantages extend from resource allocation to system-level integration, ensuring that the tool remains lightweight while fully compatible with established scheduling infrastructures.

A first trait concerns the ability to monitor execution transparently through the mechanisms already available in the external Slurm for handling batch steps. Since every batch job is internally decomposed into steps, SiS health during job execution can be observed throughout its lifecycle using standard commands such as `sacct`, if an accounting plugin has been installed along SiS, and `scontrol`. For most use cases, the optional `slurmdbd` daemons and `accounting managers` should not be needed since SiS aims to become a lightweight testing tool for several Slurm components. This can aid in minimising the memory space SiS occupies and optimise execution latency. The external Slurm commands expose detailed information on job states, transitions, and exit codes. Thus, users can trace the execution of SiS without requiring

additional monitoring infrastructure or privileged system access, let alone re-installing these components alongside SiS.

Another significant characteristic of SiS is its reliance on the the prior allocation of compute nodes. Slurm requires that resources be requested before a job begins, which ensures deterministic allocation of both nodes and cores. SiS capitalises on this property, allowing users to predict resource availability and performance with a high degree of confidence. This capability is especially relevant for workloads where consistency and reproducibility are central requirements.

A further advantage arises from the flexibility of batch scripts, which constitute the deployment interface of SiS. Since batch scripts can embed arbitrary user-defined commands, SiS may be combined seamlessly with user code, enabling heterogeneous workloads to execute under the same allocation. In practice, this means that a single job allocation can be used to perform multiple distinct computational tasks (In simple terms: one external `sbatch` can amount to multiple SiS `sbatches`), thereby reducing queue waiting times and optimizing overall throughput. This flexibility is particularly beneficial in workflows that involve both monitoring and computation, as it reduces the fragmentation of resource usage.

Finally, the integration of SiS allows it to leverage the system-level guarantees already implemented by the external resource manager, since Slurm provides well-established mechanisms for job integrity, node-level networking, and security enforcement. By operating within this framework, SiS avoids the redundancy of re-implementing such mechanisms while benefiting from their robustness. This design choice enhances both the trustworthiness and the efficiency of the tool, ensuring that its execution is both secure and consistent with broader system policies.

To conclude with, an important feature of the Slurm in Slurm (SiS) framework is its ability to deploy custom Slurm installations where the source code has been explicitly modified or extended. Until now, the dominant approach for experimenting with new scheduling strategies, authentication methods, or accounting mechanisms was through the plugin system, which relies on well-defined application programming interfaces (APIs). While plugins have proven invaluable for extending Slurm's modular architecture, they inherently impose a boundary: developers are restricted to modifying only those functionalities that are exposed through the plugin interface. In contrast, SiS eliminates this limitation by allowing researchers and practitioners to install, configure, and execute Slurm instances whose source code has been directly altered. As a result, SiS greatly expands the scope of experimental work, enabling the evaluation of changes that reach beyond the existing API boundaries.

The significance of this capability is twofold. First of all, it opens the possibility for testing experimental schedulers, resource allocation mechanisms, or security enhancements that require fundamental changes to Slurm's core components, which previously could not be implemented without tampering with a production system. Secondly, it provides a controlled and isolated execution environment where different versions of Slurm, including those with source-level modifications, can coexist and be systematically benchmarked against each other. This is a crucial advancement in the research and development lifecycle, since it bridges the gap between theoretical design of scheduling algorithms or system functionalities and their practical validation within a functioning cluster environment.

Consequently, SiS not only preserves the extensibility advantages of the traditional plugin system, but also extends them by incorporating a higher degree of flexibility and experimental freedom. By supporting deployments of custom Slurm instances compiled from modified source code, SiS transforms into a powerful research and development tool. It enables testing scenarios that previously remained inaccessible, thus fostering innovation in areas such as

job scheduling, resource selection, fault tolerance, and system scalability. This makes SiS particularly valuable for both academic research and industry settings, where the ability to experiment with and validate modifications at the source code level can lead to shorter implementation times for improvements in high-performance computing infrastructures.

Chapter 5

Implementation and Integration

5.1 ARIS version

In this section, we will examine in detail the various refinements, design choices, and sensitive implementation points that proved critical in the development of SiS. Particular emphasis will be placed on the identification of the system's critical path, highlighting the components whose proper configuration directly determines the stability and performance of the framework. Furthermore, the section will provide practical guidelines and step-by-step instructions for deploying SiS within an already established supercomputing environment. These deployment instructions are intended not only to demonstrate the feasibility of integrating SiS into existing infrastructures but also to serve as a reproducible methodology for other high-performance computing systems. In doing so, the section aims to bridge the conceptual design of SiS with its practical application, ensuring that both researchers and system administrators can benefit from a comprehensive understanding of the framework's operational requirements and integration process.

At this stage, it is natural to raise a critical question: if the primary objective of SiS is to enable the deployment of any version of Slurm independently of the underlying system, why did the developers of SiS elect to utilize the same version as that installed on the host system, namely version 16.05?

To answer this question, one must draw his/her attention back to Chapter 4.1 where the issue of the ARIS HPC system infrastructure was first discussed. In this section it was explained that ARIS' underlying operating system was `RedHat/CentOS 6.7`, which is a rather old and unsupported version of `CentOS` that reached its end-of-life on 30th of November 2020 [38], followed by an overall discontinuation of the CentOS operating system. To the surprise of the community and enterprises using RedHat, the company developing CentOS, did not provide a clear path to system upgrade. To conclude this detour, the way this is related to ARIS HPC system and the realisation of SiS is that as explained Slurm is based on `Linux` distributions and newer versions of Slurm leverage the capabilities of the updated Linux kernel. In that sense, even though it was possible to install SiS with newer versions of Slurm, the core executables -meaning those that were responsible for deploying and running the `slurmctld` and `slurmd` daemons- made use of system calls only found in more recent versions of the Linux kernel, thus rendering attempts to operate SiS with more recent releases of Slurm inoperable. Consequently, the deployment of SiS within the ARIS supercomputing infrastructure is constrained to the same version of Slurm that is natively supported by the system, namely version 16.05.11. In other words, SiS does not independently introduce a more recent Slurm stack, at least for the case of the ARIS environment, but rather conforms to the specific software environment

already established in ARIS, thereby ensuring tested consistency and compatibility.

5.1.1 Availability and Reproducibility of Code

The source code of SiS has been made publicly accessible through a **dedicated GitHub repository**, thereby ensuring reproducibility, and ease of access within the research community. Any HPC enthusiast, researcher or just curious reader is encouraged to try the repository which contains very detailed instructions on how to install and run SiS on an HPC environment. By hosting the project on a widely used version control platform, the software remains readily available for inspection, modification, and extension by interested researchers and practitioners. This accessibility facilitates both independent verification of the results presented in this thesis and the potential for collaborative development and refinement of the framework. The online repository includes brief instructions on the configuration, installation and execution of SiS along with code and data for reproducing the quarter-socket MPI experiments conducted in the scope of this thesis.

In addition to its online availability, the full implementation of SiS has been archived in Appendix A.

5.1.2 System Requirements and Dependencies

SiS has been specifically designed to operate within HPC environments that are managed by the Slurm workload manager. The version of Slurm that can be deployed through SiS is primarily constrained by the kernel version of the underlying operating system as the introductory note of this chapter indicated. This dependency arises from the low-level interactions between Slurm and the kernel, which affect compatibility with certain system calls. Apart from this restriction, SiS remains agnostic with respect to the specific Slurm release, and thus, in principle, it can operate with a broad range of versions without further modification.

Another important consideration pertains to external library dependencies. In line with best practices in HPC environments, SiS does not impose additional requirements for software beyond what is already available in the cluster infrastructure. Instead, it relies on existing system-provided libraries and modules to resolve any dependencies that arise during compilation or execution. This approach minimizes administrative overhead, ensures consistency with the cluster's software ecosystem, but, in some cases, may need thorough monitoring of possible conflicts between the system's underlying Slurm installation and SiS. Moreover, by delegating dependency resolution to the modules already curated and maintained by the HPC facility, SiS enhances its portability and ease of deployment across diverse environments. In this respect, the software can be integrated into production systems with minimal disruption, while still preserving the flexibility needed for research and evaluation.

Finally, it is worth noting that minimal restrictions on Slurm versions and reliance on pre-installed libraries reinforces the goal of making SiS both lightweight and easily reproducible. The absence of heavy external requirements aims to make SiS suitable for rapid deployment in both testbed and production environments.

For the time being, SiS has been configured and tested in `RedHat/CentOS 6.7` and kernel version `2.6.32-754.35.1.el6.x86_64`.

5.1.3 Installation Process

This section describes the steps to set up the environment and install SiS, either having downloaded it from the online repository, or copying the code from the Appendix of this thesis.

The file hierarchy that a SiS user will come across out of the box when downloading the code from the online repository, as of October 2025, is depicted in the following Fig. 7.7:

```
.
├── bash
│   ├── aris_boilerplate.sh
│   ├── environment_variables.sh
│   ├── helpers.sh
│   ├── job_queue.txt
│   ├── epilogue.sh
│   ├── prologue.sh
│   ├── v-slurm-lite.sh
│   └── unit_tests
└── quarter-socket-mpi
    ├── data
    ├── find_max_min.sh
    ├── maxnodes.sh
    ├── run_compact.sh
    ├── run_cos.NAS_NAS.sh
    ├── submit_cmp.sh
    └── submit_co.sh
```

Figure 5.1: File hierarchy of the SiS project.

The main directory containing the code for SiS execution is `bash` and the driver script for installation is `aris_boilerplate.sh`, which contains the basic operations to -by default- download the Slurm source code and create the executables on the target system.

Before dwelling further to the several aspects of the `aris_boilerplate.sh` script, it is of outmost importance to point touch on the significance of the `environment_variables.sh` configuration file.

This bash script contains the necessary paths and variables that will be needed by the other scripts to properly install and configure SiS. An example file has the contents present in Lis. 5.1:

Listing 5.1: An example layout of the environment variables and paths used by SiS

```
# Pick version of Slurm to install, currently version 16 is
   ↪ tested and operating
# but version 25 and 21 can also be installed.
version="16.05.11.1"

########## EDIT THESE PARAMETERS ##########
# Set certificate names
cert_name=<your-certificate-name>
key_name=<your-private-key-name>
pkey_name=<your-public-key-name>
password=<your-private-key-file-password>
# Pick your home directory
base_dir="$HOME/$USER"
home_dir="$HOME/$USER/slurm-install/$version"
# Set job queue path
job_queue_filepath=${base_dir%/}/job_queue.txt
# Based on underlying system
sys_cpus=<cpus-per-socket>
sockets=<based-on-system-specs>
cores_per_socket=<based-on-system-specs>
threads_per_core=<based-on-system-specs>
real_mem=<memory-allocated-to-node>
partition_name=<same-name-as-underlying-system>
max_mem_per_node=<node-available-memory>
#Execution variables
errpath=<SiS-deployment-error-output-path>
outpath=<SiS-deployment-standard-output-path>
nodes_count=<desired-node-count>
time=<walltime-total-time-to-end>
mem_per_cpu=<memory-allocated-per-node>
# Add any other SLURM variables here
###########################################
```

The structure of the installation script is divided into defined sections, each of which specifies parameters necessary for the correct deployment and execution of SiS. The first section records the target version of Slurm, which determines the release of the source code that will be downloaded and compiled during the installation procedure. As of October 2025, only version 16.05.11 of Slurm is fully operational, although more versions are available, targeting more contemporary systems. The second section establishes the file system paths that will serve as the root directory and also storage locations for complementary installation files. This section contains explicit definitions of the paths to the executable binaries, the configuration files, and the path to the file that contains the description of the job to be executed in each SiS deployment and iteration.

Subsequently, the script includes a section dedicated to the hardware specifications of the underlying system. These parameters capture system-specific characteristics that influence compilation and execution, and are therefore required for aligning the deployment process with the particular HPC environment. The final section of the script specifies the paths and parameters for the `sbatch` job of the external/underlying system Slurm will use to deploy SiS.

This ensures that the user will not have an overhead in editing multiple files.

Returning to the subject of the `aris_boilerplate.sh` script, this script automates the preparation, compilation and deployment of user-level Slurm instances. It sources environment and helper definitions, determines the target Slurm major release, and branches into installation or configuration modes depending on the command-line option (`--install`/`-i` or `--configure`/`-c`).

For each supported major release (currently available releases include version 16 for full operability and versions 21 and 25 for installation and debugging purposes) the script creates the requisite directory layout, manages module loading where required, generates SSL keys for authentication in inter-node communication. It also retrieves the Slurm source archive from the upstream repository, configures and compiles the tree, and finally stages or links the resulting plugin objects and runtime binaries into the user installation area.

In short, the script encapsulates the end-to-end build-and-deploy procedure for multiple Slurm releases within a non-privileged user directory.

The full code for the boilerplate script can be found in aris_boilerplate.sh.

5.1.4 SiS Deployment

The v-slurm-lite.sh script orchestrates the setup and execution of an ephemeral, user-level SLURM cluster within the existing batch allocation.

To assist the user in minimising the cumbersome task of having to edit multiple files, the auxiliary script of submit-v-slurm.sh is used to intermediate the allocation described in `environment_variables.sh` and the batch script of `v-slurm-lite.sh`.

At a high level, the script proceeds through a sequence of logical stages. First, it executes a `prologue.sh` script, which serves as a customisable initialisation hook for user-defined preparatory actions such as loading dependencies or configuring the execution environment. The script then sources a set of the environment as described in Lis.5.1 and helper functions in helpers.sh that define variables, paths, and utility functions necessary for subsequent configuration steps.

Following initialisation, the script determines the number of nodes allocated by the external batch scheduler and designates one as the control node. It then constructs a temporary SLURM configuration file (`slurm.conf`) tailored to the current execution context. This configuration defines the control machine (or `slurmcltd` host for newer versions), compute nodes, resource parameters, and partition information. The exact structure of this configuration varies depending on the major version of Slurm being used, ensuring compatibility across different Slurm releases.

Once configuration is complete, the script initiates the SiS controller (`slurmctld`) on the designated control node and the SiS (`slurmd`)s on the remaining nodes. These components are launched via `srun` commands, using parameters that specify node allocation, task count, and CPU binding. The controller and daemon processes collectively establish a fully functional, albeit temporary, Slurm cluster within the user's allocation.

After the cluster becomes operational, the script performs a brief validation by querying the node status using the `scontrol show nodes` command. It then enters an orchestration phase, during which it monitors the cluster queue and submits user jobs based on a predefined schedule specified in a job list file like the one on Lis.5.2:

Listing 5.2: The job list file that can host instructions to run as many applications, within SiS context, as the user has defined

```
# Insert the jobs for SiS here. Every job must be on a
  ↪ separate file
# using this format: <path/to/job/> <delay>, where path and
  ↪ delay
# are separated by a tab

# Example Usage
./mpi/vMpiJob.sh      5
```

The script ensures that all queued jobs are executed and periodically verifies their completion. If any residual jobs remain active, they are cancelled to guarantee a clean shutdown.

Upon completion of all job submissions, the script invokes an `epilogue.sh` script. This serves as a user-defined termination hook, typically used for cleanup operations, result collection, or post-processing tasks. The inclusion of separate prologue and epilogue scripts provides modularity and flexibility, allowing users to tailor pre- and post-execution behavior without modifying the main orchestration logic.

In summary, the `v-slurm-lite.sh` script functions as a dynamic orchestration layer and is the core of SiS. It automates environment setup, configuration generation, controller and daemon instantiation, job submission, and final cleanup. By combining user-defined prologue and epilogue stages with runtime configuration flexibility, it enables reproducible and isolated testing of distributed workloads in a self-contained SLURM environment.

Chapter 6

Evaluation, Conclusions, Future Work

6.1 Validation

The validation of SiS was conducted through a series of functional tests designed to verify its correct operation as a proper, fully operational Slurm workload manager and its interaction with the several tools and internal commands of SLURM. Given the current limitations of the SiS software, it was not feasible to execute full-scale MPI workloads and assess the performance of the overhead of the tool on such executions. Consequently, the validation process focused primarily on confirming that SiS correctly interprets, forwards, and executes standard Slurm commands within its configured environment.

During testing, a range of Slurm operations was issued directly to SiS, including commands for job submission, monitoring, and resource querying. In all cases, the tool exhibited the expected behavior, accurately reproducing the corresponding responses of a native Slurm installation. This consistency demonstrated that the internal mechanisms of SiS—such as configuration parsing, command translation, and interaction with the underlying Slurm daemons, function as intended.

In the next Listings, the output of several of the Slurm commands are shown to validate the proper function of SiS. The following Lis.7.1 presents the output of an `sinfo` command issued from SiS for an allocation of 3 nodes [truncated output for clarity]:

Listing 6.1: A SiS-invoked `sinfo` command with 3 nodes.

```
NODELIST   NODES PARTITION     STATE CPUS      S:C:T MEMORY
node066        1 compute*     mixed   20    2:10:1  57344
node413        1 compute*     mixed   20    2:10:1  57344
node414        1 compute*     mixed   20    2:10:1  57344
```

Likewise, the next `sinfo` command has been issued in order to demonstrate the capability of SiS to scale in many nodes, as shown in the output of Lis.7.2[truncated output for clarity]:

Listing 6.2: A SiS-invoked `sinfo` command with 10 nodes, demonstrating the ability of SiS to scale up to as many nodes as the system can provide

```
NODELIST    NODES PARTITION       STATE CPUS    S:C:T MEMORY
node323         1 compute*        mixed   20    2:10:1  57344
node324         1 compute*        mixed   20    2:10:1  57344
node325         1 compute*        mixed   20    2:10:1  57344
node326         1 compute*        mixed   20    2:10:1  57344
node361         1 compute*        mixed   20    2:10:1  57344
node362         1 compute*        mixed   20    2:10:1  57344
node363         1 compute*        mixed   20    2:10:1  57344
node364         1 compute*        mixed   20    2:10:1  57344
node365         1 compute*        mixed   20    2:10:1  57344
node366         1 compute*        mixed   20    2:10:1  57344
```

For the same execution, the ability to monitor the job queue is shown by invoking the `squeue` command from the SiS binaries:

Listing 6.3: A SiS-invoked `squeue` command while performing a job that has allocated 10 nodes.

```
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(
    ↪ REASON)
2     compute  Sinfo-10  goumas  R       0:01     10 node[323-326,
                                                      361-366]
```

Subsequently, a systematic validation procedure was conducted in which all Slurm commands were executed within the SiS environment to assess functional correctness and conformity with native Slurm behavior.

6.2 Experimental Results ¼-socket

Following the successful completion of the SiS framework in order to be able to execute MPI-based workloads, the next logical step in its development involves extending its functionality to support intelligent co-scheduling mechanisms. In this context, the design of a co-scheduling algorithm to be integrated within the SiS environment required a set of foundational experimental data to guide its formulation and subsequent evaluation.

To that end, a series of controlled experiments was conducted to examine the behavior of representative HPC workloads under co-located execution (or co-execution) conditions. Specifically, the experiments aimed to quantify the interaction between different types of computational loads when they share the same physical resources. These empirical observations serve as primary research data for assessing the viability of the proposed scheduling approach and for identifying potential optimization opportunities to be incorporated into future iterations of the SiS. The results presented, thus, constitute an essential preliminary step toward the integration of a dynamic and adaptive co-scheduling component within SiS.

For the sake of the benchmarking experiments regarding the co-execution of 4 NAS programs on the same compute nodes, each compute node in the ARIS environment consists of two sockets, with each socket providing ten physical cores, resulting in a total of twenty hardware threads available for computation. The co-execution algorithm was designed to leverage the node's pinout topology in order to assign each benchmark to a distinct subset of cores while maintaining a balanced distribution across both sockets. This approach was used

to evaluate the intra-application interference while still allowing inter-application sharing of system resources such as memory bandwidth and last-level cache.

The code for co-executing 4 different NAS benchmarks is given in Appendix B.

The mapping strategy applied during these experiments was as follows:

- **program A** was pinned to cores 0–2 and 10–11,
- **program B** to cores 3–4 and 12–14,
- **program C** to cores 5–7 and 15–16,
- and **program D** to cores 8–9 and 17–19.

Given that cores 0–9 correspond to the first socket and cores 10–19 to the second, this configuration ensures that each benchmark utilizes an equivalent number of cores from both sockets, thereby promoting symmetry in resource distribution. Such a placement policy allows for the evaluation of co-execution effects under a controlled and reproducible topology, where locality and resource contention can be systematically analyzed.

The NAS parallel benchmark version utilised was `NPB3.4.3 MPI`. All programs were executed for a 64 hardware threads and class D size problems[39].

The results of the observation were put against their compact execution, in the sense of comparing the execution speed of the program when executing by itself on the same amount of nodes as its co-executed counterpart occupied (e.g. for 64 threads, this equals 4 nodes). This measure, called *speedup* is measured by Eq.7.1:

$$\text{speedup} = \frac{\text{time}_{\text{compact}}}{\text{time}_{\text{quarter}}} \tag{6.1}$$

where,

$\text{time}_{\text{compact}}$, represents the average execution time of NAS benchmark on 64 threads of class D without co-execution,and $\text{time}_{\text{quarter}}$, represents the average execution time of NAS benchmark on 64 threads of class D co-executed with 3 other NAS benchmark programs on the same node.

Preliminary observations indicate that this configuration enables a fair assessment of performance interactions between co-executed workloads, providing insight into the impact of processor affinity and resource partitioning on overall throughput. The results obtained from these experiments form the basis for a deeper investigation into the performance trade-offs associated with node-level co-scheduling in high-performance computing systems.

6.2.1 EP - Embarassingly Parallel

The first benchmark considered is the **EP (Embarrassingly Parallel)** kernel, which represents applications that require minimal inter-process communication. The EP benchmark generates independent random numbers across multiple processes, effectively stressing the processor and memory subsystems while imposing negligible communication overhead. Due to its inherent lack of data dependencies, *EP* serves as a useful baseline for understanding the raw computational performance of co-executed workloads without interference from network contention or synchronization delays.

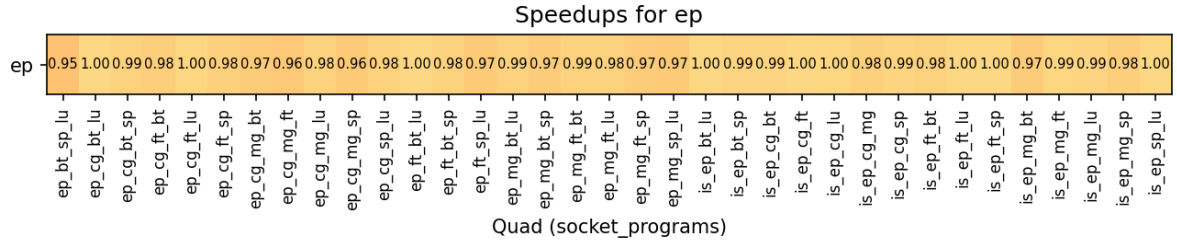


Figure 6.1: A heatmap representing the speedup of EP program when co-executed with other programs on a quarter socket environment for class D problems with 64 processes/threads

No major differences were spotted in the execution of the EP benchmark. This was expected since EP has no data dependencies and, thus, does not burden the intra-socket interference. The mean speedup for EP co-executed workloads is 0.98.

6.2.2 IS - Integer Sort

IS Integer Sort from the NAS benchmarks requires ranking an unsorted sequence of keys using bucket sort. The parallel version of IS divides up the keys among the processors. First, each processor counts its keys and writes the result in a private array of buckets. Then, the values in the private buckets are summed up. Finally, all processors read the sum and rank their keys.

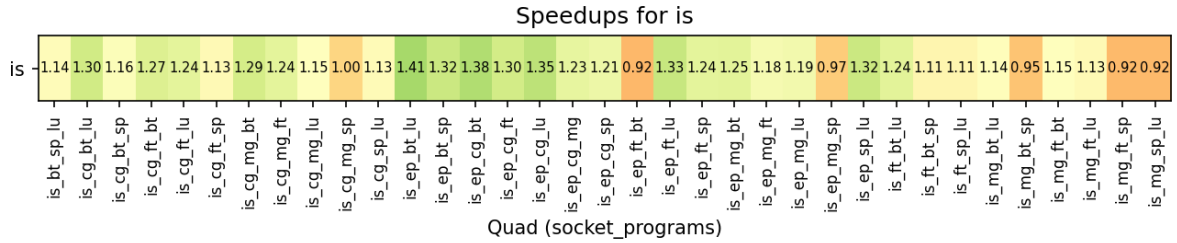


Figure 6.2: A heatmap representing the speedup of IS program when co-executed with other programs on a quarter socket environment for class D problems with 64 processes/threads

IS demonstrated significant speedup for most of the co-executed workloads, with a mean of 1.18 speedup.

6.2.3 BT - Block Tridiagonal Solver

BT (Block Tridiagonal Solver) is one of the three pseudoprograms of the NAS benchmarks suite. It is designed to solve non-linear partial differential equations using the block tridiagonal matrices algorithm. The benchmark is one of the most demanding of the NAS benchmark suite in terms of computation, but also allows for leveraging the computational capabilities of the underlying system[40].

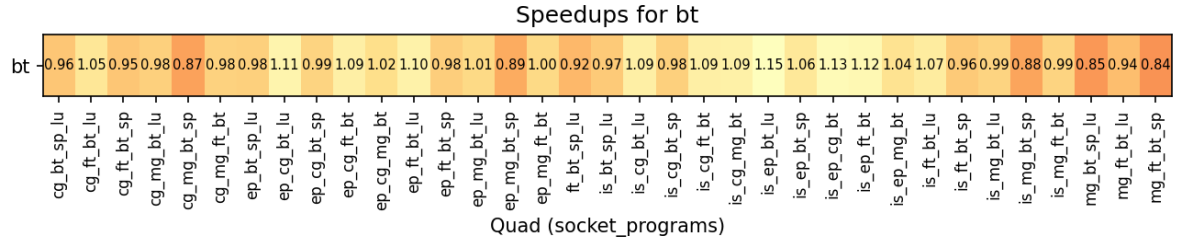


Figure 6.3: A heatmap representing the speedup of BT program when co-executed with other programs on a quarter socket environment for class D problems with 64 processes/threads

The results for BT speedup evaluation are not deemed conclusive since the program demonstrates speedups ranging from 0.84 to 1.15, with a mean of 1.00.

