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

Co-scheduling algorithms for HPC applications

Diploma Thesis

of

Myrsini Kellari

Supervisor: George Goumas
Associate Professor NTUA

Athens, February 2025

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Αλγόριθμοι συνδρομολόγησης παράλληλων εφαρμογών

Διπλωματική εργασία

της

Μυρσίνης Κελλάρη

Επιβλέπων: Γεώργιος Γκούμας
Αναπληρωτής Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 7η Μαρτίου 2025.

.....	.....	.....
Γεώργιος Γκούμας	Νεκτάριος Κοζύρης	Διονύσιος Πνευματικάτος
Αν. Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ

Αθήνα, Φεβρουάριος 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

Έχω διαβάσει και κατανοήσει τους κανόνες για τη λογοκλοπή και τον τρόπο σωστής αναφοράς των πηγών που περιέχονται στον οδηγό συγγραφής Διπλωματικών Εργασιών. Δηλώνω ότι, από όσα γνωρίζω, το περιεχόμενο της παρούσας Διπλωματικής Εργασίας είναι προϊόν δικής μου εργασίας και υπάρχουν αναφορές σε όλες τις πηγές που χρησιμοποίησα.
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.

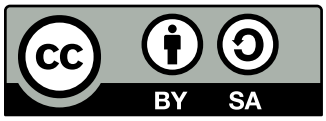
Μυρσίνη Κελλάρη, Αθήνα 2025

Statement About Plagiarism And Intellectual Property Theft

I have read and understood the rules about plagiarism and the proper way of referencing contained in the Diploma Thesis writing guide. I declare