6.2.4 CG - Conjugate Gradient

The **CG - Conjugate Gradient** method is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. This kernel is typical of unstructured grid computations in that it tests irregular long distance communication, employing unstructured matrix vector multiplication[40].

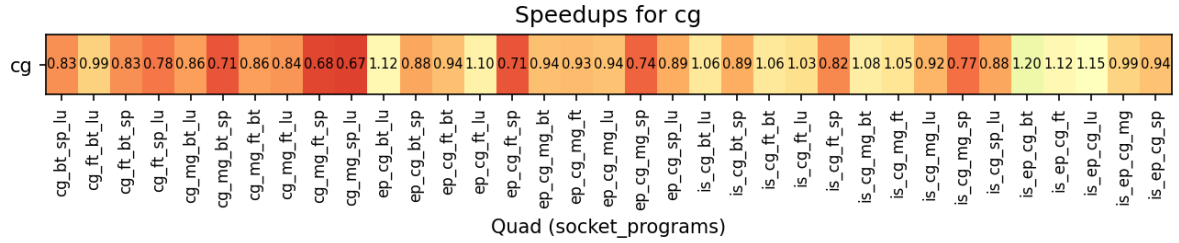


Figure 6.4: A heatmap representing the speedup of CG program when co-executed with other programs on a quarter socket environment for class D problems with 64 processes/threads

CG has exhibited a clear slowdown behaviour when co-executed with other workloads, also validated by the mean speedup of 0.91.

6.2.5 FT - Fast Fourier Transform

FT (Fast Fourier Transform) represents a 3-D partial differential equation solution using Fast Fourier Transforms (FFTs). This kernel performs the essence of many spectral codes. It is a rigorous test of long-distance communication performance[40].

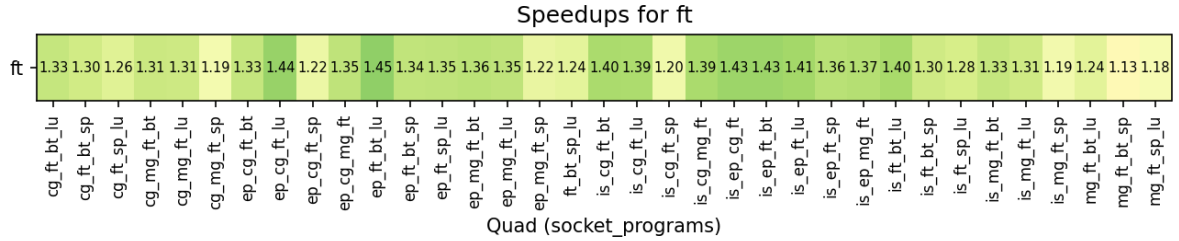


Figure 6.5: A heatmap representing the speedup of FT program when co-executed with other programs on a quarter socket environment for class D problems with 64 processes/threads

The mean speedup of 1.32 signals a major improvement on the execution time of the program when co-executing with other NAS benchmarks.

6.2.6 LU - Lower Upper symmetric Gauss-Seidel

LU (Lower Upper symmetric Gauss-Seidel performs a simulated solution of a three-dimensional, compressible Navier–Stokes equation using a regular block-structured grid[40]. It is another pseudoprogram available from the NAS benchmarks. The benchmark employs an approximate lower–upper (LU) factorization scheme to solve a system of linear equations resulting from the discretisation of the governing partial differential equations. Due to its iterative nature and frequent data dependencies, the LU benchmark is both computation- and communication-intensive, making it particularly suitable for evaluating the performance of message-passing implementations and the efficiency of interconnect networks in parallel computing environments.

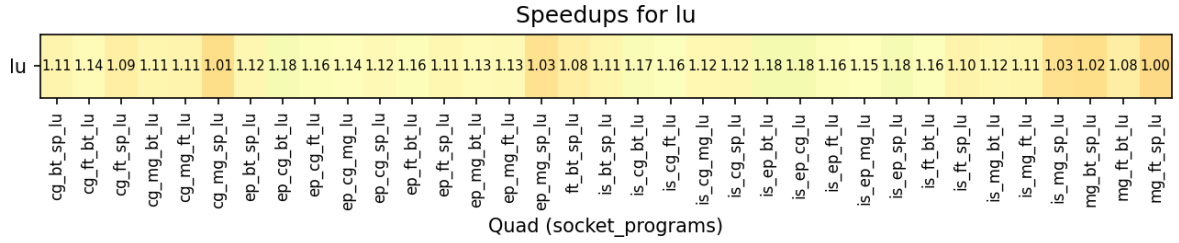


Figure 6.6: A heatmap representing the speedup of FT program when co-executed with other programs on a quarter socket environment for class D problems with 64 processes/threads

Indications of positive speedup are shown since the mean speedup of co-executing LU workloads is 1.12.

6.2.7 MG - MultiGrid

MG (MultiGrid) represents an approximate to the solution of a three-dimensional discrete Poisson equation using the V-cycle multigrid method. It requires highly structured long distance communication and tests both short (intra-node) and long distance (inter-node) data communication.

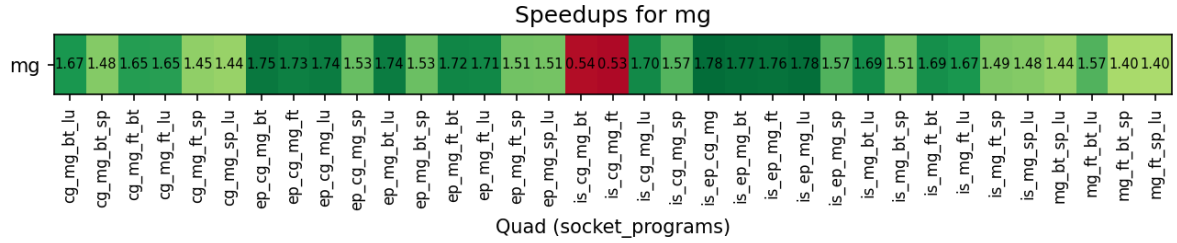


Figure 6.7: A heatmap representing the speedup of MG program when co-executing with other programs on a quarter socket environment for class D problems with 64 processes/threads

MG has manifested the greatest speedup among the tested NAS benchmarks with an average speedup of 1.55.

6.2.8 SP - Scalar Pentadiagonal

The last of the three pseudoprograms is the **SP (Scalar Pentadiagonal)** solver of partial differential equations for the compressible Navier-Stokes system of non-linear equations. The benchmark employs an algorithm based on scalar pentadiagonal systems, which arise from the discretization of implicit approximate factorization schemes. Each iteration involves solving a series of independent one-dimensional systems along each spatial direction.

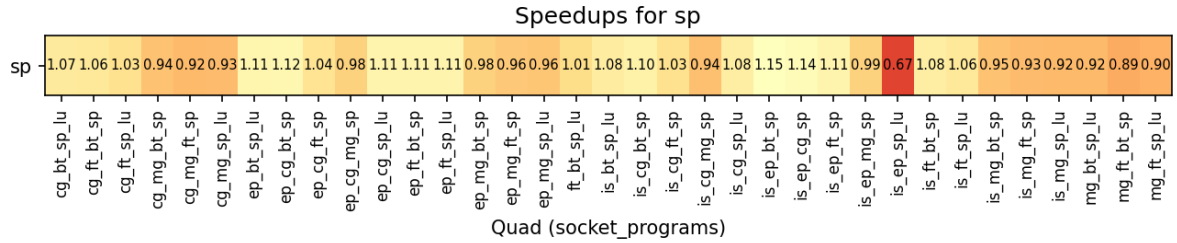


Figure 6.8: A heatmap representing the speedup of SP program when co-executed with other programs on a quarter socket environment for class D problems with 64 processes/threads

The SP pseudoprogram showed no actual mean improvements, since the average speedup is 1.01, when co-executed with other benchmarks, although, the actual speedup from one workload to the next can show significant differences.

6.2.9 Aggregate Metrics

To obtain a comprehensive understanding of the performance implications of benchmark co-execution, the speedup is also represented as a boxplot in Fig.7.16. A boxplot representation of the speedup values across all benchmark configurations provides an overview of the overall performance distribution, highlighting both median behavior and variability among different co-executed workloads. The box extends from the first quartile (Q1) to the third quartile (Q3) of the data, with a line at the median. The whiskers extend from the box to the farthest data point lying within 1.5x the inter-quartile range (IQR) from the box. Outliers are represented as 'x' 's that extend past the whiskers of each box.

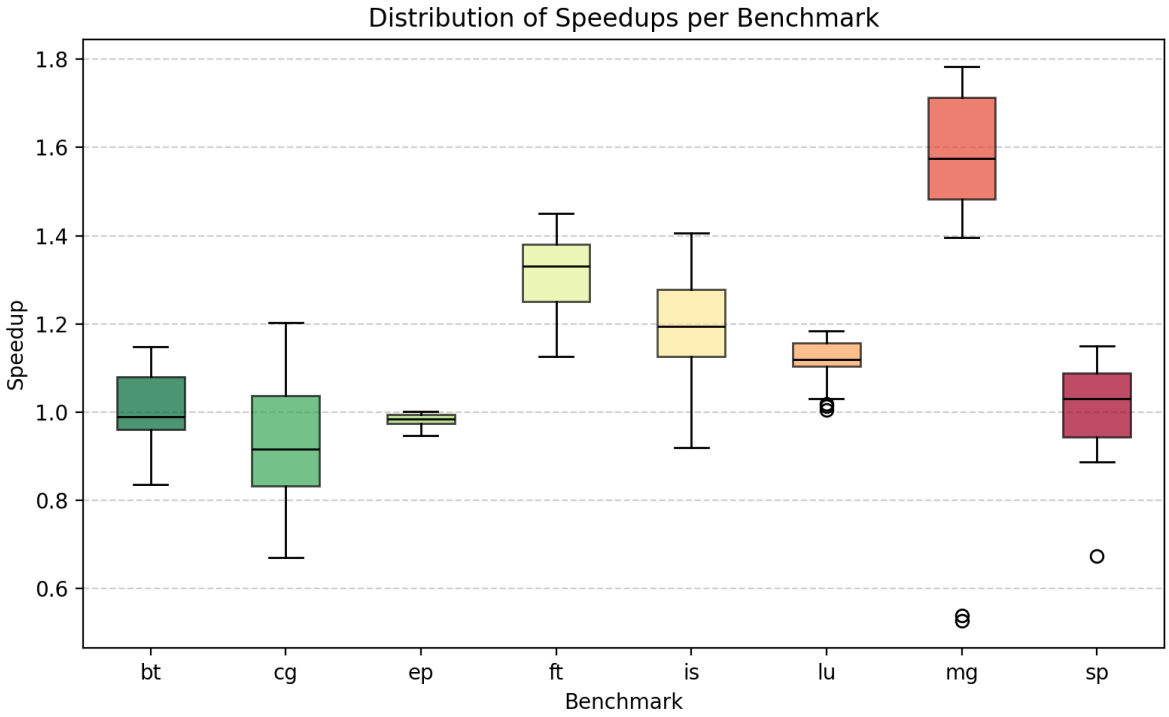


Figure 6.9: The boxplot of NAS benchmarks against their relative speedup when co-executed in a node and occupying a quarter of the nodes.

Overall, the results demonstrate substantial heterogeneity in how each benchmark responds to co-execution. Benchmarks such as `MG` and `FT` exhibit the highest median speedups, with several instances exceeding a factor of 1.6, indicating that under certain co-execution patterns, these applications benefit from improved resource utilization or reduced contention in shared system components. Conversely, `CG` and `SP` present greater variability and, in some cases, performance degradation, as reflected by speedup values below 1.0. This suggests a higher sensitivity of these workloads to shared hardware resources, particularly memory bandwidth and interconnect contention.

Benchmarks such as `EP` maintain a median speedup close to unity, implying that their performance remains largely unaffected by the presence of other workloads, consistent with their embarrassingly parallel nature. `LU` and `IS` display moderate yet consistent improvements, suggesting limited interference and balanced communication-to-computation ratios when co-executed with other benchmarks.

In summary, the observed distribution highlights that co-execution effects are highly benchmark-dependent. While some applications demonstrate potential gains from shared-node execution, others suffer measurable slowdowns.

To highlight the effects of quarter-socket execution compared to other execution policies that have been previously tested, the aggregate speedup results from the quarter-socket execution have been set against the results of half-socket executions for identical NAS benchmarks, in class and processing threads. The results are shown in Fig.7.17.

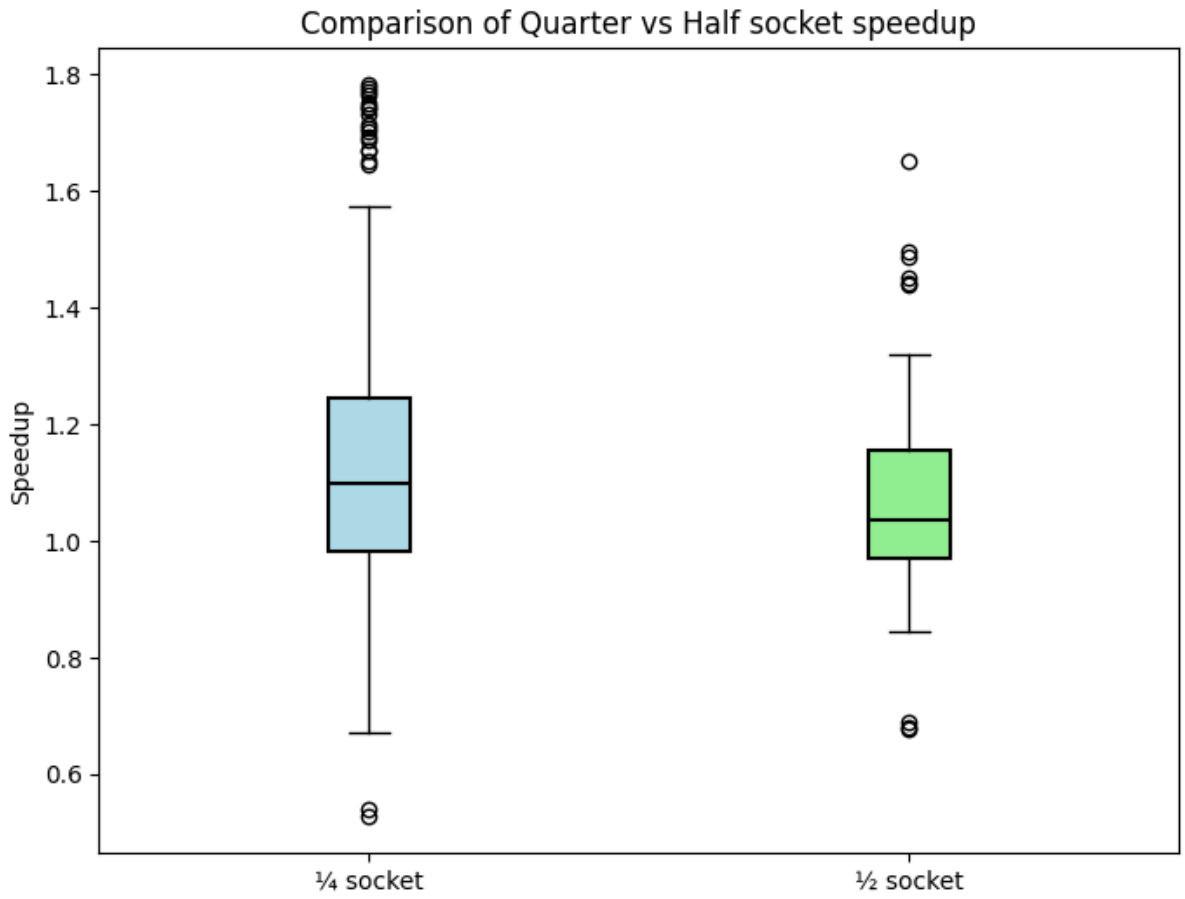


Figure 6.10: The boxplot of NAS benchmarks speedups under ¼-socket execution against the speedup distribution under ½-socket execution.

The median value for quarter-socket execution is 1.09 in comparison with 1.03 for half-socket execution.

The second metric, referred to as *sensitivity*, captures the absolute (meaning that it disregards if this influence results in a speedup or slowdown) magnitude of performance deviation, experienced by a benchmark as a result of its co-execution with another benchmark within a quarter socket workload. Sensitivity is calculated as the absolute difference between the isolated and co-executed performance, normalized to account for baseline execution time differences across all execution involving each benchmark.

$$S_{b,q} = \frac{1}{|\{b,q\} \in Q_b|} \sum_{\{b,q\} \in Q_b} \left| 1 - \text{speedup}_{b,q} \right| \tag{6.2}$$

where,

$S_{b,q}$ is the sensitivity of the benchmark b with regard to benchmark q,

Q_b is the unordered set of all quarter-socket executions involving the NAS benchmark b, and

$\text{speedup}_{b,q}$ is the measured speedup of the benchmark b in execution q.

An overall assessment of the sensitivity factor of the workloads is represented in the heatmap of Fig.7.18.

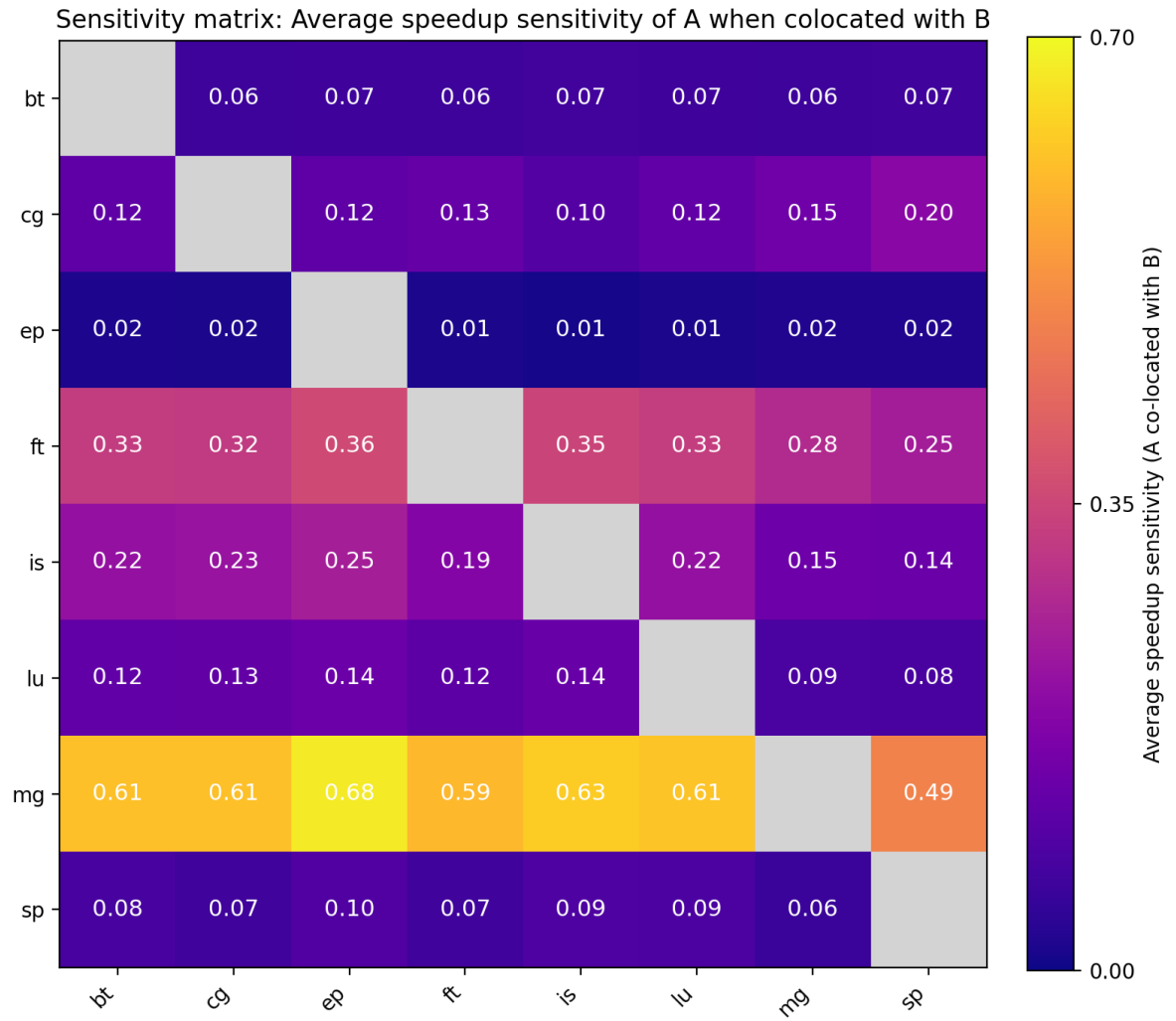


Figure 6.11: A heatmap representation of sensitivity of the absolute speedup magnitude when two NAS benchmarks are co-executed in a quarter-socket execution.

The results reveal significant variability in sensitivity across benchmark pairs. The MG benchmark exhibits the highest overall sensitivity, with values ranging from approximately 0.49 to 0.68 when co-executed with other workloads. This indicates that MG is particularly prone to performance fluctuations under shared resource conditions. Similarly, FT demonstrates substantial sensitivity levels ranging from 0.25 to 0.36.

In contrast, EP shows the lowest sensitivity values below 0.02 in most cases, confirming that it operates in an almost fully independent manner with negligible performance interference, an expected behavior for a random number generator. Benchmarks such as LU, IS, and BT occupy an intermediate position, showing limited but non-negligible sensitivity, typically below 0.2. This suggests that while these applications do experience some degree of interference under co-execution, the overall effect remains bounded and predictable.

Overall, the sensitivity matrix highlights the asymmetric nature of performance interference: while some benchmarks (e.g., MG, FT) act as "sensitive" workloads that are strongly

affected by co-execution, others (e.g., EP, SP) are relatively stable and may serve as suitable co-execution partners.

6.3 Conclusions

This dissertation has presented the design, implementation, and validation of the *SiS* (Slurm-in-Slurm) framework, a novel contribution to the field of high-performance computing resource management. The primary objective of this work was to develop a lightweight, user-space tool that can deploy and manage isolated Slurm instances within production HPC environments, thereby enabling future development to render SiS capable of hosting new and innovative scheduling policies, workload management strategies, and system-level algorithms without requiring privileged access or administrative intervention.

The results of this study have demonstrated that SiS successfully fulfills this objective. The framework operates entirely in user space, proving that complex job management and scheduling mechanisms can be instantiated and executed outside of the privileged system layer. This design allows HPC practitioners and researchers to explore, test, and validate novel scheduling strategies directly on production systems, bridging the gap between theoretical research and practical deployment. In this respect, SiS establishes a valuable experimental foundation for the HPC community, providing both flexibility and reproducibility.

Furthermore, SiS enables the deployment of customized or in-house compiled versions of the Slurm workload manager. This capability represents a significant advancement over existing testing paradigms, which typically rely on Slurm's plugin-based extensibility and the authorisation for deployment of such plugins on production systems. By allowing researchers to modify and recompile the Slurm source code itself, SiS expands the scope of experimentation beyond the limitations of plugin APIs. Although the framework is still under active development, its current implementation provides a stable and extensible basis for further enhancement and integration with modern resource management research.

Beyond the development of SiS itself, this dissertation has also contributed valuable experimental findings and methodologies for the analysis of workload co-execution. Using ARIS supercomputer, a series of controlled experiments were conducted to investigate the performance implications of co-executing different NAS Parallel Benchmarks on quarter-socket allocations. The results include a comprehensive set of performance metrics, such as speedup distributions and sensitivity analyses, which collectively shed light on the complex interactions between diverse workloads in shared-node environments. These insights can guide future research into performance-aware scheduling and resource allocation strategies that aim to maximize throughput while maintaining fairness and efficiency.

In conclusion, the SiS framework offers both a practical and conceptual advancement in the study of HPC scheduling systems. It provides an extensible platform for reproducible research, a mechanism for testing customized Slurm configurations, and a basis for future data-driven research to explore co-scheduling and resource sharing at fine granularity levels. The findings and tools developed through this work lay a strong foundation for subsequent studies in performance modeling, workload characterization, and adaptive scheduling in large-scale computing environments.

6.4 Future Work

While the SiS framework provides a functional and extensible foundation for experimental workload management in HPC environments, several directions for future development and research have emerged as a result of this work.

The immediate priority for future iterations of SiS is to enable the seamless execution of `MPI`-based workloads. Although the current implementation supports the deployment and management of Slurm instances and the submission of batch jobs, the integration of full `MPI` functionality will considerably enhance its applicability. Achieving this would allow researchers to test and evaluate distributed parallel applications within the isolated environment provided by SiS, further bridging the gap between controlled experimental setups and production-level workloads.

Another avenue for development involves extending the framework's compatibility with a broader range of Slurm versions. Presently, SiS deploys a specific version of Slurm that is closely tied to the underlying system kernel. Expanding this support to include newer and legacy versions would increase the framework's versatility and ensure its long-term usability across heterogeneous HPC platforms. This would also facilitate comparative studies of Slurm version-specific behaviors and performance characteristics.

Beyond its technical evolution, SiS offers a valuable platform for conducting research into scheduling strategies. One promising direction is the design and evaluation of novel Slurm schedulers that explicitly leverage workload co-execution strategies. Using the insights derived from the co-execution experiments presented in this dissertation, future work can explore algorithms that optimise job placement by considering shared resource contention, communication locality, and workload complementarity. Such schedulers could dynamically adjust job allocation policies to improve overall system utilization and reduce interference effects.

Furthermore, an important research direction lies in the systematic characterisation of workloads. By employing feature extraction techniques to identify key computational and memory access patterns, workloads could be classified into categories analogous to the NAS benchmark suite. This categorisation would enable predictive modeling of co-execution effects, allowing schedulers to determine in advance which workloads can safely and efficiently share computational resources. When combined with the experimental flexibility of SiS, such an approach could lead to data-driven co-scheduling methodologies that optimize performance while maintaining fairness across diverse applications.

Chapter 7

Εκτεταμένη Ελληνική Περίληψη

Εισαγωγή

Τα υπολογιστικά συστήματα υψηλών επιδόσεων (High Performance Computers - HPC) αποτελούν θεμέλιο λίθο για πληθώρα επιστημονικών και εμπορικών εφαρμογών. Η συνεχώς αυξανόμενη ενεργειακή τους κατανάλωση, σε συνδυασμό με τη διαφοροποίηση των φορτίων εργασίας, εντείνουν την ανάγκη για βελτιστοποίηση της αξιοποίησης των πόρων τους και τη διεύρυνση των λειτουργικών δυνατοτήτων τους. Καίριο ρόλο διαδραματίζει η ανάπτυξη ευέλικτων συστημάτων διαχείρισης πόρων, ικανών να ενσωματώνουν νέες πολιτικές χρονοπρογραμματισμού, μηχανισμούς δίκαιης κατανομής και διάθεσης των πόρων, αλλά και ασφάλειας, χωρίς να διακυβεύεται η σταθερότητα του παραγωγικού περιβάλλοντος.

Με την κλιμάκωση των εγκαταστάσεων HPC από δεκάδες σε χιλιάδες κόμβους, οι στατικές πολιτικές κατανομής πόρων και ο τρόπος που παραδοσιακά προγραμματίζονταν τα συστήματα αυτά οδηγούν σε αυξημένους χρόνους αναμονής, μειωμένη απόδοση και μερική μόνο αξιοποίησή τους. Η μεγέθυνση του φόρτου εργασιών στα συστήματα HPC δημιουργεί συμφόρηση στα υποσυστήματα δικτύωσης και αποθήκευσης, δυσχεραίνοντας την πρόβλεψη της απόδοσης και την εξισορρόπηση του φορτίου. Ελλείψει μηχανισμών ελέγχου για την αξιολόγηση εναλλακτικών στρατηγικών κατανομής σε ρεαλιστικές συνθήκες, οι ερευνητές/τριες και όσοι/ες διαχειρίζονται τα συστήματα HPC, αδυνατούν να ποσοτικοποιήσουν πιθανά οφέλη ή να εντοπίσουν εναλλακτικές λύσεις στον σχεδιασμό των αλγορίθμων.

Η αυξανόμενη ζήτηση για υπηρεσίες HPC, από τη μοντελοποίηση κλιματικών φαινομένων έως τη μηχανική μάθηση [1], οδηγεί σε ετερογενείς εργασίες που συνδυάζουν διαφορετικούς τύπους φορτίων, διαδραστικές αναλύσεις και μακροχρόνιες προσομοιώσεις. Τα παραδοσιακά συστήματα διαχείρισης πόρων στερούνται δυναμικής προσαρμοστικότητας, βασιζόμενα σε στατικές πολιτικές που καθίστανται σύντομα παρωχημένες.

Επιπλέον η εμφάνιση υβριδικών συστημάτων, τα οποία ενσωματώνουν κβαντικούς υπολογιστές παράλληλα με κλασικούς υπολογιστικούς κόμβους [2], προσθέτει ένα επιπλέον επίπεδο πολυπλοκότητας. Οι κβαντικές διεργασίες απαιτούν αυστηρό συν-προγραμματισμό σε συνέργεια με τις εργασίες κλασσικής υπολογιστικής, καθώς και εξειδικευμένες πολιτικές κατανομής πόρων. Οι υφιστάμενοι διαχειριστές πόρων προσφέρουν περιορισμένη υποστήριξη για τέτοιου είδους ροές εργασίας, και κάθε επέκταση πρέπει να επικυρώνεται σε πραγματικές συνθήκες.

7.1 Διαχείριση Πόρων σε Υπολογιστικά Συστήματα Υψηλών Επιδόσεων

7.1.1 Περιορισμοί στην Έρευνα και Αξιοποίηση των Συστημάτων Υψηλών Επιδόσεων

Παρά την εντυπωσιακή εξέλιξη της αρχιτεκτονικής επεξεργαστών και συστημάτων, τα υπολογιστικά συστήματα υψηλών επιδόσεων εξακολουθούν να αντιμετωπίζουν σημαντικούς περιορισμούς που επηρεάζουν τη δυνατότητα κλιμάκωσης, την ενεργειακή αποδοτικότητα και τη χρηστικότητά τους. Οι κύριες προκλήσεις εντοπίζονται στα συστήματα πρόσβασης στη μνήμη, τη διασύνδεση κόμβων, την αποθήκευση δεδομένων, την κατανάλωση ενέργειας, την ετερογένεια των αρχιτεκτονικών και τη δυσκολία αποδοτικής παραλληλοποίησης.

Η αναντιστοιχία μεταξύ της ταχύτητας επεξεργασίας και της απόδοσης της μνήμης συνιστά έναν από τους σημαντικότερους περιορισμούς. Το εύρος ζώνης και η καθυστέρηση πρόσβασης στη μνήμη δεν έχουν βελτιωθεί με τον ίδιο ρυθμό με την υπολογιστική ισχύ, με αποτέλεσμα η αναμονή δεδομένων να καθίσταται κυρίαρχος παράγοντας κόστους σε χρόνο και ενέργεια. Επιπλέον, η διακίνηση δεδομένων μεταξύ των διάφορων επιπέδων της ιεραρχίας μνήμης αλλά και μεταξύ των κόμβων αυξάνει την ενεργειακή κατανάλωση και επιβραδύνει τις εφαρμογές μεγάλης κλίμακας[4]. Η μνήμη είναι, επίσης, και ένας σημαντικός παράγοντας κόστους, καθώς αντανακλά περίπου το 20% του κόστους των εμπορικών συστημάτων υψηλών επιδόσεων[5].

Αντίστοιχα, οι τεχνολογίες διασύνδεσης (*interconnects*) αποτελούν κρίσιμο πεδίο έρευνας, καθώς η καθυστέρηση και η περιορισμένη χωρητικότητα των διαύλων επικοινωνίας μεταξύ κόμβων οδηγούν σε συστήματα που περιορίζονται από τη μεταφορά δεδομένων (*memory-bound*) αντί της καθαρής υπολογιστικής τους ισχύος. Οι επικρατέστερες τεχνολογίες διασύνδεσης στα κορυφαία HPC συστήματα είναι οι *Infiniband*, *Gigabit Ethernet*, *HPE Slingshot* και *Intel Omnipath*.

Η απόδοση των συστημάτων HPC περιορίζεται όλο και περισσότερο από το πρόβλημα της αποθήκευσης (”I/O wall”), όπου η ταχύτητα των επεξεργαστών υπερέχει της ικανότητας του υποσυστήματος αποθήκευσης να παρέχει και να αποθηκεύει δεδομένα. Ως αποτέλεσμα, μεγάλο ποσοστό των υπολογιστικών δυνατοτήτων των Συστημάτων Παράλληλης Επεξεργασίας παραμένει αδρανές αναμένοντας δεδομένα, με συνακόλουθη μείωση της συνολικής αποδοτικότητας του συστήματος. Το πρόβλημα επιδεινώνεται στην εποχή των υπολογιστικών συστημάτων με δυνατότητα 10^{18} υπολογισμών κινητής υποδιαστολής/δευτερόλεπτο, όπου ο τεράστιος βαθμός παραλληλισμού και οι απαιτητικές σε I/O εργασίες προκαλούν κύματα αιτημάτων I/O που τα συστήματα δυσκολεύονται να διαχειριστούν[9, 10].

Παράλληλα, οι ενεργειακές απαιτήσεις των συστημάτων HPC αυξάνονται ραγδαία. Η συνολική απόδοση των κορυφαίων συστημάτων έχει αυξηθεί ταχύτερα από την ενεργειακή αποδοτικότητά τους, οδηγώντας σε μεμονωμένα συστήματα που καταναλώνουν δεκάδες MWh. Γενικότερα, τα παράλληλα συστήματα επεξεργασίας έχουν ενεργειακό αποτύπωμα της τάξης των εκατοντάδων TWh ετησίως[11, 12]. Κατά συνέπεια, η ενσωμάτωση κριτηρίων ενεργειακής αποδοτικότητας στον σχεδιασμό και τη λειτουργία των HPC κέντρων αποτελεί πλέον βασική απαίτηση.

Η αρχιτεκτονική ετερογένεια (π.χ. CPU + GPU) υπόσχεται επιτάχυνση, αλλά εισάγει σοβαρά εμπόδια: το κόστος μεταφοράς δεδομένων μεταξύ ξεχωριστών χώρων μνήμης, οι περιορισμοί του εύρους ζώνης των διασυνδέσεων και η σύνθετη συνύπαρξη υλοποιήσεων διαφορετικών προγραμματιστικών μοντέλων (π.χ. OpenMP, CUDA) οδηγούν

σε υποεκμετάλλευση πόρων [13, 14]. Η κατανομή του υπολογιστικού φορτίου στις πιο κατάλληλες μονάδες κάθε φορά, καθώς και η εξάλειψη των εξαρτήσεων μεταξύ των μονάδων, αποτελούν κρίσιμα αντικείμενα έρευνας προκειμένου να απελευθερωθούν οι δυνατότητες τέτοιων αρχιτεκτονικών.

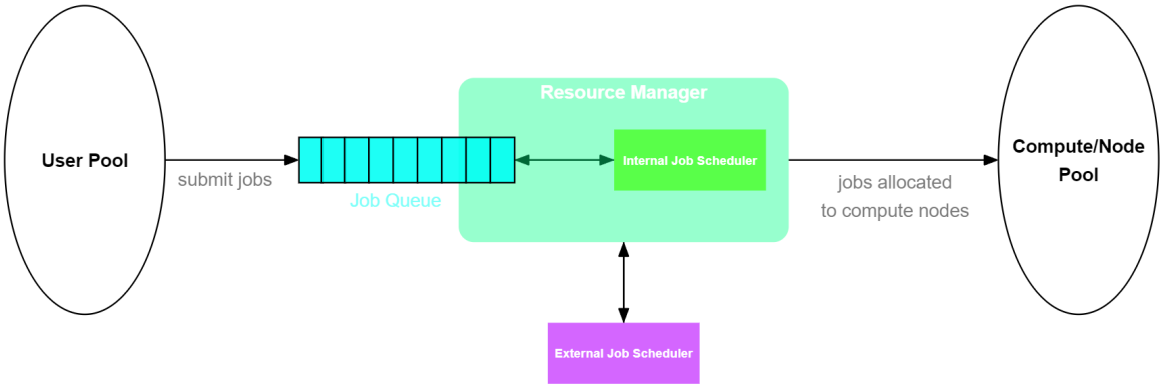
Τέλος, ο θεμελιώδης περιορισμός που διατυπώνει ο Νόμος του Amdahl υπενθυμίζει ότι το μέγιστο θεωρητικό όφελος από την παραλληλοποίηση του λογισμικού περιορίζεται από το μη παραλληλοποιήσιμο τμήμα του[15]. Στην πράξη, καθώς τα συστήματα κλιμακώνονται σε χιλιάδες έως εκατομμύρια πυρήνες, το όριο αυτό θα αποτρέπει ένα πρόγραμμα από την επίτευξη του θεωρητικά μέγιστου ορίου υπολογισμών που επιτρέπει το σύστημα στο οποίο εκτελείται, αρκετές φορές καθιστώντας την ταχύτητα εκτέλεσης του προγράμματος να μην ακολουθεί το ρυθμό κλιμάκωσης του υλικού.

7.1.2 Διαχειριστής Πόρων SLURM

Η εμφάνιση των συστημάτων διαχείρισης πόρων (Resource Management Systems) αποτέλεσε απάντηση στις προκλήσεις της ετερογένειας, της διασύνδεσης και της ενεργειακής αποδοτικότητας στα σύγχρονα συστήματα HPC. Τα RMS διαδραματίζουν καίριο ρόλο στη διαδικασία κατανομής πόρων στους χρήστες, προσφέροντας ενιαία διεπαφή πρόσβασης, ανεξαρτήτως υποκείμενης αρχιτεκτονικής. Παρέχουν, επίσης, μηχανισμούς μέσω των οποίων οι υπολογιστικοί πόροι καθίστανται διαθέσιμοι σε διαφορετικές κατηγορίες χρηστών, ενώ παράλληλα διατηρούν ακριβή καταγραφή της χρήσης και εφαρμογή των κατάλληλων πολιτικών χρέωσης.

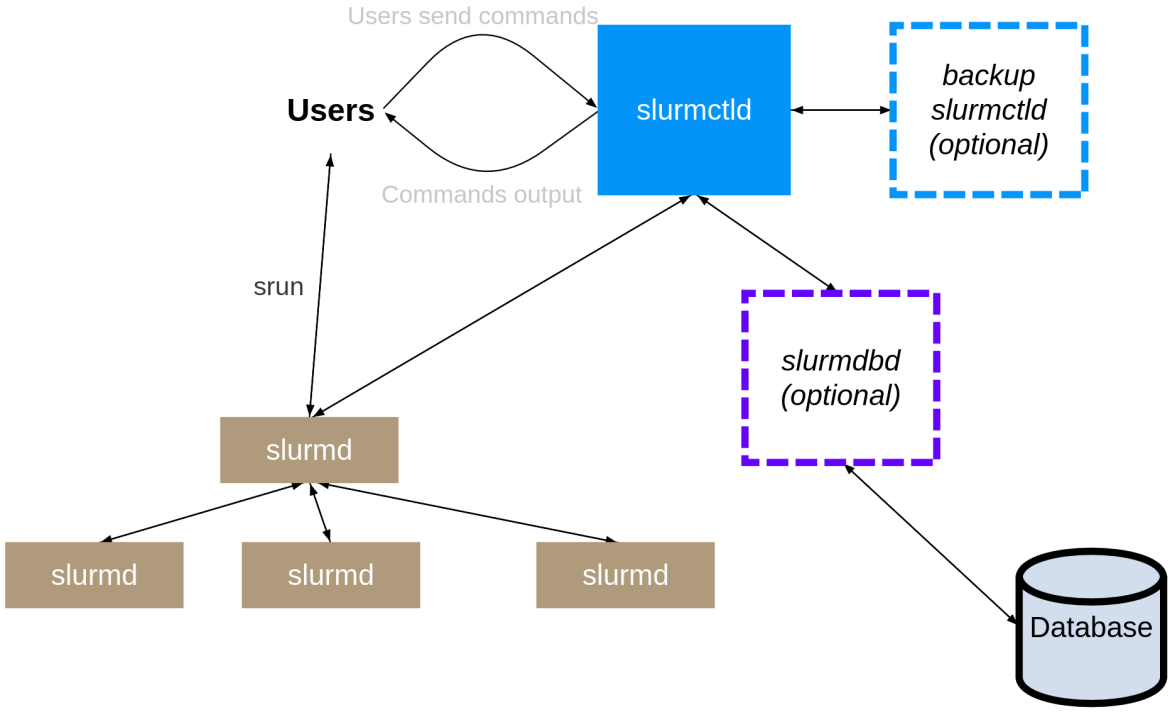
Το *Slurm* (Simple Linux Utility for Resource Management) αποτελεί ένα ανοιχτού κώδικα, και ιδιαίτερα επεκτάσιμο σύστημα διαχείρισης και προγραμματισμού εργασιών για συστοιχίες Linux [16]. Δεν απαιτεί αλλαγές στον πυρήνα του λειτουργικού συστήματος των Linux και λειτουργεί αυτόνομα, επιτελώντας τρεις βασικές λειτουργίες: (α) την ανάθεση υπολογιστικών κόμβων στους χρήστες για ορισμένο χρονικό διάστημα, (β) την εκκίνηση, παρακολούθηση και εκτέλεση παράλληλων εργασιών στους κόμβους, και (γ) τη διαχείριση του ανταγωνισμού για πόρους μέσω ουρών εργασιών, υποστηρίζοντας ταυτόχρονα επεκτάσεις για την προσαρμογή της λειτουργικότητας.

Τα παραπάνω μπορούν να γενικευτούν για τους διαχειριστές πόρων όλων συστημάτων HPC. Έτσι οι τρεις κρίσιμες λειτουργίες: (1) η κατανομή πόρων με εφαρμογή συγκεκριμένης πολιτικής, (2) ο προγραμματισμός εργασιών και (3) η παρακολούθηση εκτέλεσης αποτελούν τις ελάχιστες λειτουργίες που πρέπει να επιτελεί ένας διαχειριστής πόρων. Οι λειτουργίες αυτές παρατίθενται σχηματικά στο Σχ.2.1.



Σχήμα 7.1: Απεικόνιση της λειτουργίας ενός αφηρημένου διαχειριστή πόρων

Η αντίστοιχη αρχιτεκτονική υψηλού επιπέδου για το Slurm είναι παρόμοια, όπως φαίνεται στο Σχ.2.2.



Σχήμα 7.2: Αφαιρετική απεικόνιση του διαχειριστή εργασιών Slurm

7.1.2.1 Ελεγκτής Slurm (slurmctld)

Στο πλαίσιο λειτουργίας του Slurm, η διαχείριση της κατάστασης του συστήματος επιτελείται κυρίως από τον ελεγκτή `slurmctld`, ο οποίος αποτελεί το κεντρικό στοιχείο ελέγχου του συστήματος. Ο δαίμονας αυτός είναι πολυνηματικός και χρησιμοποιεί διακριτούς μηχανισμούς κλειδώματος για λειτουργίες ανάγνωσης και εγγραφής, εξασφαλίζοντας κλιμακωσιμότητα και αποδοτικότητα. Κατά την εκκίνηση, φορτώνει τις παραμέτρους του

αρχείου ρυθμίσεων (`slurm.conf`) και τυχόν αποθηκευμένη κατάσταση από προηγούμενες συνεδρίες, ενώ υποστηρίζει ανθεκτικότητα σε σφάλματα μέσω περιοδικής αποθήκευσης της πλήρους κατάστασής του στο δίσκο. Ο `slurmctld` δύναται να λειτουργήσει είτε σε κύρια είτε σε εφεδρική λειτουργία (master/standby), επιτρέποντας ανάληψη ελέγχου από τον εφεδρικό κόμβο σε περίπτωση σφάλματος. Δεν απαιτεί δικαιώματα διαχειριστή (root), καθώς εκτελείται υπό έναν εξειδικευμένο χρήστη συστήματος που ορίζεται μέσω της παραμέτρου `SlurmUser` στο αρχείο ρυθμίσεων.

Οι βασικές εσωτερικές υπομονάδες του `slurmctld` είναι τρεις:

- **Node Manager:** Παρακολουθεί την επιχειρησιακή κατάσταση όλων των κόμβων της συστοιχίας μέσω περιοδικής δειγματοληψίας ή ασύγχρονων ενημερώσεων από τους δαίμονες `slurmd`. Πριν ένας κόμβος χρησιμοποιηθεί για εκτέλεση, ελέγχεται αν πληροί τις απαιτούμενες παραμέτρους ρύθμισης.
- **Partition Manager:** Ομαδοποιεί τους κόμβους σε διαμερίσεις (*partitions*), αντιστοιχίζοντάς τους σε εργασίες βάσει κατάστασης και ρυθμίσεων. Επιπλέον, εκτελεί διαχειριστικές εντολές που τροποποιούν παραμέτρους κόμβων ή διαμερίσεων.
- **Job Manager:** Διαχειρίζεται τα αιτήματα υποβολής εργασιών, τα οποία διατηρεί σε ουρά προτεραιότητας έως ότου εξασφαλιστούν πόροι για την εκτέλεσή τους. Ενεργοποιείται από γεγονότα του συστήματος (π.χ. ολοκλήρωση εργασιών, ενεργοποίηση κόμβων κ.ά.) και, όταν οι συνθήκες το επιτρέπουν, επιλέγει και εκκινεί τις κατάλληλες εργασίες στις αντίστοιχες διαμερίσεις, μεταδίδοντας στους επιλεγμένους κόμβους τα απαραίτητα δεδομένα εκτέλεσης.

7.1.2.2 Δαίμονας Slurm (slurmd)

Αντίστοιχα, οι δαίμονες `slurmd` αποτελούν πολυνηματικά προγράμματα εγκατεστημένα στους υπολογιστικούς κόμβους, επιφορτισμένα με την εκτέλεση εντολών του `slurmctld`. Οι κύριες αρμοδιότητές τους περιλαμβάνουν την ανάλυση των ρυθμίσεων συστήματος, την καταχώριση της τρέχουσας κατάστασης στον ελεγκτή, την αναμονή και εκτέλεση εργασιών, καθώς και την επιστροφή αποτελεσμάτων εκτέλεσης.

Η επικοινωνία μεταξύ `slurmd` και `slurmctld` είναι ασύγχρονη και αμφίδρομη, με ανταλλαγή πληροφοριών για την κατάσταση των ενεργών εργασιών και των αντίστοιχων κόμβων. Η διαχείριση κατάστασης στον `slurmd` παραμένει εσκεμμένα ελαχιστοποιημένη, περιοριζόμενη μόνο στα δεδομένα που αφορούν τις τρέχουσες εκτελέσεις. Το σχεδιαστικό αυτό χαρακτηριστικό διασφαλίζει την αποδοτικότητα, την ανθεκτικότητα και την ευκολία ανάκαμψης του συστήματος σε περιβάλλοντα μεγάλης κλίμακας.

7.1.3 Άλλοι Διαχειριστές Πόρων για HPC

Πέραν του Slurm, υπάρχουν και άλλοι διαχειριστές πόρων για συστήματα υψηλών επιδόσεων, από τους οποίους θα αναφερθούν οι πιο διαδεδομένοι που είναι ανοικτού κώδικα.

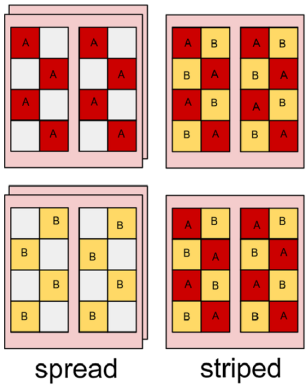
Flux: Το Flux αποτελεί ένα ευέλικτο πλαίσιο διαχείρισης πόρων και εργασιών που αντιμετωπίζει περιορισμούς των παραδοσιακών χρονοδρομολογητών σε ετερογενή ή πολυεπίπεδα περιβάλλοντα. Υιοθετεί ιεραρχική αρχιτεκτονική με φωλιασμένους χρονοδρομολογητές, προσφέροντας λεπτομερή έλεγχο και υψηλή κλιμακωσιμότητα μέσω αποκεντρωμένης λήψης αποφάσεων. Μία καινοτομία του είναι πως μπορεί να υιοθετηθεί σε συστήματα που ήδη έχουν έναν εγκατεστημένο διαχειριστή πόρων[18].

PBS Pro: Το PBS Professional είναι ώριμο σύστημα διαχείρισης εργασιών και χρονοδρομολόγησης, προερχόμενο από πρωτοβουλία της NASA. Υποστηρίζει προχωρημένες δυνατότητες, όπως δεσμεύσεις πόρων, κατανεμημένη χρονοδρομολόγηση και δυναμική αξιοποίηση ανενεργών συστημάτων. Παρέχει επίσης πλαίσιο ορισμού προσαρμοσμένων πόρων (π.χ. GPU, άδειες χρήσης), προσφέροντας μεγάλη ευελιξία, αν και απαιτεί πρόσθετη διαχειριστική υποστήριξη[19].

7.2 Υπόβαθρο και Σχετικές Εργασίες

Στα συστήματα υψηλών επιδόσεων, η παραδοσιακή πολιτική κατανομής πόρων απομονώνει ταυτόχρονες εργασίες σε διαφορετικούς κόμβους, καθότι αποτελεί αφενός την απλούστερη πολιτική κατανομής και κατά δεύτερον αφαιρεί πολυπλοκότητα από σύνθετες αλληλεπιδράσεις μεταξύ των διάφορων πόρων της μνήμης και των διασυνδέσεων. Ωστόσο, αυτή η προσέγγιση οδηγεί σε μειωμένη αποδοτικότητα και χαμηλότερη ενεργειακή αξιοποίηση. Η συνεκτέλεση (*co-execution* ή *co-location*) προτείνεται ως εναλλακτική στρατηγική, κατά την οποία εργασίες μοιράζονται τους ίδιους υπολογιστικούς πόρους, βελτιώνοντας τη συνολική χρησιμοποίηση του συστήματος[24].

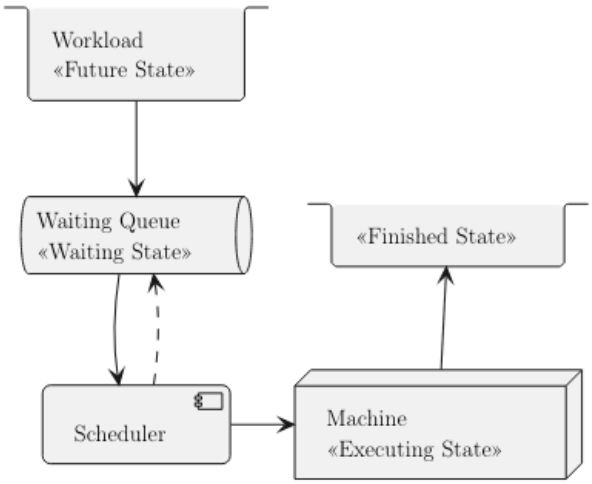
Μελέτες[25, 26] έχουν δείξει ότι η συνεκτέλεση ετερογενών εφαρμογών (π.χ. ο συνδυασμός υπολογιστικά απαιτητικών και απαιτητικών σε μνήμη εφαρμογών) μπορεί να αυξήσει σημαντικά την απόδοση του συστήματος. Από φυσική άποψη, η συνεκτέλεση ερμηνεύεται ως τεχνική αντιστοίχισης διεργασιών σε πυρήνες, με δύο κύριες μορφές: **striped** και **spread**.



Σχήμα 7.3: Μέθοδοι χρονοδρομολόγησης δύο εφαρμογών 16 διεργασιών (*A* και *B*) σε υπερυπολογιστή με δύο υποδοχές επεξεργαστών 8 πυρήνων: στο *spread* κάθε εφαρμογή εκτελείται απομονωμένα σε ξεχωριστούς κόμβους, ενώ στο *striped* μοιράζονται τους ίδιους πόρους[24].

Η επιλογή των εργασιών που θα συνεκτελεστούν αποτελεί ενεργό πεδίο έρευνας[27–30], με προτάσεις που βασίζονται σε πολιτικές του χρονοδρομολογητή για τη βέλτιστη κατανομή πόρων. Η κατανομή εφαρμογών σε πολλούς κόμβους αλλά και σε διαφορετικούς πυρήνες εντός των κόμβων αυτών, μπορεί να οδηγήσει σε μείωση των συγκρούσεων για κοινούς πόρους και να βελτιστοποιήσει τη χρήση επιπέδων ιεραρχίας της μνήμης καθώς και των επιταχυντών. Επίσης, η μειωμένη επικοινωνία εντός του κόμβου αρκετές φορές οδηγεί στη βελτίωση της απόδοση παράλληλων εφαρμογών ανταλλαγής μηνυμάτων. Από τη σκοπιά του συστήματος, ο συνδυασμός στρατηγικών *spread* και *striped* μειώνει τον κατακερματισμό των πόρων, ενισχύοντας τη συνολική αποδοτικότητα του συστήματος[31].

Η πολυπλοκότητα της ανάλυσης και αξιολόγησης πολιτικών κατανομής πόρων οδήγησε στην ανάπτυξη του **Efficient Lightweight Scheduling Estimator (ELiSE)** από το Εργαστήριο Υπολογιστικών Συστημάτων του ΕΜΠ[36]. Το ELiSE είναι ένας προσομοιωτής σε Python που κάνει χρήση της λειτουργικότητας `perf` των Linux για ταχεία ανάπτυξη και αξιολόγηση αλγορίθμων χρονοδρομολόγησης με δυνατότητα συνεκτέλεσης. Εστιάζει στην προσομοίωση πολιτικών χρονοδρομολόγησης αντί της πλήρους προσομοίωσης συστημάτων, προσφέροντας ευελιξία και ταχύτητα στη διερεύνηση συμπεριφορών συνεκτέλεσης.



Σχήμα 7.4: Αρχιτεκτονική υψηλού επιπέδου του εργαλείου ELiSE[31]

Το εργαλείο δέχεται ως είσοδο μία περιγραφή του συστήματος HPC το οποίο θα προσομοιώσει. Ως έξοδο μπορεί να παράξει σχηματικές αναπαραστάσεις της εκτέλεσης μέσω μετρικών ή γραφικών που αποτυπώνουν την επίδραση της συνεκτέλεσης και χρονοδρομολόγησης των εφαρμογών. Η επεκτασιμότητα του ELiSE επιτρέπει την υλοποίηση και δοκιμή νέων αλγορίθμων, μέσω αφηρημένης ιεραρχίας κλάσεων. Οι ήδη ενσωματωμένοι χρονοδρομολογητές, επί του παρόντος, περιλαμβάνουν παραλλαγές του FCFS καθώς και έναν έξυπνο χρονοδρομολογητή, τον *Filler*, που αξιοποιεί αδρανείς πόρους.

7.3 Σχεδιασμός και Αρχιτεκτονική του Εργαλείου Εφήμερων Slurm

7.3.1 Το Σύστημα ARIS

Το **ARIS** αποτελεί τον ελληνικό υπερυπολογιστή που αναπτύχθηκε και λειτουργεί από τη GRNET στην Αθήνα. Εντάχθηκε στη λίστα Top500 τον Ιούνιο του 2015 (θέση 468) και, παρότι σήμερα θεωρείται σύστημα αρκετά ”ηλικιωμένο”, παραμένει ενεργό με σημαντική συμβολή σε επιστημονικά και ερευνητικά έργα. Το περιβάλλον ανάπτυξης του εργαλείου **SiS** βασίστηκε εξ ολοκλήρου στο σύστημα ARIS, με όλα τα πειράματα να πραγματοποιούνται σε κανονική λειτουργία του συστήματος χωρίς επίδραση στην απόδοσή του.

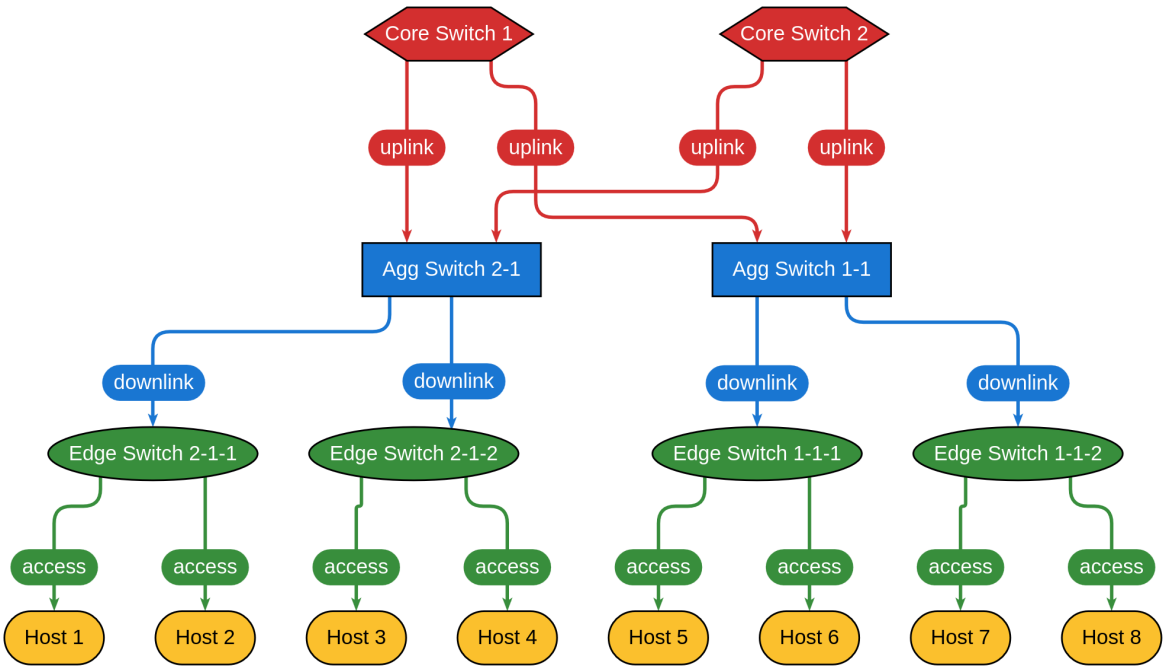
Το ARIS αποτελείται από 533 υπολογιστικούς κόμβους οργανωμένους σε πέντε λογικές διαμερίσεις, οι οποίοι συνδέονται μέσω δικτύου Infiniband FDR σε τοπολογία fat tree, όπως φαίνεται στο Σχ. 7.5. Το σύστημα χρησιμοποιεί λειτουργικό Red Hat/CentOS 6.7, διαχειριστή πόρων Slurm, διαχείριση μέσω xCAT IBM και παρακολούθηση με Nagios και Ganglia. Η

αποθήκευση επιτυγχάνεται μέσω IBM GPFS χωρητικότητας 2 PB και ρυθμαπόδοσης 6 GB/s, ενώ η πρόσβαση παρέχεται μέσω δύο κόμβων σύνδεσης (login nodes) με SSH.

Πίνακας 7.1: Χαρακτηριστικά Συστήματος ARIS

Architecture	x86-64
Operating System	Redhat/Centos 6.7
Interconnect	
Technology	Infiniband FDR
Topology	Fat tree
Bandwidth [Gb/s]	56
Storage	
Type	IBM GPFS
Size [PByte]	2
Bandwidth [GB/s]	6
System Software	
Batch system	SLURM
System Management	xCat IBM
Monitoring	Nagios, Ganglia

Η τοπολογία του ARIS ακολουθεί τη διάταξη Fat Tree, όπως φαίνεται στο Σχ.7.5.



Σχήμα 7.5: Αφηρημένη τοπολογία τύπου Fat Tree, την οποία ακολουθούν οι κόμβοι του συστήματος ARIS.

Οι υπολογιστικοί κόμβοι του ARIS περιγράφονται στον Πίν.7.2

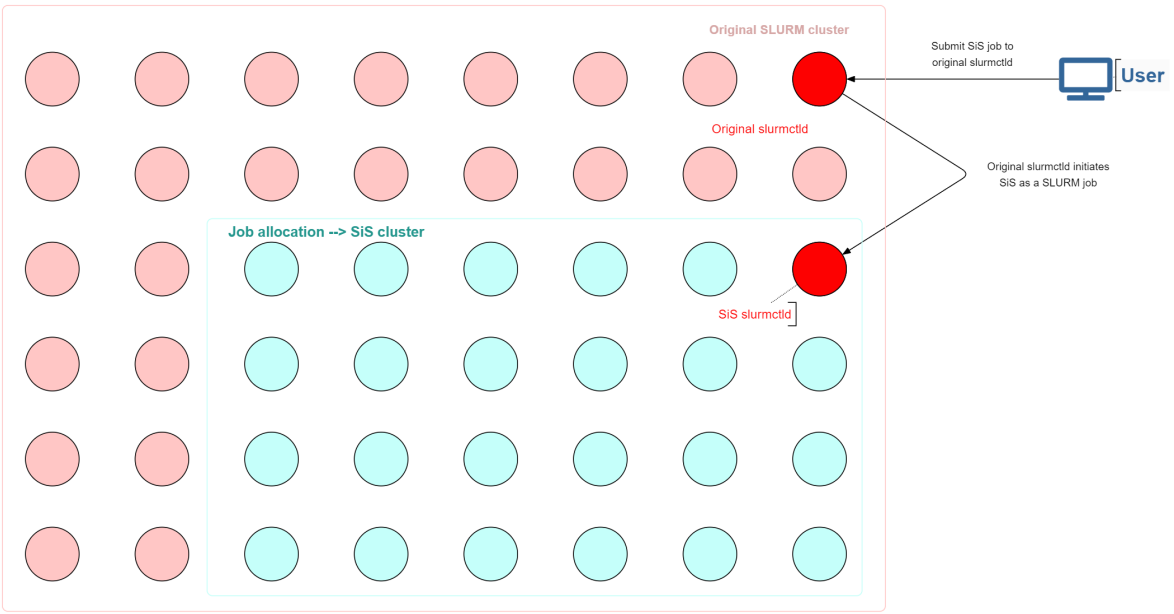
Πίνακας 7.2: Κατηγορίες Υπολογιστικών Κόμβων του ARIS

Τύπος Κόμβου	Πλήθος	Επιταχυντής	Μνήμη	Πυρήνες
THIN	426	–	64 GB	20@2.8 GHz
GPU	44	2x Tesla K40m	64 GB	20@2.6 GHz
PHI	18	2x Xeon Phi 7120p	64 GB	20@2.6 GHz
FAT	44	–	512 GB	40@2.4 GHz
ML	1	8x Volta V100	512 GB	40@2.2 GHz

Η ανάπτυξη του **SiS** πραγματοποιήθηκε κυρίως με τη χρήση των **THIN** κόμβων, οι οποίοι αποτελούν την διαμέριση compute του ARIS.

7.3.2 Αρχιτεκτονική Εργαλείου Παραγωγής Εφήμερων Slurm

Το SiS σχεδιάστηκε ως ένα ελαφρύ εργαλείο σε επίπεδο χρήστη που εκτελείται πλήρως εντός του συστήματος batch του Slurm, χωρίς την ανάγκη αυξημένων δικαιωμάτων, πέραν του δικαιώματος υποβολής κανονικών εργασιών στο σύστημα Slurm του συστήματος. Όπως φαίνεται στο Σχ. 7.6, το SiS λειτουργεί ως κανονική εργασία Slurm ενώ ταυτόχρονα δημιουργεί ένα εσωτερικό, εμφωλευμένο Slurm.



Σχήμα 7.6: Απλοποιημένη απεικόνιση της λειτουργίας του SiS εντός ενός Slurm cluster

7.3.2.1 Κύκλος Ζωής Εφήμερου Slurm

Μια εργασία SiS είναι εφήμερη, αυτόνομη και απομονωμένη από το Slurm του εξωτερικού συστήματος. Αυτό σημαίνει πως σε κάθε εκτέλεση το SiS αναπτύσσεται εκ νέου και εξαρχής, με όσες πληροφορίες κατάστασης θέλει ο χρήστης να έχει διατηρήσει σε κάθε εκτέλεση, με τη διαγραφή κάθε προηγούμενης πληροφορίας κατάστασης να αποτελεί την προεπιλογή. Ο κύκλος ζωής του SiS, λοιπόν, ακολουθεί τα εξής βήματα:

1. Υποβολή του `v-slurm-lite.sh` προγράμματος μέσω `sbatch` του εξωτερικού Slurm. Το συγκεκριμένο αρχείο πραγματοποιεί την εκκίνηση του SiS.
2. Την ίδια στιγμή γίνεται η εκχώρηση κόμβων από τον εξωτερικό ελεγκτή `slurmctld`.
3. Κατά τη διάρκεια εκτέλεσης του `v-slurm-lite.sh` δημιουργείται ο SiS cluster με τον δικό του ελεγκτή `slurmctld` και παραμένει σε λειτουργία αναλόγως των ρυθμίσεων που έχει ορίσει ο χρήστης.
4. Επικοινωνία μεταξύ του ελεγκτή SiS και των δαιμόνων SiS `slurmd` για εκτέλεση εργασιών που έχει ορίσει ο χρήστης.
5. Ομαλός τερματισμός των δαιμόνων του SiS μετά την ολοκλήρωση, διακοπή από το χρήστη ή λήξη του χρόνου της εργασίας.

7.3.2.2 Ενσωμάτωση με Εργασίες εξωτερικού Slurm

Το SiS εκτελείται ως batch εργασίας του εξωτερικού Slurm και αξιοποιεί τους μηχανισμούς του:

- Οι διεργασίες και τα βήματα εργασίας ορίζονται μέσω `--ntasks`, `--ntasks-per-node` και `--cpus-per-task` καθώς και άλλων σχετικών παραμέτρων που προσφέρει το Slurm.
- Κάθε διεργασία του SiS καταναλώνει πραγματικούς πόρους του συστήματος σε μία διαμέρισή του και "νοητούς" πόρους του SiS, δηλαδή αποτυπώσεις των πραγματικών πόρων του εξωτερικού συστήματος όπως αναγιγνώσκονται από το SiS.
- Οι ταυτότητες εργασιών (`jobid`) χρησιμοποιούνται για την παρακολούθηση των εργασιών. Μία εργασία SiS έχει δικό της αριθμό εργασίας εντός του SiS που δεν είναι γνωστός στο εξωτερικό Slurm.
- Οι πληροφορίες σχετικά με τη διαθσιμότητα και τη χρήση πόρων από τους χρήστες.

Επιπλέον, μία πολύτιμη ικανότητα του SiS είναι η ανάπτυξη προσαρμοσμένων εγκαταστάσεων Slurm, στις οποίες ο πηγαίος κώδικας έχει υποστεί άμεσες τροποποιήσεις. Αυτή η προσέγγιση αντιδιαστέλλεται με την κυρίαρχη μεθοδολογία χρήσης plugins, η οποία, αν και επεκτάσιμη, περιορίζεται εγγενώς από τις προκαθορισμένες διεπαφές προγραμματισμού εφαρμογών (APIs). Το SiS υπερβαίνει αυτόν τον περιορισμό, επιτρέποντας στους ερευνητές να εκτελούν και να διαμορφώνουν στιγμιότυπα εγκαταστάσεων και τροποποιημένων εκδόσεων του Slurm στα οποία ο πηγαίος κώδικας έχει αλλαγές, διευρύνοντας έτσι σημαντικά το πεδίο της πειραματικής έρευνας.

Η σημασία αυτής της δυνατότητας είναι διττή. Πρωτίστως, καθιστά εφικτή τη δοκιμή πειραματικών μηχανισμών χρονοπρογραμματισμού και συνεκτέλεσης, κατανομής πόρων ή βελτιώσεων ασφαλείας που απαιτούν θεμελιώδεις αλλαγές στα κεντρικά στοιχεία του Slurm, κάτι που προηγουμένως ήταν ανέφικτο χωρίς παρέμβαση σε συστήματα που βρίσκονται σε λειτουργία. Δευτερευόντως, το SiS παρέχει ένα ελεγχόμενο και απομονωμένο περιβάλλον εκτέλεσης, όπου διαφορετικές εκδόσεις του Slurm, συμπεριλαμβανομένων εκείνων με τροποποιημένο πηγαίο κώδικα, μπορούν να συνυπάρχουν και να υποβάλλονται σε συστηματική συγκριτική αξιολόγηση (benchmarking). Αυτή η εξέλιξη γεφυρώνει το χάσμα μεταξύ του θεωρητικού σχεδιασμού αλγορίθμων και της πρακτικής τους επικύρωσης σε ένα λειτουργικό περιβάλλον.

Συμπερασματικά, το SiS όχι μόνο διατηρεί τα πλεονεκτήματα επεκτασιμότητας του παραδοσιακού συστήματος των plugins, αλλά τα διευρύνει, επιτυγχάνοντας υψηλότερο βαθμό ευελιξίας και πειραματικής ελευθερίας. Επίσης, είναι πλέον εφικτά, σενάρια δοκιμών που προηγουμένως παρέμεναν απρόσιτα, προάγοντας την καινοτομία σε τομείς όπως ο χρονοπρογραμματισμός εργασιών και η διαχείριση πόρων. Ως εκ τούτου, το SiS καθίσταται ιδιαίτερα πολύτιμο τόσο για την ακαδημαϊκή έρευνα όσο και για τη βιομηχανία, επιταχύνοντας την πρακτική επικύρωση και εφαρμογή βελτιώσεων σε υποδομές HPC.

7.4 Υλοποίηση και Ενσωμάτωση

7.4.1 Εκδοχή στο Σύστημα ARIS

Ένας αναγνώστης πολύ εύλογα θα θέσει το ερώτημα σχετικά με το λόγο για τον οποίο το SiS, ενώ υπήρξε εκτεταμμένη αναφορά σε προηγούμενα κεφάλαια, πως μπορεί να αναπτύξει και να εκτελέσει οποιαδήποτε έκδοση του Slurm, μάλιστα, αυτός είναι και ένας από τους διακηρυγμένους λόγους δημιουργίας του, περιορίστηκε στην ίδια έκδοση Slurm (16.05) με αυτήν του εξωτερικού συστήματος του ARIS.

Ως προς το παραπάνω ερώτημα, η απάντηση συνδέεται άμεσα με την υποδομή του συστήματος ARIS, όπως αυτή αναλύθηκε στην Ενότητα 7.3.1. Το υποκείμενο λειτουργικό σύστημα του ARIS είναι το `RedHat/CentOS 6.7`, μια παλιότερη και εδώ και αρκετά χρόνια μη υποστηριζόμενη έκδοση των Linux(ημερομηνία λήξης υποστήριξης: 30 Νοεμβρίου 2020 [38]). Το πρόβλημα προκύπτει εξαιτίας του γεγονότος πως οι νεότερες εκδόσεις του Slurm αξιοποιούν κλήσεις συστήματος που παρέχονται μόνο από ενημερωμένους πυρήνες του Linux. Παρόλα αυτά, ακόμα και αυτό δεν αποτελεί πρόβλημα για το SiS στην περίπτωση που ο χρήστης έχει τη διάθεση να εμπλακεί με τον πηγαίο κώδικα των νεότερων εκδόσεων του Slurm, ώστε να τον καταστήσει συμβατό με παλιότερες εκδόσεις του πυρήνα των Linux. Ωστόσο, αυτή η απαίτηση ξεπερνά τα πλαίσια της παρούσας διπλωματικής εργασίας.

Κατά συνέπεια, η ανάπτυξη του SiS εντός της υποδομής του ARIS περιορίστηκε αναγκαστικά στην έκδοση Slurm 16.05.11, η οποία είναι η εγγενώς υποστηριζόμενη έκδοση από το σύστημα. Εν προκειμένω, το SiS δεν εισάγει αυτόνομα μια πιο πρόσφατη στοίβα λογισμικού Slurm, αλλά συμμορφώνεται με το συγκεκριμένο περιβάλλον λογισμικού που είναι ήδη εγκατεστημένο στο ARIS.

7.4.1.1 Διαθεσιμότητα και Δυνατότητα Αναπαραγωγής Κώδικα

Ο πηγαίος κώδικας του πλαισίου SiS (Slurm in Slurm) έχει καταστεί δημοσίως προσβάσιμος μέσω ενός **αποθετηρίου GitHub**, με στόχο τη διασφάλιση της δυνατότητας αναπαραγωγής και της ευκολίας πρόσβασης από την ερευνητική κοινότητα. Το εν λόγω αποθετήριο περιέχει λεπτομερείς οδηγίες για την εγκατάσταση και εκτέλεση του SiS σε διάφορα περιβάλλοντα. Επίσης, παρέχει τον κώδικα και τα δεδομένα που απαιτούνται για την αναπαραγωγή των πειραμάτων MPI για ¼-socket που διεξήχθησαν στο πλαίσιο της παρούσας εργασίας. Η διάθεση του έργου διευκολύνει την ανεξάρτητη επαλήθευση των αποτελεσμάτων, την τροποποίηση και την πιθανή συλλογική ανάπτυξη του εργαλείου. Επιπροσθέτως, η πλήρης υλοποίηση του SiS, όπως παρουσιάστηκε κατά τον Οκτώβρη του 2025, έχει αρχειοθετηθεί στο Παράρτημα Α.

7.4.1.2 Εξαρτήσεις Κώδικα

Μια σημαντική σχεδιαστική απόφαση αφορά τις εξωτερικές εξαρτήσεις βιβλιοθηκών. Το SiS δεν επιβάλλει πρόσθετες απαιτήσεις λογισμικού πέραν όσων είναι ήδη διαθέσιμες στην υπάρχουσα υποδομή. Αντιθέτως, αξιοποιεί τις παρεχόμενες από το σύστημα βιβλιοθήκες και modules για την επίλυση τυχόν εξαρτήσεων κατά τη μεταγλώττιση ή την εκτέλεση. Αυτή η προσέγγιση ελαχιστοποιεί τον διαχειριστικό φόρτο και ενισχύει τη φορητότητα του SiS, αν και ενδέχεται να απαιτείται ενδελεχής παρακολούθηση για πιθανές συγκρούσεις μεταξύ της υποκείμενης εγκατάστασης Slurm του συστήματος και του SiS.

7.4.1.3 Διαδικασία Εγκατάστασης

Η διαδικασία εγκατάστασης του SiS περιλαμβάνει τη ρύθμιση του περιβάλλοντος και την εκτέλεση συγκεκριμένων scripts για τη μεταφόρτωση, μεταγλώττιση και εγκατάσταση του εργαλείου, είτε από το διαδικτυακό αποθετήριο είτε από τον κώδικα που περιλαμβάνεται στο Παράρτημα της παρούσας εργασίας. Η προεπιλεγμένη δομή φακέλων του έργου παρουσιάζεται στο Σχ. 7.7.

```
.
├── bash
│   ├── aris_boilerplate.sh
│   ├── environment_variables.sh
│   ├── helpers.sh
│   ├── job_queue.txt
│   ├── epilogue.sh
│   ├── prologue.sh
│   ├── v-slurm-lite.sh
│   └── unit_tests
└── quarter-socket-mpi
    ├── data
    ├── find_max_min.sh
    ├── maxnodes.sh
    ├── run_compact.sh
    ├── run_cos.NAS_NAS.sh
    ├── submit_cmp.sh
    └── submit_co.sh
```

Σχήμα 7.7: Ιεραρχία φακέλων του εργαλείου SiS.

Ο κύριος κατάλογος που περιλαμβάνει τον εκτελέσιμο κώδικα είναι ο `bash`, ενώ το βασικό script εγκατάστασης είναι το `aris_boilerplate.sh`, το οποίο εκτελεί τις βασικές λειτουργίες μεταφόρτωσης του πηγαίου κώδικα του Slurm και δημιουργίας των εκτελέσιμων στον στόχο σύστημα.

Κρίσιμο ρόλο στη διαδικασία αυτή κατέχει το αρχείο `environment_variables.sh`, το οποίο ορίζει τις μεταβλητές περιβάλλοντος και τις διαδρομές που χρησιμοποιούνται από τα υπόλοιπα scripts κατά την εγκατάσταση και εκτέλεση του SiS. Ένα ενδεικτικό παράδειγμα του αρχείου αυτού παρατίθεται στον Κώδ. 5.1, όπου παρουσιάζονται οι μεταβλητές και οι αντίστοιχες διαδρομές που απαιτούνται για την ομαλή λειτουργία του συστήματος.

Η δομή του script εγκατάστασης διαιρείται σε επιμέρους ενότητες:

- Η πρώτη ενότητα καθορίζει την έκδοση του Slurm που θα εγκατασταθεί. Ως τον Οκτώβριο 2025, η έκδοση `16.05.11` είναι η πλήρως λειτουργική, ενώ οι νεότερες εκδόσεις (21 και 25) χρησιμοποιούνται για δοκιμές.

- Η δεύτερη ενότητα καθορίζει τις διαδρομές στο σύστημα αρχείων, καθώς και τις θέσεις

αποθήκευσης των εκτελέσιμων και των αρχείων ρυθμίσεων.

- Η τρίτη ενότητα αφορά τις παραμέτρους υλικού του υποκείμενου συστήματος, οι οποίες είναι απαραίτητες για την ορθή προσαρμογή της διαδικασίας μεταγλώττισης και εκτέλεσης σε κάθε συγκεκριμένο HPC περιβάλλον.
- Τέλος, η τελευταία ενότητα περιγράφει τις διαδρομές και τις παραμέτρους της εργασίας `sbatch` του εξωτερικού Slurm.

Το script `aris_boilerplate.sh` αυτοματοποιεί ολόκληρη τη διαδικασία προετοιμασίας, μεταγλώττισης και εγκατάστασης εκδόσεων του Slurm σε επίπεδο χρήστη. Συγκεκριμένα:

1. Εισάγει τις απαραίτητες ρυθμίσεις περιβάλλοντος και βοηθητικά scripts.
2. Εντοπίζει την έκδοση του Slurm που θα εγκατασταθεί και εκτελεί τη διαδικασία μεταγλώττισης.
3. Δημιουργεί τη δομή φακέλων, φορτώνει τα απαιτούμενα modules, και παράγει τα SSL κλειδιά (για τις εκδόσεις που αυτό είναι απαραίτητο, όπως η έκδοση 16.05) για την επικοινωνία μεταξύ κόμβων.
4. Τέλος, ανακτά το αρχείο πηγαίου κώδικα του Slurm από το επίσημο αποθετήριο της διανομής του, το μεταγλωττίζει και εγκαθιστά τα εκτελέσιμα στο χώρο του χρήστη.

7.4.1.4 Εκτέλεση του SiS

Το script `v-slurm-lite.sh` αποτελεί τον πυρήνα του SiS για την ανάπτυξή του με χρήση της υπάρχουσας κατανομής πόρων του εξωτερικού Slurm.

Για να αποφευχθεί η ανάγκη επεξεργασίας πολλών αρχείων, χρησιμοποιείται ο βοηθητικός Κώδ.A.4, ο οποίος λειτουργεί ως ενδιάμεσος μεταξύ του αρχείου ρυθμίσεων `environment_variables.sh` και του Κώδ.A.5.

Σε γενικές γραμμές, η εκτέλεση του `v-slurm-lite.sh` ακολουθεί μια σειρά από διακριτά στάδια:

1. Εκτελείται αρχικά το `prologue.sh`, το οποίο λειτουργεί ως σημείο αρχικοποίησης, όπου ο χρήστης μπορεί να φορτώσει απαραίτητες βιβλιοθήκες ή να ρυθμίσει το περιβάλλον εκτέλεσης.
2. Στη συνέχεια, φορτώνονται οι μεταβλητές περιβάλλοντος (βλ. Κώδ.5.1) και οι βοηθητικές συναρτήσεις από του Κώδ.A.6, που ορίζουν διαδρομές και υλοποιούν χρηστικές λειτουργίες για τη ρύθμιση του SiS.
3. Έπειτα ανιχνεύει τον αριθμό κόμβων που έχουν εκχωρηθεί από το εξωτερικό Slurm και επιλέγει έναν κόμβο ως ελεγκτή.
4. Δημιουργείται προσωρινό αρχείο ρυθμίσεων `slurm.conf` για το SiS, το οποίο περιγράφει τον ελεγκτή (`slurmctld`), τους υπολογιστικούς κόμβους, τις παραμέτρους πόρων τις πληροφορίες διαμέρισης κ.ά. Η δομή του αρχείου προσαρμόζεται ανάλογα με την έκδοση του Slurm, εξασφαλίζοντας συμβατότητα με διαφορετικές εκδόσεις.
5. Εκκινείται ο ελεγκτής `slurmctld` και οι δαίμονες `slurmd` στους υπόλοιπους κόμβους, μέσω εντολών `srun` που καθορίζονται από την κατανομή κόμβων, το πλήθος των εργασιών και την αντιστοίχιση CPU.

Αφού ολοκληρωθεί η εκκίνηση, γίνεται έλεγχος λειτουργικότητας με την εντολή `scontrol show nodes`. Έπειτα, το σύστημα εισέρχεται στη φάση ορχήστρωσης, κατά την οποία παρακολουθεί την ουρά του cluster και εκτελεί εργασίες που καθορίζονται στο αρχείο λίστας εργασιών (βλ. Κώδ. 5.2).

Το εργαλείο διασφαλίζει ότι όλες οι εργασίες εκτελούνται και ελέγχει περιοδικά για τυχόν ενεργές διεργασίες. Σε περίπτωση που παραμείνουν ενεργές εργασίες όταν εξαντληθεί το όριο χρόνου αυτές ακυρώνονται για να επιτευχθεί τερματισμός.

Με την ολοκλήρωση των εργασιών, εκτελείται το `epilogue.sh`, το οποίο αποτελεί το τελικό στάδιο καθαρισμού, συλλογής αποτελεσμάτων ή μετα-επεξεργασίας. Η χρήση ξεχωριστών `prologue` και `epilogue` scripts προσδίδει ευελιξία στο SiS.

Συνοπτικά, το `v-slurm-lite.sh` λειτουργεί ως δυναμικό επίπεδο ορχήστρωσης που αυτοματοποιεί την προετοιμασία περιβάλλοντος, τη δημιουργία ρυθμίσεων, την εκκίνηση των υπηρεσιών, την υποβολή εργασιών και την τελική απελευθέρωση πόρων. Η δυνατότητα συνδυασμού προσαρμοσμένων σταδίων εκτέλεσης με παραμετροποιήσιμη ρύθμιση καθιστά το SiS ένα ιδιαίτερα προσβάσιμο περιβάλλον δοκιμών για διάφορους τύπους εργασιών στο πλαίσιο του Slurm.

7.5 Αποτελέσματα, Αξιολόγηση, Μελλοντική Δουλειά

7.5.1 Επικύρωση Ορθότητας Λειτουργίας SiS

Η επικύρωση του SiS πραγματοποιήθηκε μέσω μιας σειράς δοκιμών που είχαν ως στόχο να επαληθεύσουν την ορθότητα εκτέλεσης ως πλήρως λειτουργικού διαχειριστή πόρων, καθώς και την ορθή αλληλεπίδρασή του με τα διάφορα εργαλεία και εντολές Slurm.

Λόγω των υφιστάμενων περιορισμών της τρέχουσας έκδοσης του SiS, δεν ήταν εφικτή η εκτέλεση πλήρους κλίμακας MPI εφαρμογών. Ως εκ τούτου, η διαδικασία επικύρωσης επικεντρώθηκε κυρίως στην επιβεβαίωση ότι το SiS ερμηνεύει, προωθεί και εκτελεί ορθά το σύνολο εντολών του Slurm.

Κατά τη διάρκεια των δοκιμών, εκτελέστηκαν λειτουργίες του Slurm απευθείας μέσω του SiS, όπως εντολές υποβολής εργασιών, παρακολούθησης κατάστασης και παρακολούθησης του συστήματος. Σε όλες τις περιστάσεις, το εργαλείο επέδειξε την αναμενόμενη συμπεριφορά, αναπαράγοντας με ακρίβεια τις αποκρίσεις ενός εγγενώς εγκατεστημένου Slurm. Η συνέπεια αυτή αποδεικνύει ότι οι εσωτερικοί μηχανισμοί του SiS — όπως η ανάλυση των αρχείων ρυθμίσεων, η μετάφραση εντολών και η επικοινωνία με τους δαίμονες, λειτουργούν όπως προβλέπεται.

Στη συνέχεια, παρουσιάζονται τα αποτελέσματα εκτέλεσης ενδεικτικών εντολών Slurm, προκειμένου να επιδειχθεί η ορθή λειτουργία του SiS. Ο Κώδ. 7.1 δείχνει την έξοδο της εντολής `sinfo` από το SiS για λειτουργία σε τρεις κόμβους (το αποτέλεσμα έχει περικοπεί για λόγους σαφήνειας):

Κώδικας 7.1: Εκτέλεση της εντολής `sinfo` μέσω του SiS σε 3 κόμβους.

```
NODELIST   NODES PARTITION     STATE CPUS      S:C:T MEMORY
node066        1 compute*     mixed   20    2:10:1  57344
node413        1 compute*     mixed   20    2:10:1  57344
node414        1 compute*     mixed   20    2:10:1  57344
```

Αντίστοιχα, η επόμενη εντολή `sinfo` επιδεικνύει τη δυνατότητα του SiS να κλιμακώνεται σε μεγαλύτερο αριθμό κόμβων, όπως φαίνεται στον Κώδ. 7.2 (επίσης περικομμένο αποτέλεσμα για λόγους σαφήνειας):

Κώδικας 7.2: Εκτέλεση της εντολής `sinfo` μέσω του SiS σε 10 κόμβους, αποδεικνύοντας τη δυνατότητα κλιμάκωσης του εργαλείου.

```
NODELIST    NODES PARTITION     STATE CPUS      S:C:T MEMORY
node323         1 compute*      mixed   20     2:10:1  57344
node324         1 compute*      mixed   20     2:10:1  57344
node325         1 compute*      mixed   20     2:10:1  57344
node326         1 compute*      mixed   20     2:10:1  57344
node361         1 compute*      mixed   20     2:10:1  57344
node362         1 compute*      mixed   20     2:10:1  57344
node363         1 compute*      mixed   20     2:10:1  57344
node364         1 compute*      mixed   20     2:10:1  57344
node365         1 compute*      mixed   20     2:10:1  57344
node366         1 compute*      mixed   20     2:10:1  57344
```

Για την ίδια εκτέλεση, παρουσιάζεται η δυνατότητα παρακολούθησης της ουράς εργασιών μέσω της εντολής `squeue` από τα εκτελέσιμα του SiS:

Κώδικας 7.3: Εκτέλεση της εντολής `squeue` μέσω του SiS κατά τη διάρκεια εργασίας που έχει δεσμεύσει 10 κόμβους.

```
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(
    ↪ REASON)
2     compute     Sinfo-10 goumas  R       0:01     10     node[323-326,
                                                           361-366]
```

Τέλος, πραγματοποιήθηκε μια συστηματική διαδικασία επικύρωσης, κατά την οποία εκτελέστηκαν όλες οι βασικές εντολές του Slurm εντός του περιβάλλοντος του SiS, προκειμένου να αξιολογηθεί η λειτουργική ορθότητα και η συμβατότητά του με ένα παραγωγικό σύστημα.

7.5.2 Πειραματικά Αποτελέσματα ¼-socket

Όταν ολοκληρωθεί η δυνατότητα εκτέλεσης MPI εργασιών από το SiS, θα χρειαστεί μία πειραματική βάση για τα πειράματα συνεκτέλεσης διαφορετικών εφαρμογών σε κοινούς κόμβους με σκοπό τη χρήση των αποτελεσμάτων για την αποτίμηση του αντικτύπου της συνεκτέλεσης στα διάφορα σχήματα χρονοδρομολόγησης.

Κάθε κόμβος του συστήματος ARIS διαθέτει δύο sockets με δέκα πυρήνες το καθένα (σύνολο 20 πυρήνες), όπου οι πυρήνες 0 εως 9 αντιστοιχούν στο $1^{ο}$ socket ενώ οι πυρήνες 10 εως 19 στο $2^{ο}$ socket. Για τα πειράματα συνεκτέλεσης τεσσάρων NAS προγραμμάτων, κάθε εφαρμογή αντιστοιχίστηκε σε συγκεκριμένο υποσύνολο πυρήνων, όπως φαίνεται παρακάτω:

- **Πρόγραμμα A:** πυρήνες 0–2, 10–11
- **Πρόγραμμα B:** πυρήνες 3–4, 12–14
- **Πρόγραμμα C:** πυρήνες 5–7, 15–16
- **Πρόγραμμα D:** πυρήνες 8–9, 17–19

Η τοποθέτηση αυτή εξασφαλίζει συμμετρική χρήση πόρων, σε επίπεδο κόμβου, και ελέγξιμη συνθήκη συνεκτέλεσης. Ο κώδικας για τη συνεκτέλεση των NAS benchmarks κατά ¼-socket δίνεται στο Παράρτημα Β.

Τα πειράματα βασίστηκαν στη σουίτα `NPB3.4.3 MPI` με προβλήματα κλάσης D σε 64 νήματα. Τα αποτελέσματα συγκρίθηκαν με τις «compact» εκτελέσεις, δηλαδή την εκτέλεση κάθε προγράμματος μόνου του σε κάθε κόμβο. Η *επιτάχυνση*, ή *speedup*, ορίζεται από την Εξ. 7.1:

$$\text{speedup} = \frac{\text{time}_{\text{compact}}}{\text{time}_{\text{quarter}}} \tag{7.1}$$

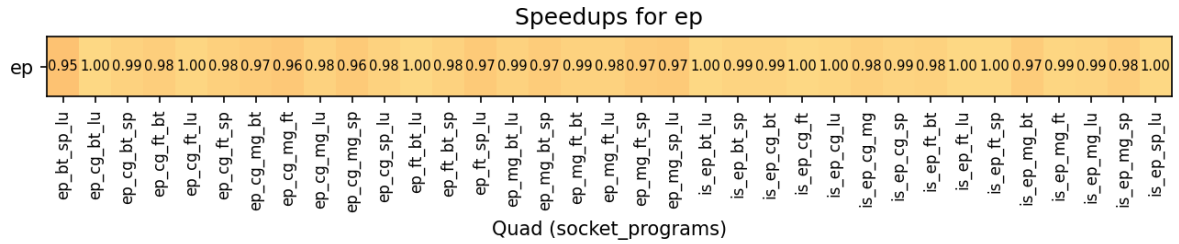
όπου: $\text{time}_{\text{compact}}$ είναι ο μέσος χρόνος εκτέλεσης χωρίς συνεκτέλεση, και $\text{time}_{\text{quarter}}$ ο αντίστοιχος χρόνος με συνεκτέλεση τεσσάρων εφαρμογών στον ίδιο κόμβο.

Τα πρώτα αποτελέσματα δείχνουν ότι η διάταξη αυτή επιτρέπει δίκαιη αξιολόγηση της αλληλεπίδρασης των εργασιών. Τα δεδομένα αυτά αποτελούν τη βάση για μελλοντική ενσωμάτωση δυναμικών μηχανισμών συνεκτέλεσης στο SiS.

Πιο συγκεκριμένα, τα αποτελέσματα για κάθε ένα από τα προγράμματα NAS, παρουσιάζεται στις επόμενες υποενότητες.

7.5.2.1 EP – Embarrassingly Parallel

Το **EP** δημιουργεί ανεξάρτητους τυχαίους αριθμούς χωρίς επικοινωνία διεργασιών, αξιολογώντας καθαρά την υπολογιστική ισχύ.

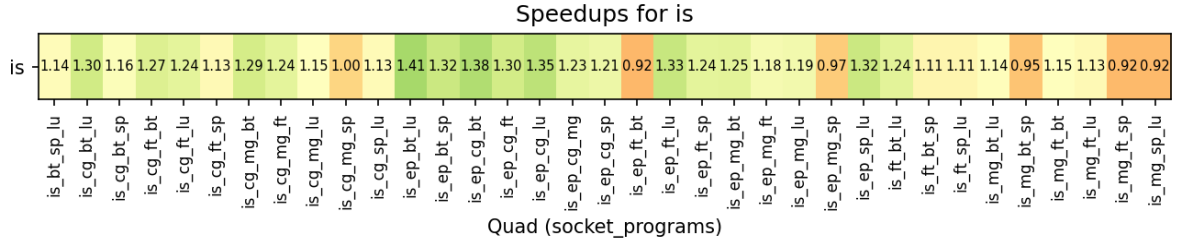


Σχήμα 7.8: Speedup του προγράμματος EP με συνεκτέλεση σε περιβάλλον ¼-socket για προβλήματα κλάσης D με 64 διεργασίες/νήματα.

Καμία ουσιαστική διαφοροποίηση δεν παρατηρήθηκε, όπως αναμενόταν. Μέσος όρος speedup: 0.98.

7.5.2.2 IS – Integer Sort

Το **IS** ταξινομεί ακολουθίες κλειδιών μέσω bucket sort, δοκιμάζοντας τη συνοχή και την επικοινωνία μεταξύ διεργασιών.

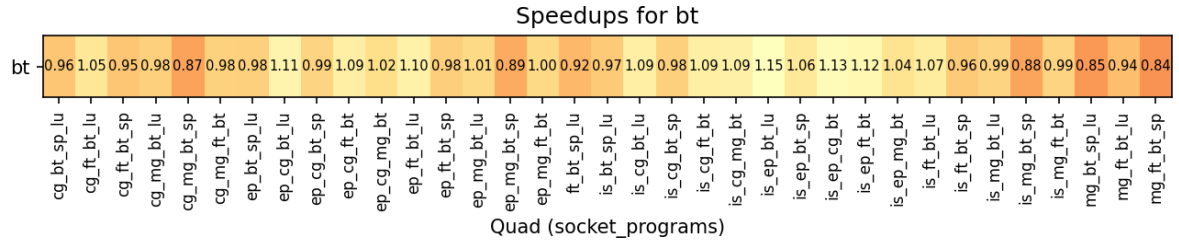


Σχήμα 7.9: Speedup του προγράμματος IS με συνεκτέλεση σε περιβάλλον ¼-socket για προβλήματα κλάσης D με 64 διεργασίες/νήματα.

Παρουσιάστηκε σαφής βελτίωση ταχύτητας σε όλες τις περιπτώσεις. Μέσος όρος speedup: 1.18.

7.5.2.3 BT – Block Tridiagonal Solver

Το **BT** λύνει μη γραμμικές διαφορικές εξισώσεις μέσω block tridiagonal αλγορίθμου, απαιτώντας υψηλή υπολογιστική ισχύ.

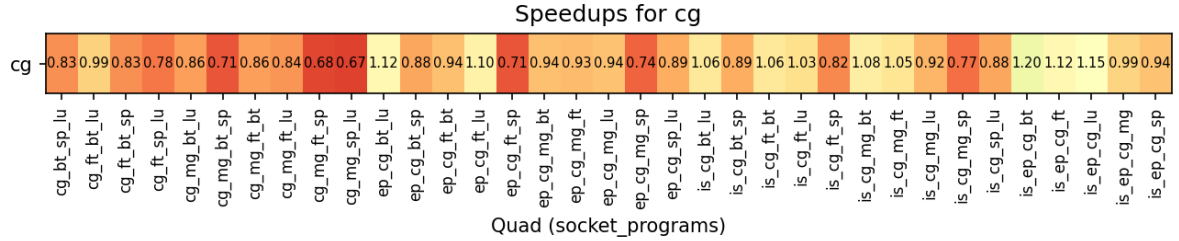


Σχήμα 7.10: Speedup του προγράμματος BT με συνεκτέλεση σε περιβάλλον ¼-socket για προβλήματα κλάσης D με 64 διεργασίες/νήματα.

Παρατηρήθηκε μεγάλη διακύμανση. Μέσος όρος speedup: 1.00.

7.5.2.4 CG – Conjugate Gradient

Το **CG** υπολογίζει ιδιοτιμές αραιών συμμετρικών πινάκων, δοκιμάζοντας μη ντετιρμινιστικά μοτίβα επικοινωνιών.

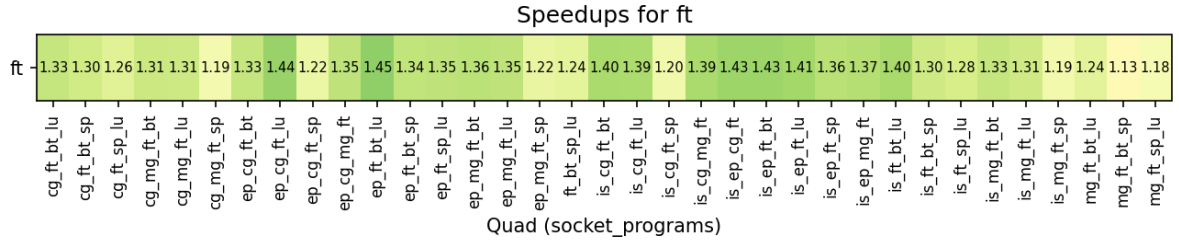


Σχήμα 7.11: Speedup του προγράμματος CG με συνεκτέλεση σε περιβάλλον ¼-socket για προβλήματα κλάσης D με 64 διεργασίες/νήματα.

Παρατηρήθηκε σαφής επιβράδυνση κατά τη συνεκτέλεση. Μέσος όρος speedup: 0.91.

7.5.2.5 FT – Fast Fourier Transform

Το **FT** εκτελεί τρισδιάστατους μετασχηματισμούς Fourier, αποτελώντας αυστηρό δείκτη απόδοσης επικοινωνιών μεγάλων αποστάσεων.

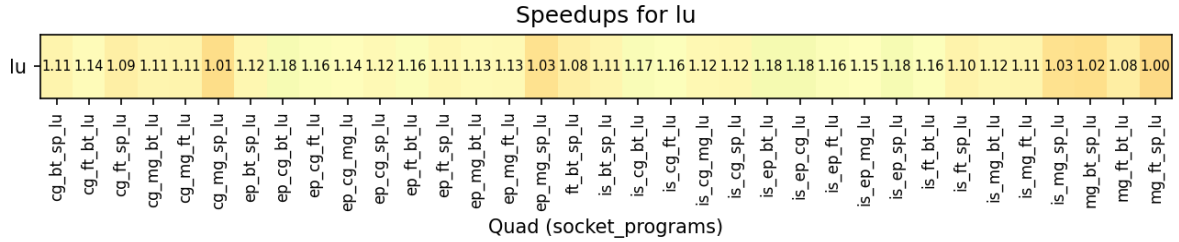


Σχήμα 7.12: Speedup του προγράμματος FT με συνεκτέλεση σε περιβάλλον ¼-socket για προβλήματα κλάσης D με 64 διεργασίες/νήματα.

Παρατηρήθηκε αξιοσημείωτη βελτίωση χρόνου εκτέλεσης. Μέσος όρος speedup: 1.32.

7.5.2.6 LU – Lower Upper Symmetric Gauss-Seidel

Το **LU** επιλύει εξισώσεις Navier–Stokes μέσω προσεγγιστικής LU παραγοντοποίησης, συνδυάζοντας έντονη ανάγκη για υπολογιστική ισχύ αλλά και επικοινωνίες μεταξύ διεργασιών.

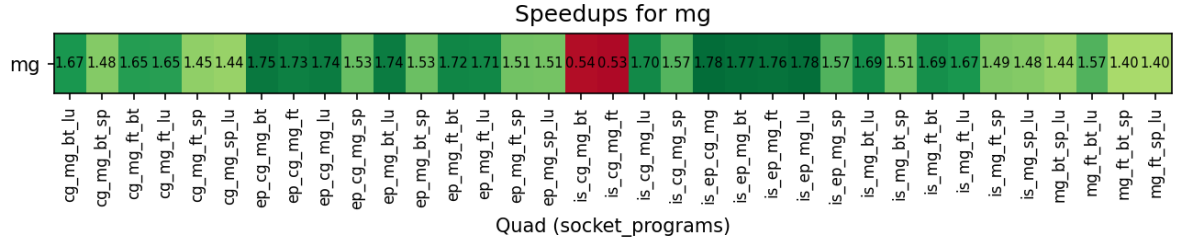


Σχήμα 7.13: Speedup του προγράμματος LU με συνεκτέλεση σε περιβάλλον ¼-socket για προβλήματα κλάσης D με 64 διεργασίες/νήματα.

Παρουσιάζει θετική βελτίωση σε όλες τις δοκιμές. Μέσος όρος speedup: 1.12.

7.5.2.7 MG – MultiGrid

Το **MG** επιλύει την τριδιάστατη εξίσωση Poisson με τη μέθοδο πολυπλεγμάτων, απαιτώντας επικοινωνία μεταξύ κόμβων.

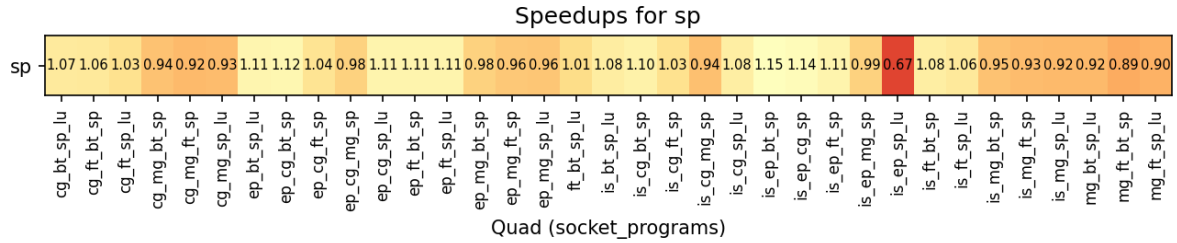


Σχήμα 7.14: Speedup του προγράμματος MG συνεκτέλεση σε περιβάλλον ¼-socket για προβλήματα κλάσης D με 64 διεργασίες/νήματα.

Παρουσιάζει τη μεγαλύτερη επιτάχυνση από όλα τα benchmarks. Μέσος όρος speedup: 1.55.

7.5.2.8 SP – Scalar Pentadiagonal

Το **SP** λύνει εξισώσεις Navier–Stokes μέσω penta-diagonal συστημάτων/πινάκων.

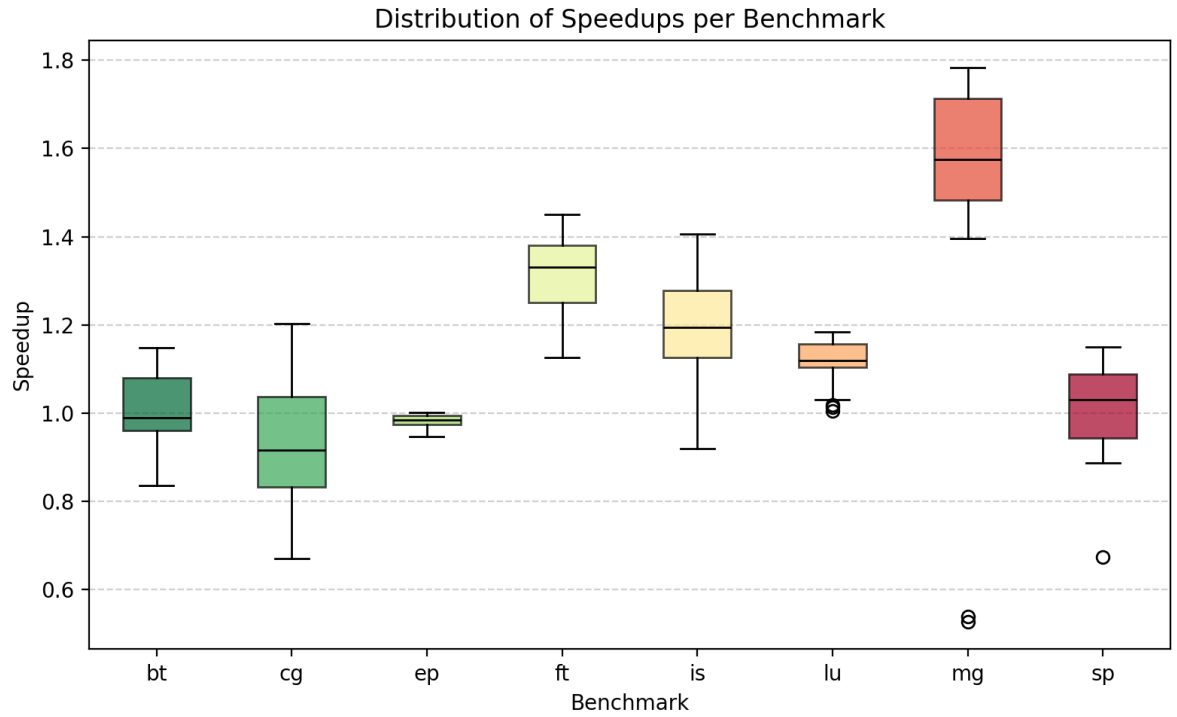


Σχήμα 7.15: Speedup του προγράμματος SP σε συνεκτέλεση σε περιβάλλον ¼-socket για προβλήματα κλάσης D με 64 διεργασίες/νήματα.

Αθροιστικά, δεν παρουσιάζει ουσιαστική αλλαγή στις επιδόσεις του. Μέσος όρος speedup: 1.01.

7.5.3 Συγκεντρωτικά Μεγέθη

Για τη συνολική προβολή των επιπτώσεων της συνεκτέλεσης, η μεταβλητή *speedup* απεικονίζεται στο Σχ. 7.16 ως boxplot, όπου για κάθε ένα benchmark απεικονίζεται η μέση τιμή, η διασπορά καθώς και τα εξωκείμενα σημεία του *speedup*

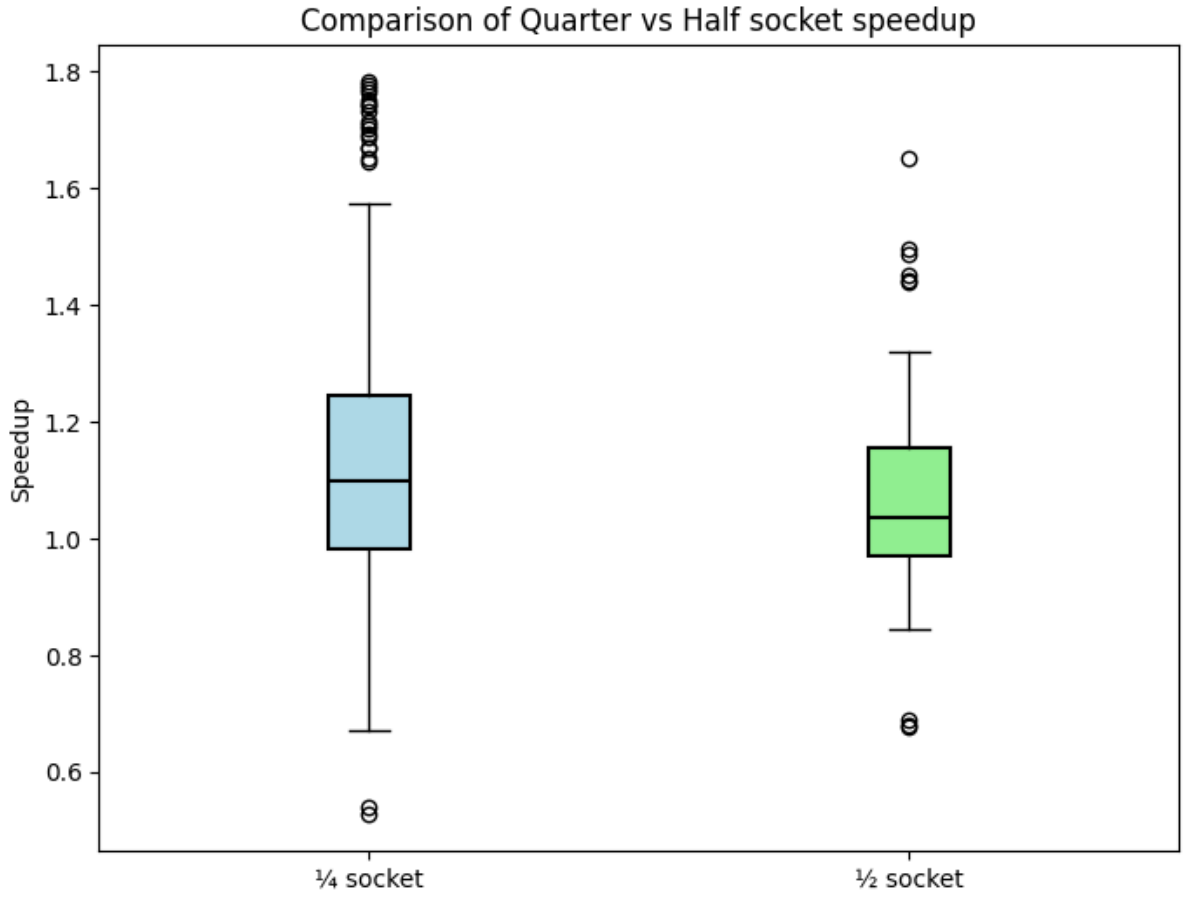


Σχήμα 7.16: Boxplot των NAS benchmarks ως προς το σχετικό τους speedup σε συνεκτέλεση ¼-socket.

Τα αποτελέσματα δείχνουν σημαντική ετερογένεια στη συμπεριφορά κάθε benchmark. Τα `MG` και `FT` εμφανίζουν τα υψηλότερα μέσα speedups (έως 1.6). Αντίθετα, τα `CG` και `SP`

παρουσιάζουν υψηλή μεταβλητότητα και σε ορισμένες περιπτώσεις επιβράδυνση (< 1.0), δείχνοντας τον υψηλό βαθμό εξάρτησης από τα χαρακτηριστικά των συνεκτελούμενων εφαρμογών. Το EP διατηρεί *speedup* κοντά στη μονάδα, ενώ τα LU και IS δείχνουν σταθερή βελτίωση με περιορισμένες μεταβολές. Συνολικά, οι επιδράσεις της συνεκτέλεσης εξαρτώνται έντονα από τον τύπο του benchmark.

Στο Σχ.7.17 καταγράφεται η κατανομή των speedups για τις συνθήκες συνεκτέλεσης κατά ¼-socket και κατά ½-socket.



Σχήμα 7.17: Το boxplot των speedups της εκτέλεσης κατά ¼-socket, εν αντιθέσει με τα speedups κατά την εκτέλεση κατά ½-socket.

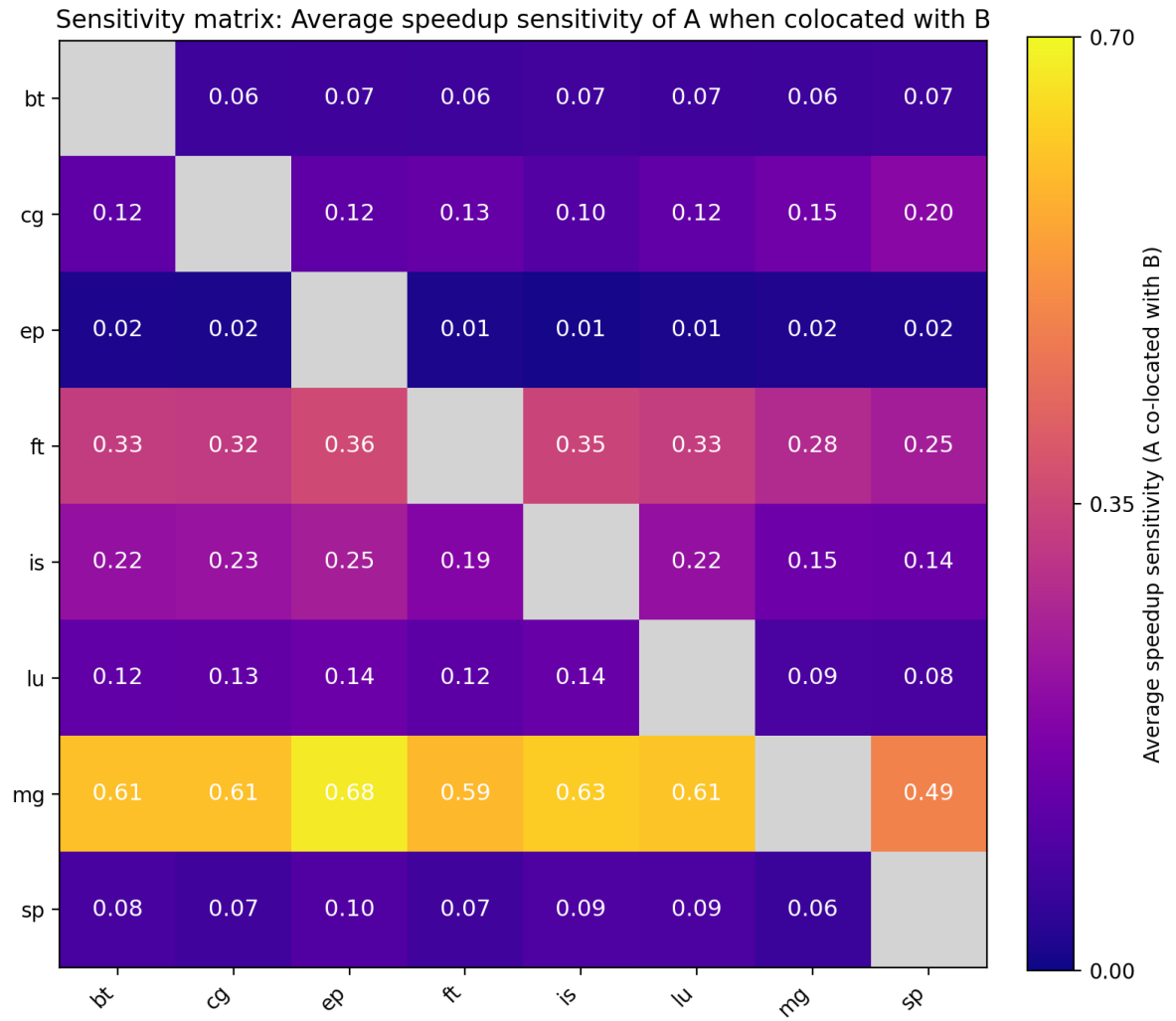
Η διάμεσος για την εκτέλεση κατά ¼-socket υπολογίστηκε σε 1.09 εν συγκρίσει με 1.03 για την εκτέλεση κατά ½-socket.

Ένας δεύτερος δείκτης, η *ευαισθησία (sensitivity)*, εκφράζει το απόλυτο μέγεθος μεταβολής στην απόδοση (speedup ή slowdown) λόγω συνεκτέλεσης, κανονικοποιημένο ως προς τον χρόνο αναφοράς.

$$S_{b,q} = \frac{1}{|\{b,q\} \in Q_b|} \sum_{\{b,q\} \in Q_b} \left| 1 - \text{speedup}_{b,q} \right| \tag{7.2}$$

όπου $S_{b,q}$ είναι η ευαισθησία του benchmark b ως προς το q, Q_b το σύνολο όλων των εκτελέσεων ¼-socket που περιλαμβάνουν το benchmark b, και $\text{speedup}_{q,b}$ το μετρημένο speedup του b στη q.

Η συνολική εκτίμηση της ευαισθησίας φαίνεται στο Σχ. 7.18.



Σχήμα 7.18: Η μετρική ευαισθησίας (sensitivity) κατά τη συνεκτέλεση δύο NAS benchmarks σε ¼-socket περιβάλλον.

Τα αποτελέσματα δείχνουν υψηλή ευαισθησία για το `MG` (0.49–0.68) και το `FT` (0.25–0.36), δηλώνοντας μεγάλη επίδραση από τη συνεκτέλεση. Το `EP` παρουσιάζει μικρή εως καθόλου ευαισθησία (< 0.02), επιβεβαιώνοντας την ανεξαρτησία του. Τα `LU`, `IS` και `BT` βρίσκονται στη μέση κλίμακα (< 0.2). Συνολικά, ο πίνακας ευαισθησίας αναδεικνύει τις διαφοροποιήσεις μεταξύ benchmarks: ορισμένα benchmarks (π.χ. `MG`, `FT`) είναι ιδιαίτερα "ευαίσθητα", ενώ άλλα (π.χ. `EP`, `SP`) σταθερά σε συνθήκες συνεκτέλεσης.

7.5.4 Συμπεράσματα

Η παρούσα διπλωματική εργασία παρουσίασε τον σχεδιασμό, την υλοποίηση και την επικύρωση του εργαλείου *SiS* (Slurm-in-Slurm), μιας συνεισφοράς στη διαχείριση πόρων υπολογιστικών συστημάτων υψηλών επιδόσεων. Κύριος στόχος υπήρξε η ανάπτυξη ενός εργαλείου σε επίπεδο χρήστη, ικανού να αναπτύσσει και να διαχειρίζεται εργασίες στο

πλαίσιο του Slurm εντός παραγωγικών περιβαλλόντων HPC, επιτρέποντας μελλοντικά την ενσωμάτωση νέων λειτουργικοτήτων και τον πειραματισμό με νέες πολιτικές χρονοδρομολόγησης και αλγορίθμων χωρίς ανάγκη διαχειριστικών προνομίων.

Τα αποτελέσματα έδειξαν ότι το SiS επιτυγχάνει πλήρως αυτόν τον στόχο. Επιπλέον, το SiS υποστηρίζει την ανάπτυξη προσαρμοσμένων ή εκ νέου μεταγλωττισμένων εκδόσεων του Slurm, υπερβαίνοντας τους περιορισμούς των plugin APIs. Έτσι, δημιουργεί μια επεκτάσιμη βάση για μελλοντική ενσωμάτωση σύγχρονων ερευνητικών προσεγγίσεων στη διαχείριση πόρων HPC.

Πέρα από την ανάπτυξη του εργαλείου, η διπλωματική εργασία παρείχε και σημαντικά πειραματικά ευρήματα σχετικά με τη συνεκτέλεση εφαρμογών. Μέσω δοκιμών στο σύστημα ARIS, διερευνήθηκαν οι επιπτώσεις της συνεκτέλεσης διαφορετικών NAS Parallel Benchmarks σε εκτελέσεις ¼-socket. Τα αποτελέσματα, ανέδειξαν τη σύνθετη αλληλεπίδραση διαφορετικών φορτίων σε περιβάλλον διαμοιραζόμενων πόρων.

7.5.5 Μελλοντική Έρευνα

Αν και το SiS παρέχει λειτουργική και επεκτάσιμη βάση για πειραματική διαχείριση φορτίων σε περιβάλλοντα HPC, η εργασία αυτή ανέδειξε πολλαπλές κατευθύνσεις για μελλοντική έρευνα και ανάπτυξη.

Άμεση προτεραιότητα αποτελεί η πλήρης ενσωμάτωση υποστήριξης εκτελέσεων `MPI`. Αν και το τρέχον σύστημα διαχειρίζεται ανεξάρτητες διεργασίες Slurm, η πλήρης υποστήριξη κατανεμημένων `MPI` εφαρμογών θα ενισχύσει ουσιαστικά τη χρησιμότητά του, επιτρέποντας την αξιολόγηση παράλληλων εφαρμογών εντός του απομονωμένου περιβάλλοντος του SiS.

Επιπλέον, προτείνεται η επέκταση της συμβατότητας του SiS με περισσότερες εκδόσεις του Slurm, τόσο νεότερες όσο και παλαιότερες. Αυτό θα διευκολύνει συγκριτικές μελέτες συμπεριφοράς και επιδόσεων μεταξύ διαφορετικών εκδόσεων του Slurm και θα κάνει ακόμα πιο ευέλικτη την ανάπτυξη λογισμικού για τον πηγαίο κώδικα του Slurm.

Σε ερευνητικό επίπεδο, το SiS παρέχει πρόσφορο έδαφος για ανάπτυξη και αξιολόγηση νέων χρονοδρομολογητών που αξιοποιούν στρατηγικές συνεκτέλεσης φορτίων αλλά και άλλες. Με βάση τα αποτελέσματα των πειραμάτων συνεκτέλεσης της παρούσας διπλωματικής εργασίας, μελλοντική έρευνα θα μπορούσε να αναπτύξει αλγορίθμους που βελτιστοποιούν την εκτέλεση εργασιών λαμβάνοντας υπόψη τη συμφόρηση, την τοπικότητα επικοινωνίας και την αλληλεπίδραση των φορτίων. Τέτοιοι δυναμικοί χρονοδρομολογητές θα μπορούσαν να αυξήσουν τη συνολική αξιοποίηση συστήματος μειώνοντας παράλληλα τις αλληλεπιδράσεις παρεμβολής.

Τέλος, μία ακόμη σημαντική κατεύθυνση αφορά το συστηματικό χαρακτηρισμό φορτίων. Μέσω εξαγωγής χαρακτηριστικών βάσει των οποίων οι εργασίες θα μπορούν να ταξινομηθούν σε κατηγορίες εφαρμογών ανάλογες με τα NAS benchmarks. Έτσι, θα καταστεί εφικτή η προγνωστική μοντελοποίηση των επιδράσεων της συνεκτέλεσης και της συνδρομολόγησης, επιτρέποντας στους χρονοδρομολογητές να προβλέπουν ποιες εφαρμογές μπορούν να μοιραστούν αποδοτικά τους ίδιους πόρους και πότε να τις δρομολογούν. Σε συνδυασμό με την ευελιξία του SiS, αυτή η προσέγγιση μπορεί να οδηγήσει σε προσαρμοζόμενες πολιτικές συνεκτέλεσης που μεγιστοποιούν την απόδοση διατηρώντας τη δικαιοσύνη μεταξύ εφαρμογών.

Bibliography

[1] A. Konya and P. Nematzadeh, "Recent applications of ai to environmental disciplines: A review," *Science of The Total Environment*, vol. 906, p. 167 705, 2024, ISSN: 0048-9697. DOI: https://doi.org/10.1016/j.scitotenv.2023.167705. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0048969723063325.

[2] I. Sitdikov *et al.*, *Quantum resources in resource management systems*, 2025. arXiv: 2506.10052 [quant-ph]. [Online]. Available: https://arxiv.org/abs/2506.10052.

[3] J. McCalpin, *Memory bandwidth and system balance in hpc systems*, Nov. 2016. DOI: 10.13140/RG.2.2.18269.54245.

[4] N. P. NAGENDRA, "Improving instruction cache performance for modern processors with growing workloads," English, PhD Thesis, PRINCETON UNIVERSITY, USA, Sep. 2021. [Online]. Available: https://liberty.princeton.edu/Publications/phdthesis_nagendra.pdf.

[5] I. Peng, I. Karlin, M. Gokhale, K. Shoga, M. Legendre, and T. Gamblin, "A holistic view of memory utilization on hpc systems: Current and future trends," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '21, Washington DC, DC, USA: Association for Computing Machinery, 2022, ISBN: 9781450385701. DOI: 10.1145/3488423.3519336. [Online]. Available: https://doi.org/10.1145/3488423.3519336.

[6] F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, "Snitch: A tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads," *IEEE Transactions on Computers (TOC)*, Sep. 2020.

[7] T. Website, *TOP500 Description | TOP500*. [Online]. Available: https://top500.org/project/top500_description/ (visited on 09/03/2025).

[8] F. A. Cornejo, *Maximizing Memory-Bound Applications: How Azure HBv5 Breaks Barriers*, en-US. [Online]. Available: https://www.hpcwire.com/solution_content/microsoft-amd/maximizing-memory-bound-applications-how-azure-hbv5-breaks-barriers/ (visited on 09/03/2025).

[9] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *High Performance Computing for Computational Science – VECPAR 2010*, J. M. L. M. Palma, M. Daydé, O. Marques, and J. C. Lopes, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1–25, ISBN: 978-3-642-19328-6.

[10] W. Schenck, S. El Sayed, M. Foszczynski, W. Homberg, and D. Pleiter, "Evaluation and performance modeling of a burst buffer solution," *SIGOPS Oper. Syst. Rev.*, vol. 50, no. 2, pp. 12–26, Jan. 2017, ISSN: 0163-5980. DOI: 10.1145/3041710.3041714. [Online]. Available: https://doi.org/10.1145/3041710.3041714.

[11] K. O'brien, I. Pietri, R. Reddy, A. Lastovetsky, and R. Sakellariou, "A survey of power and energy predictive models in hpc systems and applications," *ACM Computing Surveys (CSUR)*, vol. 50, no. 3, pp. 1–38, 2017.

[12] E. Suarez *et al.*, "Energy efficiency trends in hpc: What high-energy and astrophysicists need to know," *Frontiers in Physics*, vol. 13, Apr. 2025, ISSN: 2296-424X. DOI: 10.3389/fphy.2025.1542474. [Online]. Available: http://dx.doi.org/10.3389/fphy.2025.1542474.

[13] D. Lustig and M. Martonosi, "Reducing gpu offload latency via fine-grained cpu-gpu synchronization," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 354–365. DOI: 10.1109/HPCA.2013.6522332.

[14] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011. DOI: https://doi.org/10.1002/cpe.1631. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1631. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1631.

[15] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring), Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 483–485, ISBN: 9781450378956. DOI: 10.1145/1465482.1465560. [Online]. Available: https://doi.org/10.1145/1465482.1465560.

[16] S. Contributors, *Slurm Workload Manager - Overview*. [Online]. Available: https://slurm.schedmd.com/overview.html (visited on 09/04/2025).

[17] Wikipedia Contributors, *Slurm Workload Manager*, en, Page Version ID: 1308513329, Aug. 2025. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Slurm_Workload_Manager&oldid=1308513329 (visited on 09/04/2025).

[18] D. H. Ahn *et al.*, "Flux: Overcoming scheduling challenges for exascale workflows," *Future Generation Computer Systems*, vol. 110, pp. 202–213, 2020, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2020.04.006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X19317169.

[19] B. Nitzberg, J. M. Schopf, and J. P. Jones, "PBS Pro: Grid Computing and Scheduling Attributes," in *Grid Resource Management*, F. S. Hillier, J. Nabrzyski, J. M. Schopf, and J. Węglarz, Eds., vol. 64, Boston, MA: Springer US, 2004, pp. 183–190, ISBN: 978-1-4613-5112-2 978-1-4615-0509-9. DOI: 10.1007/978-1-4615-0509-9_13. (visited on 09/06/2025).

[20] C. Cavazzoni, A. Federico, D. Galetti, G. Morelli, and A. Pieretti, "Resource Scheduling Best Practice in Hybrid Clusters," *PRACE online publications*, 2014.

[21] A. Merzky, M. Turilli, M. Titov, A. Al-Saadi, and S. Jha, *RADICAL-Pilot*, Aug. 2025. (visited on 09/06/2025).

[22] *Balsam* Contributors, *Balsam - ALCF User Guides*, https://docs.alcf.anl.gov/polaris/workflows/balsam/. (visited on 09/06/2025).

[23] *Parsl* Contributors, *Parsl: Parallel Scripting in Python*, https://parsl-project.org/. (visited on 09/06/2025).

[24] A. D. Breslow *et al.*, "The case for colocation of high performance computing workloads," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 2, pp. 232–251, 2016, ISSN: 1532-0634. DOI: 10.1002/cpe.3187. (visited on 02/09/2024).

[25] J. Breitbart, J. Weidendorfer, and C. Trinitis, "Case Study on Co-scheduling for HPC Applications," in *2015 44th International Conference on Parallel Processing Workshops*, Sep. 2015, pp. 277–285. DOI: 10.1109/ICPPW.2015.38. (visited on 09/06/2025).

[26] A. d. Blanche and T. Lundqvist, "Node sharing for increased throughput and shorter runtimes - an industrial co-scheduling case study," in *HIPEAC 2018, 3rd COSH Workshop on Co-Scheduling of HPC Applications*, Jan. 2018. DOI: 10.14459/2018md1428535.

[27] F. V. Zacarias, V. Petrucci, R. Nishtala, P. Carpenter, and D. Mossé, "Intelligent colocation of hpc workloads," *Journal of Parallel and Distributed Computing*, vol. 151, pp. 125–137, May 2021, ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2021.02.010. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2021.02.010.

[28] D. Álvarez, K. Sala, and V. Beltran, *Nos-v: Co-executing hpc applications using system-wide task scheduling*, 2022. arXiv: 2204.10768 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2204.10768.

[29] D. Buchaca, J. Marcual, J. L. Berral, and D. Carrera, "Sequence-to-sequence models for workload interference prediction on batch processing datacenters," *Future Generation Computer Systems*, vol. 110, pp. 155–166, Sep. 2020, ISSN: 0167-739X. DOI: 10.1016/j.future.2020.03.058. [Online]. Available: http://dx.doi.org/10.1016/j.future.2020.03.058.

[30] M. Copik, M. Chrapek, L. Schmid, A. Calotoiu, and T. Hoefler, *Software resource disaggregation for hpc with serverless computing*, 2024. arXiv: 2401.10852 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2401.10852.

[31] M. Kellari, "Co-scheduling algorithms for HPC applications," M.S. thesis, Computer Science Laboratory, Electrical and Computer Engineering School, National Technical University of Athens (NTUA), Mar. 2025. (visited on 09/06/2025).

[32] A. d. Blanche and T. Lundqvist, "Terrible twins: A simple scheme to avoid bad co-schedules," in *Proceedings of the 1st COSH Workshop on Co-Scheduling of HPC Applications*, J. Trinitis Carsten ; Weidendorfer, Ed., Jan. 2016, p. 25. DOI: 10.14459/2016md1286952. [Online]. Available: http://wwwi10.lrr.in.tum.de/%7Etrinitic/COSH2016/cosh.html.

[33] A. Blanche and T. Lundqvist, "Disallowing Same-program Co-schedules to Improve Efficiency in Quad-core Servers," in *HIPEAC 2017, 2st COSH Workshop on Co-Scheduling of HPC Applications*, Jan. 2017.

[34] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There goes the neighborhood: Performance degradation due to nearby jobs," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13, Denver, Colorado: Association for Computing Machinery, 2013, ISBN: 9781450323789. DOI: 10.1145/2503210.2503247. [Online]. Available: https://doi.org/10.1145/2503210.2503247.

[35] J. Hall, A. Lathi, D. K. Lowenthal, and T. Patki, "Evaluating the potential of coscheduling on high-performance computing systems," in *Job Scheduling Strategies for Parallel Processing*, D. Klusáček, J. Corbalán, and G. P. Rodrigo, Eds., Cham: Springer Nature Switzerland, 2023, pp. 155–172, ISBN: 978-3-031-43943-8.

[36] N. Triantafyllis, A. Tsoukleidis-Karydakis, E. Karapanagiotis, M. Kellari, G. Goumas, and N. Koziris, "Outlining an HPC co-scheduler," in *International Parallel & Distributed Processing Symposium*, Jun. 2025. DOI: 10.5281/zenodo.15540244.

[37] ARIS' contributors, *ARIS DOCUMENTATION*, https://doc.aris.grnet.gr/. (visited on 09/07/2025).

[38] *CentOS*, https://endoflife.date/centos, Aug. 2025. (visited on 09/13/2025).

[39] *NAS Parallel Benchmarks*, https://www.nas.nasa.gov/software/npb.html. (visited on 10/11/2025).

[40] D. Bailey *et al.*, "The NAS Parallel Benchmarks," *RNR Technical Report*, vol. 94, no. 007, Mar. 1997.

Appendix A

Listing A.1: The ARIS boilerplate script. Its use is to install and compile the preferred Slurm version to be used for SiS

```
#!/bin/bash -l

#!/bin/bash -l
. ./environment_variables.sh
. ./helpers.sh

mkdir --parents $home_dir $base_dir
cd $home_dir
major="${version%%.*}"
generate_slurm_conf $major
if (( major <= 16 )); then
        module purge
    module load gnu/8
    module load intel/18
    module load intelmpi/2018
        if [[ $1 == "--install" || $1 == "-i" ]]; then

            mkdir --parents ${home_dir%/}{/sbin,/lib/slurm,/
               ↪ etc,/spool/{slurm,slurmd},/sys,/sys/{fs,fs/
               ↪ cgroup},/var/{run,log,log/slurm,slurm,spool/{
               ↪ slurm,slurmd}},/spool/slurmd}
            touch ${home_dir%/}/var/run/slurmctld.pid ${
               ↪ home_dir%/}/var/spool/slurm/{node_state,
               ↪ job_state,trigger_state}
            chmod 755 -R ${home_dir%/}
            # Create certificates
            openssl req -x509 -newkey rsa:4096 -passout pass:
               ↪ $password -keyout $key_name -out $cert_name -
               ↪ sha256 -days 365
            # Store them in $home_dir/etc/
            cp $cert_name $key_name ${home_dir%/}/etc/

            # Copy slurm.conf.template and cgroup.conf to
               ↪ $home_dir/etc/
            cp {${base_dir%/}/slurm.conf.template,${base_dir
               ↪ %/}/cgroup.conf} ${home_dir%/}/etc
```

```
        cd ${home_dir%/}/etc/
        openssl genpkey -algorithm RSA -out $key_name -
            ↪ pkeyopt rsa_keygen_bits:2048
        openssl rsa -pubout -in $key_name -out $pkey_name
        cd ${home_dir%/}
        wget https://github.com/SchedMD/slurm/archive/refs
            ↪ /tags/slurm-16-05-11-1.zip
        unzip *slurm*.zip
        cp *slurm*.zip ${base_dir%/}
        rm -f *slurm*.zip
        cd ${home_dir%/}/slurm*/
        ./configure --prefix=${home_dir%/} --exec-prefix=$
            ↪ {home_dir%/} --sysconfdir=${home_dir%/}/etc
            ↪ --localstatedir=${home_dir%/}/var --with-ssl
            ↪ =/usr/lib64/openssl #/engines/lib
        # if the script is in unfinished state then you
            ↪ have to manually create and copy the slurm.
            ↪ conf file
        cd ${home_dir%/}/slurm*/src/
        make -j8

    elif [[ $1 == "--configure" || $1 == "-c" ]]; then
        # Remove old installation
        rm -rf ${home_dir%/}/*
        # Copy source from home_dir to installation
            ↪ directory
        cp ${base_dir%/}/slurm*.zip $home_dir
        # Unzip the source code
        unzip *slurm*.zip
        # Create directories
        mkdir --parents ${home_dir%/}/{lib/slurm,etc,spool
            ↪ /{slurm,slurmd},sys,sys/{fs,fs/cgroup},var/{
            ↪ run,log,log/slurm,slurm,spool/slurm},spool/
            ↪ slurmd}
        touch ${home_dir%/}/var/run/slurmctld.pid ${
            ↪ home_dir%/}/var/spool/slurm/{node_state,
            ↪ job_state,trigger_state} ${home_dir%/}/var/
            ↪ slurm/slurmd.pid
        # Store keys in $home_dir/etc/
        cp ${base_dir%/}/{$cert_name,$key_name} ${home_dir
            ↪ %/}/etc/

        # Copy slurm.conf.template and cgroup.conf to
            ↪ $home_dir/etc/
        cp ${base_dir%/}/slurm.conf.template.$version ${
            ↪ base_dir%/}/cgroup.conf ${home_dir%/}/etc/
            ↪ slurm.conf.template
```

```
        cd ${home_dir%/}/etc/
        openssl genpkey -algorithm RSA -out $key_name -
          ↪ pkeyopt rsa_keygen_bits:2048
        openssl rsa -pubout -in $key_name -out $pkey_name
        # openssl rsa -passin pass:$password -in $key_name
          ↪  -out $key_name # Remove passphrase

        cd ${home_dir%/}/slurm*/
        ./configure --prefix=$home_dir --exec-prefix=
          ↪ $home_dir --sysconfdir=${home_dir%/}/etc --
          ↪ localstatedir=${home_dir%/}/var --with-ssl=/
          ↪ usr/lib64/openssl #/engines/lib
        cd ${home_dir%/}/slurm*/src/
        make -j8

    else
        echo "Invalid argument. Choose either '--install -
          ↪ i' or '--configure -c'"
    fi

    if [[ $1 == "--install" || $1 == "-i" || $1 == "--
      ↪ configure" || $1 == "-c" ]]; then
        # Plugin linking
        lib_dir="${home_dir%/}/lib/slurm"
        so_files=$(find ${home_dir%/}/slurm-*/src/plugins/
          ↪  -type f -name "*.so")
        cd ${lib_dir%/}

        IFS=$'\n'
        # Create symlinks for object code
        for so_file in $so_files; do
            ln -s $so_file $(basename $so_file)
        done
        ln -s ${home_dir%/}/slurm-*/src/slurmd/slurmstepd/
          ↪ slurmstepd ${home_dir%/}/sbin/slurmstepd
    fi
elif (( major == 25 )); then
    if [[ $1 == "--install" || $1 == "-i" ]]; then
          cd ${home_dir%/}
    wget https://github.com/SchedMD/slurm/archive/refs/
      ↪ tags/slurm-25-05-1-1.zip
    unzip *slurm*.zip
    cp *slurm*.zip ${base_dir%/}
    rm -f *slurm*.zip
    mkdir --parents ${home_dir%/}{/sbin,/lib/slurm,/etc,/
      ↪ spool/{slurm,slurmd},/sys,/sys/{fs,fs/cgroup},/
      ↪ var/{run,log,log/slurm,slurm,spool/{slurm,slurmd
      ↪ }},/spool/slurmd}
```

```
            cp ${base_dir%/}/slurm.conf.template.$version
               ↪ ${home_dir%/}/etc/slurm.conf.template
            cp ${base_dir%/}/cgroup.conf ${home_dir%/}/etc
               ↪ /cgroup.conf
            cd ${home_dir%/}/slurm*/
    ./configure --prefix=${home_dir%/} --exec-prefix=
       ↪ $home_dir --sysconfdir=${home_dir%/}/etc --
       ↪ localstatedir=${home_dir%/}/var
    cd ${home_dir%/}/slurm*/src/
    make -j8
            find ${home_dir%/}/slurm-slurm-25-05-1-1/ -
               ↪ type f -name "*.so" -exec cp {} ${
               ↪ home_dir%/}/lib/slurm/ \;
    find ${home_dir%/}/slurm*/etc/ -type f -name "*lua*" -
       ↪  exec cp {} ${home_dir%/}/etc/
    cp ${home_dir%/}/etc/job_submit.lua.example ${home_dir
       ↪ %/}/etc/job_submit.lua
    elif [[ $1 == "--configure" || $1 == "-c" ]]; then
            rm -rf ${home_dir%/}/*
            cp ${base_dir%/}/slurm*.zip $home_dir
            Unzip the source code
            unzip *slurm*.zip
            Create directories
            mkdir --parents ${home_dir%/}/{lib/slurm,etc,
               ↪ spool/{slurm,slurmd},sys,sys/{fs,fs/
               ↪ cgroup},var/{run,log,log/slurm,slurm,
               ↪ spool/slurm},spool/slurmd}
            cp ${base_dir%/}/slurm.conf.template.$version
               ↪ ${home_dir%/}/etc/slurm.conf.template
    cp ${base_dir%/}/cgroup.conf ${home_dir%/}/etc/cgroup.
       ↪ conf
            cd ${home_dir%/}/slurm*/
    ./configure --prefix=${home_dir%/} --exec-prefix=
       ↪ $home_dir --sysconfdir=${home_dir%/}/etc --
       ↪ localstatedir=${home_dir%/}/var
    cd ${home_dir%/}/slurm*/src/
    make -j8
            find ${home_dir%/}/slurm-slurm-25-05-1-1/ -
               ↪ type f -name "*.so" -exec cp {} ${
               ↪ home_dir%/}/lib/slurm/ \;
            find ${home_dir%/}/slurm*/etc/ -type f -name "
               ↪ *lua*" - exec cp {} ${home_dir%/}/etc/
            cp ${home_dir%/}/etc/job_submit.lua.example ${
               ↪ home_dir%/}/etc/job_submit.lua
else
        echo "Invalid␣argument.␣Choose␣either␣'--install␣-
           ↪ i'␣or␣'--configure␣-c'"
    fi
```

```
elif (( major == 21 )); then
    if [[ $1 == "--install" || $1 == "-i" ]]; then
        rm -rf ${home_dir%/}/*
                cd ${home_dir%/}
        wget https://github.com/SchedMD/slurm/archive/refs/
            ↪ tags/slurm-21-08-6-1.zip
        unzip *slurm*.zip
        cp *slurm*.zip ${base_dir%/}
        rm -f *slurm*.zip
        mkdir --parents ${home_dir%/}{/sbin,/lib/slurm,/etc,/
            ↪ spool/{slurm,slurmd},/sys,/sys/{fs,fs/cgroup},/
            ↪ var/{run,log,log/slurm,slurm,spool/{slurm,slurmd
            ↪ }},/spool/slurmd}
        cp {${base_dir%/}/slurm.conf.template,${base_dir%/}/
            ↪ cgroup.conf} ${home_dir%/}/etc
        cd ${home_dir%/}/slurm*/
        ./configure --prefix=${home_dir%/} --exec-prefix=
            ↪ $home_dir --sysconfdir=${home_dir%/}/etc --
            ↪ localstatedir=${home_dir%/}/var
        cd ${home_dir%/}/slurm*/src/
        make -j8
    elif [[ $1 == "--configure" || $1 == "-c" ]]; then
        rm -rf ${home_dir%/}/*
        # Copy source from home_dir to installation directory
        cp ${base_dir%/}/slurm*.zip $home_dir
        Unzip the source code
        unzip *slurm*.zip
        Create directories
        mkdir --parents ${home_dir%/}/{lib/slurm,etc,spool/{
            ↪ slurm,slurmd},sys,sys/{fs,fs/cgroup},var/{run,log
            ↪ ,log/slurm,slurm,spool/slurm},spool/slurmd}
        cp {${base_dir%/}/slurm.conf.template,${base_dir%/}/
            ↪ cgroup.conf} ${home_dir%/}/etc
        cd ${home_dir%/}/slurm*/
        ./configure --prefix=${home_dir%/} --exec-prefix=
            ↪ $home_dir --sysconfdir=${home_dir%/}/etc --
            ↪ localstatedir=${home_dir%/}/var
        cd ${home_dir%/}/slurm*/src/
        make -j8
        find ${home_dir%/}/slurm-slurm-25-05-1-1/ -type f -
            ↪ name "*.so" -exec cp {} ${home_dir%/}/lib/slurm/
            ↪ \;
    else
            echo "Invalid argument. Choose either '--install -
                ↪ i' or '--configure -c'"
        fi

fi
```

Listing A.2: A prologue script meant for custom user code to be executed right before SiS deployment

```
#!/bin/bash

# Run HPC specific commands and more

# ARIS modules
module purge
module load gnu/8
module load intel/18
module load intelmpi/2018
module load python/3.9.18

# User custom code can be inserted below
```

Listing A.3: An epilogue script meant for custom user code to be executed right after SiS shutdown

```
#!/bin/bash
. ./environment_variables.sh
# Just sleep before shutdown
sleep 1
# User custom code can be inserted here
```

Listing A.4: An intermediary script executed in order to connect the variables set by the user regarding SiS allocation

```
#!/bin/bash
. ./environment_variables.sh
sbatch --nodes=$nodes_count --time=$time --partition=
    ↪ $partition_name --time=$time --mem-per-cpu=$mem_per_cpu
    ↪ --error=$errpath --output=$outpath --export=NODES_COUNT=
    ↪ $nodes_count v-slurm-lite.sh
```

Listing A.5: The main script responsible for deploying SiS and executing jobs through SiS

```
#!/bin/bash -l
#SBATCH --job-name=SlurmVirtualCluster
# Run prologue script
./prologue.sh

# Nodes for configuration and SiS sruns (have to agree with
    ↪ SBATCH options)
# total nodes - 1 (controller node)
n_slurmd=$(( NODES_COUNT - 1 ))
n_cpus=1
# Make helper functions available
. ./helpers.sh
. ./environment_variables.sh
major="${version%%.*}"
if (( major == 16 )); then
```

```
    # Avoid state errors from previous executions
    rm -r ${home_dir%/}/var/spool/{slurmd,slurm}
    mkdir --parents ${home_dir%/}{/spool/{slurm,slurmd},/var/{
        ↪ run,log,log/slurm,slurm,spool/{slurm,slurmd}}}
    #touch ${home_dir%/}/var/run/slurmctld.pid ${home_dir%/}/
        ↪ var/spool/slurm/{node_state,job_state.old,node_state.
        ↪ old,resv_state,resv_state.old,job_state,trigger_state
        ↪ ,trigger_state.old}
    chmod 755 -R ${home_dir%/}
fi
# Older versions of SLURM uses this runtime variable
nodelist=$SLURM_NODELIST
cpus=$SLURM_JOB_CPUS_PER_NODE

cnodes_txt=''

# Parse the nodelist
cnodes_list=$(parse_nlist "$nodelist")
# Get the control machine (slurmctld will run here)
control_machine=$(get_first_node_old_bash "$nodelist")

# We have to create a new slurm.conf file for each execution
# Creates a slurm.conf template that contains generic cluster
    ↪ information
# that is changing per execution
cnodes_txt+="NodeName=${cnodes_list} CPUs=${sys_cpus} Sockets=
    ↪ ${sockets} CoresPerSocket=${cores_per_socket} 
    ↪ ThreadsPerCore=${threads_per_core} RealMemory=${real_mem
    ↪ }\n"
cnodes_txt+="PartitionName=${partition_name} Nodes=${
    ↪ cnodes_list} Default=YES MaxMemPerNode=${max_mem_per_node
    ↪ } DefaultTime=24:00:00"
config_text=$(head -n -1 ${home_dir%/}/etc/slurm.conf.template
    ↪  | tail -n +2 )

major="${version%%.*}"
if (( major <= 16 )); then
        mkdir --parents ${home_dir%/}{/spool/{slurm,slurmd},/
            ↪ var/{run,log,log/slurm,slurm,spool/{slurm,slurmd
            ↪ }}}
        touch ${home_dir%/}/var/run/slurmctld.pid ${home_dir
            ↪ %/}/var/spool/slurm/{node_state,job_state,
            ↪ trigger_state}
        chmod 755 -R ${home_dir%/}

    # Create slurm.conf for older versions
    config_text=$(echo -e "ControlMachine=${control_machine}\
        ↪ n$config_text\n$cnodes_txt")
```

```
elif (( major == 25 ));then

    # Create slurm.conf for newer versions
        config_text=$(echo -e "SlurmctldHost=${control_machine
            ↪ }\n$config_text\n$cnodes_txt")
fi

echo -e "$config_text" > ${home_dir%/}/etc/slurm.conf
# Setup slurmctld
# -N, --nodes, request -N nodes allocated for the job
# -n, --ntasks, specify the -n tasks to run, -c changes this
   ↪ defualt
# -c, --cpus-per-task, request that -c cpus be allocated per
   ↪ process

# SLURM controller is ought to run in single machine - Hard
   ↪ coded options
srun -N 1 -n 1 -c 1 --nodelist=${control_machine} ${home_dir
   ↪ %/}/slurm-slurm-*/src/slurmctld/slurmctld -f ${home_dir
   ↪ %/}/etc/slurm.conf -Dvvvv &
sleep 10
# Setup slurmd
srun -N $n_slurmd -n $n_slurmd -c $n_cpus --nodelist=${
   ↪ cnodes_list} ${home_dir%/}/slurm-slurm-*/src/slurmd/
   ↪ slurmd/slurmd -f ${home_dir%/}/etc/slurm.conf -Dvvvv &
sleep 10
(
    unset SLURM_JOBID SLURM_JOB_ID SLURM_NPROCS

    ${home_dir%/}/slurm-slurm-*/src/scontrol/scontrol show
       ↪ nodes

    env -i \
    PATH=${home_dir%/}/slurm-slurm-*/src \
    HOME=$HOME \
    SLURM_NODELIST=$cnodes_list \
    SLURM_CLUSTER_NAME="v_slurm" \
    USER=$USER \
    SHELL=/bin/bash \
    SIS_CONFIG_PATH=${home_dir%/}/etc/ \
    SLURM_CONF=${home_dir%/}/etc/slurm.conf \
    # Stalls main script until all jobs are finished.
    while (( $(${home_dir%/}/slurm-slurm-*/src/squeue/squeue -
       ↪ u goumas | wc -l) > 2 )); do
        ${home_dir%/}/slurm-slurm-*/src/scancel/scancel --user
           ↪ =$USER
        sleep 1
```

```
    done
    grep -Ev '^(#|$)' job_queue.txt | while read -r path delay
       ↪ ; do
        ${home_dir%/}/slurm-slurm-*/src/sbatch/sbatch --nodes=
           ↪ $n_slurmd --time=$SBATCH_TIMELIMIT $path
        sleep $delay
    done
)
./epilogue.sh
```

Listing A.6: A collection of helper functions used for parsing and supporting installation and execution scripts

```
generate_slurm_conf (){
        ./ environment-variables.sh
    major=$1
    if (( major <= 16 )); then
        echo "ControlMachine=<some-dynamically-allocated-node>
␣␣␣␣␣␣␣␣AuthType=auth/slurm
␣␣␣␣␣␣␣␣CacheGroups=0
␣␣␣␣␣␣␣␣CryptoType=crypto/openssl
␣␣␣␣␣␣␣␣JobCredentialPrivateKey=${home_dir%/}/etc/${key_name}
␣␣␣␣␣␣␣␣JobCredentialPublicCertificate=${home_dir%/}/etc/${
   ↪ pkey_name}
␣␣␣␣␣␣␣␣EnforcePartLimits=YES
␣␣␣␣␣␣␣␣KillOnBadExit=1
␣␣␣␣␣␣␣␣LaunchType=launch/slurm
␣␣␣␣␣␣␣␣MpiDefault=pmi2
␣␣␣␣␣␣␣␣ProctrackType=proctrack/linuxproc
␣␣␣␣␣␣␣␣PropagateResourceLimitsExcept=CPU,NPROC
␣␣␣␣␣␣␣␣ReturnToService=1
␣␣␣␣␣␣␣␣SlurmctldPidFile=${home_dir%/}/var/slurm/slurmctld.pid
␣␣␣␣␣␣␣␣SlurmctldPort=50001
␣␣␣␣␣␣␣␣SlurmdPidFile=${home_dir%/}/var/slurm/slurmd.pid
␣␣␣␣␣␣␣␣SlurmdPort=50002
␣␣␣␣␣␣␣␣SlurmdSpoolDir=${home_dir%/}/var/spool/slurmd
␣␣␣␣␣␣␣␣SlurmUser=${USER}
␣␣␣␣␣␣␣␣SlurmdUser=${USER}
␣␣␣␣␣␣␣␣StateSaveLocation=${home_dir%/}/var/spool/slurm
␣␣␣␣␣␣␣␣SwitchType=switch/none
␣␣␣␣␣␣␣␣TaskPlugin=task/none
␣␣␣␣␣␣␣␣InactiveLimit=0
␣␣␣␣␣␣␣␣KillWait=30
␣␣␣␣␣␣␣␣MinJobAge=300
␣␣␣␣␣␣␣␣SlurmctldTimeout=120
␣␣␣␣␣␣␣␣SlurmdTimeout=300
␣␣␣␣␣␣␣␣Waittime=0
␣␣␣␣␣␣␣␣#
␣␣␣␣␣␣␣␣#
```

```
␣␣␣␣␣␣␣␣#␣SCHEDULING
␣␣␣␣␣␣␣␣DefMemPerCPU=2800
␣␣␣␣␣␣␣␣FastSchedule=1
␣␣␣␣␣␣␣␣MaxMemPerCPU=2800
␣␣␣␣␣␣␣␣SchedulerType=sched/backfill
␣␣␣␣␣␣␣␣SelectType=select/cons_res
␣␣␣␣␣␣␣␣SelectTypeParameters=CR_Core_Memory
␣␣␣␣␣␣␣␣#
␣␣␣␣␣␣␣␣#
␣␣␣␣␣␣␣␣#␣JOB␣PRIORITY
␣␣␣␣␣␣␣␣PriorityFlags=FAIR_TREE
␣␣␣␣␣␣␣␣PriorityType=priority/multifactor
␣␣␣␣␣␣␣␣PriorityDecayHalfLife=0
␣␣␣␣␣␣␣␣PriorityCalcPeriod=300
␣␣␣␣␣␣␣␣PriorityFavorSmall=NO
␣␣␣␣␣␣␣␣PriorityMaxAge=30-00:00:00
␣␣␣␣␣␣␣␣PriorityUsageResetPeriod=WEEKLY
␣␣␣␣␣␣␣␣PriorityWeightAge=5000
␣␣␣␣␣␣␣␣PriorityWeightJobSize=5000
␣␣␣␣␣␣␣␣PriorityWeightFairshare=20000
␣␣␣␣␣␣␣␣PriorityWeightPartition=0
␣␣␣␣␣␣␣␣PriorityWeightQOS=0

␣␣␣␣␣␣␣␣#␣LOGGING␣AND␣ACCOUNTING
␣␣␣␣␣␣␣␣JobCompType=jobcomp/none
␣␣␣␣␣␣␣␣SlurmctldDebug=info
␣␣␣␣␣␣␣␣SlurmctldLogFile=${home_dir%/}/var/log/slurm/slurmctld
    ↪ .log
␣␣␣␣␣␣␣␣SlurmdDebug=info
␣␣␣␣␣␣␣␣SlurmdLogFile=${home_dir%/}/var/log/slurm/slurmd.log

␣␣␣␣␣␣␣␣#␣COMPUTE␣NODES

␣␣␣␣␣␣␣␣NodeName=node[001-002]␣CPUs=${sys_cpus}␣Sockets=${
    ↪ sockets}␣CoresPerSocket=${cores_per_socket}␣
    ↪ ThreadsPerCore=${threads_per_core}␣RealMemory=${real_mem}
    ↪ ␣State=UNKNOWN" > ${base_dir%/}/slurm.conf.template.
    ↪ $version

        # Create cgroup.conf

        echo "#␣CGROUPS␣CONFIGURATION
␣␣␣␣␣␣␣␣CgroupMountpoint=${home_dir%/}/sys/fs/cgroup
␣␣␣␣␣␣␣␣ConstrainCores=no
␣␣␣␣␣␣␣␣ConstrainRAMSpace=no" > ${base_dir%/}/cgroup.conf


    elif (( major == 25 )); then
```

```
        echo "SlurmctldHost=<some-dynamically-allocated-node>
␣␣␣␣␣␣␣␣ClusterName=aris
␣␣␣␣␣␣␣␣DisableRootJobs=NO
␣␣␣␣␣␣␣␣EnforcePartLimits=YES
␣␣␣␣␣␣␣␣MaxJobId=67043328
␣␣␣␣␣␣␣␣GresTypes=gpu
␣␣␣␣␣␣␣␣JobSubmitPlugins=lua
␣␣␣␣␣␣␣␣KillOnBadExit=1
␣␣␣␣␣␣␣␣LaunchType=launch/slurm
␣␣␣␣␣␣␣␣MpiDefault=pmix
␣␣␣␣␣␣␣␣ProctrackType=proctrack/cgroup
␣␣␣␣␣␣␣␣PropagateResourceLimits=CORE
␣␣␣␣␣␣␣␣PropagateResourceLimitsExcept=CPU,NPROC,NOFILE,STACK
␣␣␣␣␣␣␣␣ReturnToService=1
␣␣␣␣␣␣␣␣SlurmctldPidFile=${home_dir%/}/var/slurm/slurmctld.pid
␣␣␣␣␣␣␣␣SlurmctldPort=50001
␣␣␣␣␣␣␣␣SlurmdPidFile=${home_dir%/}/var/slurm/slurmd.pid
␣␣␣␣␣␣␣␣SlurmdPort=50002
␣␣␣␣␣␣␣␣SlurmdSpoolDir=${home_dir%/}/var/spool/slurmd
␣␣␣␣␣␣␣␣SlurmdParameters=l3cache_as_socket
␣␣␣␣␣␣␣␣SlurmUser=$USER
␣␣␣␣␣␣␣␣StateSaveLocation=${home_dir%/}/var/spool/slurm
␣␣␣␣␣␣␣␣SwitchType=switch/none
␣␣␣␣␣␣␣␣TaskPlugin=task/cgroup,task/affinity
␣␣␣␣␣␣␣␣InactiveLimit=0
␣␣␣␣␣␣␣␣KillWait=30
␣␣␣␣␣␣␣␣MinJobAge=300
␣␣␣␣␣␣␣␣SlurmctldTimeout=120
␣␣␣␣␣␣␣␣SlurmdTimeout=300
␣␣␣␣␣␣␣␣Waittime=0
␣␣␣␣␣␣␣␣DefMemPerCPU=3968
␣␣␣␣␣␣␣␣SchedulerType=sched/backfill
␣␣␣␣␣␣␣␣SelectType=select/cons_tres
␣␣␣␣␣␣␣␣SelectTypeParameters=CR_Core_Memory
␣␣␣␣␣␣␣␣PriorityFlags=FAIR_TREE
␣␣␣␣␣␣␣␣PriorityType=priority/multifactor
␣␣␣␣␣␣␣␣PriorityDecayHalfLife=0
␣␣␣␣␣␣␣␣PriorityCalcPeriod=300
␣␣␣␣␣␣␣␣PriorityFavorSmall=NO
␣␣␣␣␣␣␣␣PriorityMaxAge=15-00:00:00
␣␣␣␣␣␣␣␣PriorityUsageResetPeriod=WEEKLY
␣␣␣␣␣␣␣␣PriorityWeightAge=5000
␣␣␣␣␣␣␣␣PriorityWeightFairshare=20000
␣␣␣␣␣␣␣␣PriorityWeightJobSize=2000
␣␣␣␣␣␣␣␣PriorityWeightPartition=0
␣␣␣␣␣␣␣␣PriorityWeightQOS=0
␣␣␣␣␣␣␣␣AccountingStorageEnforce=limits
␣␣␣␣␣␣␣␣AccountingStorageHost=$USER
```

```
␣␣␣␣␣␣␣␣AccountingStorageType=accounting_storage/none
␣␣␣␣␣␣␣␣AccountingStorageTRES=gres/gpu,gres/gpu:1g.10gb,gres/
    ↪ gpu:2g.20gb,gres/gpu:3g.40gb,gres/gpu:a100,gres/gpumem,
    ↪ gres/gpuutil
␣␣␣␣␣␣␣␣JobCompHost=$USER
␣␣␣␣␣␣␣␣JobCompLoc=/var/log/slurm/job_completions
␣␣␣␣␣␣␣␣JobCompUser=$USER
␣␣␣␣␣␣␣␣JobAcctGatherFrequency=30
␣␣␣␣␣␣␣␣JobAcctGatherType=jobacct_gather/cgroup
␣␣␣␣␣␣␣␣SlurmctldDebug=info
␣␣␣␣␣␣␣␣SlurmctldLogFile=${home_dir%/}/var/log/slurm/slurmctld
    ↪ .log
␣␣␣␣␣␣␣␣SlurmdDebug=quiet
␣␣␣␣␣␣␣␣SlurmdLogFile=${home_dir%/}/var/log/slurm/slurmd.log
␣␣␣␣␣␣␣␣AcctGatherEnergyType=acct_gather_energy/ipmi
␣␣␣␣␣␣␣␣AcctGatherInterconnectType=acct_gather_interconnect/
    ↪ ofed
␣␣␣␣␣␣␣␣NodeName=m[01-48]␣CPUs=128␣Sockets=2␣CoresPerSocket=64
    ↪ ␣ThreadsPerCore=1␣RealMemory=507904␣State=UNKNOWN
␣␣␣␣␣␣␣␣PartitionName=compute␣Nodes=m[01-48]␣Default=YES␣
    ↪ MaxTime=48:00:00␣MaxMemPerNode=507904␣State=DOWN" > ${
    ↪ base_dir%/}/slurm.conf.template.$version
    echo "#␣CGROUPS␣CONFIGURATION
␣␣␣␣␣␣␣␣CgroupMountpoint=${home_dir%/}/sys/fs/cgroup
␣␣␣␣␣␣␣␣ConstrainCores=no
␣␣␣␣␣␣␣␣ConstrainRAMSpace=no" > ${base_dir%/}/cgroup.conf

    elif (( major == 21 )); then
        echo "ControlMachine=slurmctld
␣␣␣␣␣␣␣␣ClusterName=v-slurm
␣␣␣␣␣␣␣␣ControlAddr=slurmctld
␣␣␣␣␣␣␣␣SlurmUser=slurm
␣␣␣␣␣␣␣␣SlurmctldPort=50001
␣␣␣␣␣␣␣␣SlurmdPort=50002
␣␣␣␣␣␣␣␣AuthType=auth/munge
␣␣␣␣␣␣␣␣StateSaveLocation=${home_dir%/}/var/lib/slurmd
␣␣␣␣␣␣␣␣SlurmdSpoolDir=${home_dir%/}/var/spool/slurmd
␣␣␣␣␣␣␣␣SwitchType=switch/none
␣␣␣␣␣␣␣␣MpiDefault=intelmpi
␣␣␣␣␣␣␣␣SlurmctldPidFile=${home_dir%/}/var/run/slurmd/
    ↪ slurmctld.pid
␣␣␣␣␣␣␣␣SlurmdPidFile=${home_dir%/}/var/run/slurmd/slurmd.pid
␣␣␣␣␣␣␣␣ProctrackType=proctrack/linuxproc
␣␣␣␣␣␣␣␣ReturnToService=0
␣␣␣␣␣␣␣␣SlurmctldTimeout=300
␣␣␣␣␣␣␣␣SlurmdTimeout=300
␣␣␣␣␣␣␣␣InactiveLimit=0
␣␣␣␣␣␣␣␣MinJobAge=300
```

```
␣␣␣␣␣␣␣␣KillWait=30
␣␣␣␣␣␣␣␣Waittime=0
␣␣␣␣␣␣␣␣SchedulerType=sched/backfill
␣␣␣␣␣␣␣␣SelectType=select/cons_res
␣␣␣␣␣␣␣␣SelectTypeParameters=CR_CPU_Memory
␣␣␣␣␣␣␣␣FastSchedule=1
␣␣␣␣␣␣␣␣SlurmctldDebug=3
␣␣␣␣␣␣␣␣SlurmctldLogFile=${home_dir%/}/var/log/slurm/slurmctld
    ↪ .log
␣␣␣␣␣␣␣␣SlurmdDebug=3
␣␣␣␣␣␣␣␣SlurmdLogFile=${home_dir%/}/var/log/slurm/slurmd.log
␣␣␣␣␣␣␣␣JobCompType=jobcomp/filetxt
␣␣␣␣␣␣␣␣JobCompLoc=${home_dir%/}/var/log/slurm/jobcomp.log
␣␣␣␣␣␣␣␣JobAcctGatherType=jobacct_gather/linux
␣␣␣␣␣␣␣␣JobAcctGatherFrequency=30
␣␣␣␣␣␣␣␣NodeName=m[01-48]␣CPUs=128␣Sockets=2␣CoresPerSocket=64
    ↪ ␣ThreadsPerCore=1␣RealMemory=507904␣State=UNKNOWN
␣␣␣␣␣␣␣␣PartitionName=compute␣Nodes=m[01-48]␣Default=YES␣
    ↪ MaxTime=48:00:00␣MaxMemPerNode=507904␣State=DOWN" > ${
    ↪ base_dir%/}/slurm.conf.template
    echo "#␣CGROUPS␣CONFIGURATION
␣␣␣␣␣␣␣␣CgroupMountpoint=${home_dir%/}/sys/fs/cgroup
␣␣␣␣␣␣␣␣ConstrainCores=no
␣␣␣␣␣␣␣␣ConstrainRAMSpace=no" > ${base_dir%/}/cgroup.conf

}

parse_nlist() {
    # If arguments of function different from 2 exit
    if [[ $# -ne 1 ]]; then
        echo "Function␣parse_nlist␣requires␣1␣arguments␣<
            ↪ nodelist>"
        exit 1
    fi
    # Extract how many chars on a range entry
    num_chars=$(echo "$1" | cut -d'[' -f 2 | cut -d']' -f 1 |
        ↪ cut -d'-' -f 2 | cut -d ',' -f 1 | wc -c)
    num_chars=$(expr $num_chars - 1)
    hostchar=$(echo "$1" | cut -d'[' -f1)
    range=$(echo "$1" | cut -d'[' -f2 | cut -d']' -f1)
    IFS=, read -r -a parts <<< "$range"
    for i in "${!parts[@]}"; do
        if [[ ${parts[$i]} == *-* ]]; then
            start_node="$(echo␣${parts[$i]}␣|␣cut␣-d'-'␣-f1)"
            end_node="$(echo␣${parts[$i]}␣|␣cut␣-d'-'␣-f2)"
            if [[ $num_chars -eq 3 ]]; then
                printf -v start_node "%03i" $(( 10#$start_node
                    ↪ ))
```

```
                printf -v end_node "%03i" $(( 10#$end_node ))
            else
                printf -v start_node "%02i" $(( 10#$start_node
                    ↪ ))
                printf -v end_node "%02i" $(( 10#$end_node ))
            fi
            parts[$i]="${start_node}-${end_node}"
        else
            node=${parts[$i]}
            if [[ $num_chars -eq 3 ]]; then
                printf -v node "%03i" $(( 10#$node ))
            else
                printf -v node "%02i" $(( 10#$node ))
            fi
            parts[$i]=$node
        fi
    done
    part1=${parts[0]}
    if [[ $part1 == *-* ]]; then
        start_node=$(echo $part1 | cut -d'-' -f1)
        end_node=$(echo $part1 | cut -d'-' -f2)
        if [[ $(expr $end_node - $start_node) -eq 1 ]]; then
            parts[0]=$end_node
        else
            start_node=$(echo "$(expr␣$start_node␣+␣1)")
            if [[ $num_chars -eq 3 ]]; then
                printf -v start_node "%03i" $(( 10#$start_node
                    ↪ ))
            else
                printf -v start_node "%02i" $(( 10#$start_node
                    ↪ ))
            fi
            parts[0]="${start_node}-${end_node}"
        fi
    else
        parts=( "${parts[@]/$part1}" )
    fi
    echo "$hostchar[$(echo␣${parts[@]}␣|␣sed␣'s/␣/,/g')]"
}

## get_first_node_old_bash: parses node list
## Takes 1 argument: the part of nodelist
## which is delimited with '[' and '-'
## Returns the starting node of the nodelist
get_first_node_old_bash () {
    hostchar=$(echo "$1" | cut -d'[' -f1)
    range=$(echo "$1" | cut -d'[' -f2 | cut -d']' -f1)
    IFS=, read -r -a parts <<< "$range"
```

```
    if [[ $parts[0] == *-* ]]; then
        node=$(echo $parts[0] | cut -d '-' -f1 )
        echo "$hostchar$node"
    elif [[ $parts == $hostchar ]]; then
        echo "$hostchar"
    else
        echo "$hostchar$parts"
    fi
}

## get_start_number: parses node list
## Takes 1 argument: the part of nodelist
## which is delimited with commas
## Returns the starting node of the nodelist
get_start_number() {
    input_string="$1"
    regex="^([^[]+)\[([0-9]+)-[0-9]+\]$"

    if [[ $input_string =~ $regex ]]; then
        some_name="${BASH_REMATCH[1]}"
        start_number="${BASH_REMATCH[2]}"
        result="${some_name}${start_number}"
        echo "$result"
    else
        echo "Invalid input format"
    fi
}

## check2nodes: parses node list and cpus
## Takes 3 arguments: the part of nodelist
## which is delimited with commas and the
## corresponding cpus of that part of the nodelist
## and the index of the nodelist
## Returns the number of nodes along with the
## cpus for that part of the nodelistt
## if index is 1, then the first node is left out

check2_nodes(){
    # If arguments of function different from 2 exit
    if [[ $# -ne 3 ]]; then
        echo "Function check2_nodes requires 3 arguments <
            ↪ nodelist> <cpus> <index>"
        exit 1
    fi
    # If nodelist is empty exit
    if [[ -z $1 ]]; then
        echo "Nodelist is empty"
        exit 1
```

```
    fi
    # If cpus is empty exit
    if [[ -z $2 ]]; then
        echo "CPUs is empty"
        exit 1
    fi
    local cpus=$(echo $2 | cut -d'(' -f1)

    if [[ $3 -eq 1 ]]; then
        if [[ $1 == *[* ]]; then
            hostchar=$(echo "$1" | cut -d'[' -f1)
            start_node=$(echo "$1" | cut -d'[' -f2 | cut -d'-'
              ↪  -f1)
            end_node=$(echo "$1" | cut -d'-' -f2 | cut -d']' -
              ↪ f1)
            if [[ $(expr $end_node - $start_node) -eq 1 ]];
              ↪ then
                printf -v end_node "%03d" "$end_node"
                echo "$hostchar$end_node $cpus"
            else
                start_node=$(echo "$(expr $start_node + 1)")
                printf -v start_node "%03d" "$start_node"
                printf -v end_node "%03d" "$end_node"
                echo "$hostchar[${start_node}-${end_node}]
                  ↪ $cpus"
            fi
        else
            echo "control $cpus"
        fi
    else
        echo "$1 $cpus"
    fi

}
################## --END-- ##################
```

Appendix B

Listing B.1: Helper functions for sorting workloads based on requested threads

```
#!/bin/bash -l

function find_max {
    # Find max in a list of four integers

    [[ $# -ne 5 ]] || { echo "Warning: find_max() requires
       ↪ exactly 4 arguments." >&2; return 1; }
    local max_val=$1
    shift
    for num in "$@"; do
        [[ "$num" =~ ^-?[0-9]+$ ]] || { echo "Error: '$num' is
           ↪ not an integer." >&2; return 1; }
        (( num > max_val )) && max_val="$num"
    done
    echo "$max_val" # Print the final maximum value
}

function find_min {
    # Find max in a list of four integers

    [[ $# -ne 5 ]] || { echo "Warning: find_min() requires
       ↪ exactly 4 arguments." >&2; return 1; }
    local min_val=$1
    shift
    for num in "$@"; do
        [[ "$num" =~ ^-?[0-9]+$ ]] || { echo "Error: '$num' is
           ↪ not an integer." >&2; return 1; }
        (( num < min_val )) && min_val="$num"
    done
    echo "$min_val" # Print the final maximum value
}

function sort_apps {
    [[ $# -ne 5 ]] || { echo "Warning: sort_apps() requires
       ↪ exactly 4 arguments." >&2; return 1; }
    local app1=$1
    local app2=$2
```

```
    local app3=$3
    local app4=$4
    printf "%s\n" "$@" | sort -t '.' -k3,3n
}
```

Listing B.2: The code for basic parsing and configuration before sumbitting the script to run the benchmarks in a co-located manner

```
#! /bin/bash -l
source find_max_min.sh
# Quarter node cores
qnc=5

cases=(mg.D.64 ft.D.64 bt.D.64 sp.D.64)
DEBUG=false
POLICY="maxnodes"

if $DEBUG; then
        echo "Debugging mode"
        mode="dbg"
        time="00:20:00"
        mkdir -p $PWD/logs_dbg
else
        mode="prod"
        time="00:10:00"
        mkdir -p $PWD/logs
fi

for case in "${cases[@]}";do

        IFS='_' read -r app app2 app3 app4<<< "$case"
        IFS='.' read -r name class proc <<< "$app"
        IFS='.' read -r name2 class2 proc2 <<< "$app2"
        IFS='.' read -r name3 class3 proc3 <<< "$app3"
        IFS='.' read -r name4 class4 proc4 <<< "$app4"

        apps=$(sort_apps $app $app2 $app3 $app4)
        readarray -t apps_array <<< "$apps"

        # Get how many nodes will be needed
        max_app=${apps_array[${#apps_array[@]}-1]}
        max_proc=$(awk -F. '{print $3}' <<< "$max_app")
        count_processes=$(( max_proc * 4 ))
        nodes_count=$(( ($count_processes + 19)/20 ))
        apps_with_copies=()
        # Calculate how many copies of each program will be
          ↪ needed
        for app in "${apps_array[@]}"; do
                proc=$(awk -F. '{print $3}' <<< "$app")
```

```
                copies=$(( ( max_proc + proc - 1 ) / proc ))
                apps_with_copies+=("${app}.${copies}")
        done
        apps_joined=$(IFS=_; echo "${apps_with_copies[*]}")
        printf '%s\n' "${apps_with_copies[@]}"
        echo "max_proc=$max_proc␣count_processes=
            ↪ $count_processes␣nodes_count=$nodes_count"
        sbatch --nodes=$nodes_count --partition=compute --time
            ↪ =$time --export=POLICY=$POLICY,CASE=$case,
            ↪ APPS_ARRAY="$apps_joined",CORES=$qnc,NODES_COUNT=
            ↪ $nodes_count,DEBUG=$DEBUG,TIME=$time run_cos.
            ↪ NAS_NAS.sh
done
```

Listing B.3: The main script to pin the processes based on the nodelist allocation and execute the co-located benchmarks

```
#!/bin/bash

#SBATCH --job-name=run
#SBATCH --output=/users/pa23/goumas/nfloros/
    ↪ quarter_socket_workloads/logs/runs/run.%j.out
#SBATCH --error=/users/pa23/goumas/nfloros/
    ↪ quarter_socket_workloads/logs/runs/run.%j.err
#SBATCH --cpus-per-task=1
#SBATCH --account=pa220401
#SBATCH --exclusive

module purge

module load gnu/8
module load intel/18
module load intelmpi/2018

DBG=""
I_DBG=""

if $DEBUG; then
    DBG="_dbg"
    I_DBG="-genv␣I_MPI_DEBUG=+5"
fi
IFS=: read -r h m s <<< "$TIME"
walltime=$((10#$h * 3600 + 10#$m * 60 + 10#$s))

IFS='_' read -r -a apps_array <<< "$APPS_ARRAY"
n_apps=${#apps_array[@]}
if (( n_apps == 0 )); then
    echo "No␣apps␣parsed␣from␣apps_joined='$APPS_ARRAY'" >&2
    exit
```

```
fi

# Source the function to create the machine files
if [[ "${POLICY}" == "dense" ]];then
        source policies/dense.sh
else
        # Default
        source policies/maxnodes.sh
fi

export NPB_TIMER_FLAG=on

NAS_PATH="/users/pa23/goumas/nfloros/NAS-Benchmarks/NPB3.4.2/
    ↪ NPB3.4-MPI/bin/"
LOG_PATH="/users/pa23/goumas/nfloros/quarter_socket_workloads/
    ↪ logs${DBG}/${CASE}"

mkdir -p $LOG_PATH

max_app=${apps_array[${#apps_array[@]}-1]}
max=$(awk -F. '{print $3}' <<< "$max_app")

nodes=($(scontrol show hostname $SLURM_NODELIST))

# First remove all the machine files in the path
find "${LOG_PATH}" -maxdepth 1 -name 'nodelist*' -exec rm {}
    ↪ \;
# Then remove the pipe lock file
find  "${LOG_PATH}" -maxdepth 1 -name 'pipe' -exec rm {} \;
# Then create the machine files for each app
create_nodelist_files "${LOG_PATH}" "$CORES" "$APPS_ARRAY" "${
    ↪ nodes[@]}"

declare -a app_name app_class app_procs app_copies
for (( i=0; i<n_apps; i++ )); do
    IFS='.' read -r app_name[i] app_class[i] app_procs[i]
        ↪ app_copies[i] <<< "${apps_array[i]}"
    # validate
    if ! [[ "${app_procs[i]}" =~ ^[0-9]+$ ]] || ! [[ "${
        ↪ app_copies[i]}" =~ ^[0-9]+$ ]]; then
        echo "Malformed app entry: '${apps_array[i]}'.
            ↪ Expected PROGRAM.CLASS.PROCS.COPIES with numeric
            ↪ PROCS and COPIES." >&2
        return 1
    fi
done
# Current slot combinations. Left array is socket 0, right
    ↪ array is socket 1
```

```
# A = [2 2 2] [3 3 3]
# B = [2 2 2] [3 3 3]
# C = [3 3 3] [2 2 2]
# D = [3 3 3] [2 2 2]
exclude_list=(0,1,4-12,16-19 0-3,7-15,18,19 0-6,10-17
    ↪ 2-9,13-19)
for (( i=0; i<n_apps; i++ )); do
        for (( c=0; c < app_copies[i] ; c++))
        {
                while [ ! -f "$LOG_PATH/pipe" ]; do
                        copy=$(( c + 1))
                        local_name=${app_name[i]}.${app_class[
                            ↪ i]}.${app_procs[i]}-${i}.${copy}

                        app_start=$(date +%s)
                        mpirun ${I_DBG} -genv
                            ↪ I_MPI_PIN_PROCESSOR_EXCLUDE_LIST=
                            ↪ "${exclude_list[i]}" -genv
                            ↪ I_MPI_PIN_PROCESSOR_LIST allcores
                            ↪  -machinefile ${LOG_PATH}/
                            ↪ nodelist.${app_name[i]}.${
                            ↪ app_class[i]}.${app_procs[i]}-${i
                            ↪ }.${copy} -np ${app_procs[i]} ${
                            ↪ NAS_PATH}/${app_name[i]}.${
                            ↪ app_class[i]}.x 1>> ${LOG_PATH}/$
                            ↪ {local_name}.out
                        app_end=$(date +%s)

                        difference=$(( app_end - app_start ))
                        echo "DATE␣of␣${local_name}␣in␣seconds
                            ↪ :␣$difference" 1>> $LOG_PATH/${
                            ↪ local_name}.out
                done
        } &
done
sleep $walltime
touch ${LOG_PATH}/pipe
```

Listing B.4: The code for basic parsing and configuration before sumbitting the main script for running the benchmarks on a single node each

```
#! /bin/bash -l

# Enter your cases here
cases=(
bt.D.64
)

for case in "${cases[@]}";do
```

```
        IFS='.' read -r name class procs <<< "$case"

        nodes=$(( ${procs} / 20 ))
        if (( ${nodes} * 20 < ${procs} ));then
                nodes=$(( ${nodes} + 1 ))
        fi

        echo "app=${name}.${class}.${procs},nodes=$nodes,
           ↪ allocation=compact"
        sbatch --nodes=$nodes --partition=compute --export=APP
           ↪ =$name,CLASS=$class,PROCS=$procs run_cmp.NAS.sh

done
```

Listing B.5: The main script to execute the benchmarks each on exclusive nodes

```
#!/bin/bash

#SBATCH --job-name=compact
#SBATCH --output=/users/pa23/goumas/nfloros/
   ↪ quarter_socket_workloads/logs/runs/compact.%j.out
#SBATCH --error=/users/pa23/goumas/nfloros/
   ↪ quarter_socket_workloads/logs/runs/compact.%j.err
#SBATCH --cpus-per-task=1
#SBATCH --account=pa220401
#SBATCH --exclusive
#SBATCH --nodes=4
#SBATCH --time=1-01:25:00

module load gnu/8
module load intel/18
module load intelmpi/2018
LOG_PATH="/users/pa23/goumas/nfloros/quarter_socket_workloads/
   ↪ logs"
nas=(ep cg bt ft mg lu sp is)

for prog in "${nas[@]}"; do
        mkdir -p $LOG_PATH/${prog}_compact/
        while [ ! -f "$LOG_PATH/${prog}_compact/pipe" ]; do
                mpirun -np 64 /users/pa23/goumas/nfloros/NAS-
                   ↪ Benchmarks/NPB3.4.2/NPB3.4-MPI/bin/${prog
                   ↪ }.D.x > $LOG_PATH/${prog}_compact/
                   ↪ compact_${prog}.out
        done &
        sleep 600
        touch "$LOG_PATH/${prog}_compact/pipe"
done
```

Listing B.6: A function implementing the allocation policy by generating the nodelist definition files to assign each node the applications to execute

```
#!/usr/bin/env bash
set -euo pipefail

# create_nodelist_files
#
# Usage:
# create_nodelist_files <path> <qnc> <apps_joined> <host1> [
    ↪ host2 ...]
#
# - path: directory to write nodelist files (created if
    ↪ missing)
# - qnc: quarter-node cores (e.g. 5)
# - apps_joined: underscore-separated "PROGRAM.CLASS.PROCS.
    ↪ COPIES" entries
# - remaining args: hostnames (slurm nodenames) to use for
    ↪ placement
#
# Writes files: ${path}/nodelist.<app_idx>.<copy_idx>  and
    ↪ node-layout.<node_idx> for debug.

######################## NOTE TO SELF ########################
# There will always be an underutilized node if processes  #
# are multiples of 2. That happens because 2^n has only a  #
# prime factor (2) whilst an ARIS socket contains 20 cores #
# and 20 has 2 prime factors 2 and 5, and 5 is not a prime #
# factor of 2^n.                                           #
############################################################
function create_nodelist_files {
local path=$1
local qnc=$2
local apps_joined=$3
shift 3

# hosts
local hostnames=( "$@" )
local host_count=${#hostnames[@]}
if (( host_count == 0 )); then
    echo "create_nodelist_files: no hostnames provided" >&2
    return 1
fi

mkdir -p "$path"

# parse apps
IFS='_' read -r -a apps_array <<< "$apps_joined"
local n_apps=${#apps_array[@]}
```

```
if (( n_apps == 0 )); then
    echo "No apps parsed from apps_joined='$apps_joined'" >&2
    return 1
fi

# decode each app entry into arrays: name, class, procs,
  ↪ copies
declare -a app_name app_class app_procs app_copies
local i
for (( i=0; i<n_apps; i++ )); do
    IFS='.' read -r app_name[i] app_class[i] app_procs[i]
      ↪ app_copies[i] <<< "${apps_array[i]}"
    # validate
    if ! [[ "${app_procs[i]}" =~ ^[0-9]+$ ]] || ! [[ "${
      ↪ app_copies[i]}" =~ ^[0-9]+$ ]]; then
        echo "Malformed app entry: '${apps_array[i]}'.
          ↪ Expected PROGRAM.CLASS.PROCS.COPIES with numeric
          ↪ PROCS and COPIES." >&2
        return 1
    fi
done



# Slots_per_node is 4 (four copies per node).
local slots_per_node=4

# validate qnc
if ! [[ "$qnc" =~ ^[0-9]+$ ]] || (( qnc <= 0 )); then
    echo "Invalid qnc (must be positive integer): '$qnc'" >&2
    return 1
fi

# Prepare per-copy files and clear any previous ones
for (( i=0; i<n_apps; i++ )); do
    for (( c=0; c<app_copies[i]; c++ )); do
        copy=$(( c + 1))
        : > "${path}/nodelist.${app_name[i]}.${app_class[i]}.$
          ↪ {app_procs[i]}-${i}.${copy}"
    done
done

# For each app and for each copy, allocate app_procs processes
for (( i=0; i<n_apps; i++ )); do
    local need_per_copy=$(( app_procs[i]))
    local node_idx=0
    for (( c=0; c<app_copies[i]; c++ )); do
        local remaining_for_copy=$need_per_copy
```

```
        while (( remaining_for_copy > 0 )); do
            local host="${hostnames[ node_idx % host_count ]}"

            # assign up to qnc cores from this slot (may be
              ↪ partial on last slot)
            local assign_cores=$(( remaining_for_copy < qnc ?
              ↪ remaining_for_copy : qnc ))

            # append to the per-copy machine file
            copy=$(( c + 1 ))
            printf '%s:%d\n' "$host" "$assign_cores" >> "${
              ↪ path}/nodelist.${app_name[i]}.${app_class[i
              ↪ ]}.${app_procs[i]}-${i}.${copy}"

            # decrement remaining and advance slot
            remaining_for_copy=$(( remaining_for_copy -
              ↪ assign_cores ))
            node_idx=$(( node_idx + 1 ))

            # safety guard
            if (( node_idx > 10000000 )); then
                echo "create_nodelist_files: aborting after
                  ↪ too many slots (possible bug)" >&2
                return 1
            fi
        done
        # this copy finished
    done
done

echo "create_nodelist_files: created per-copy machine files
  ↪ under $path (no socket splitting)."
return 0
}
```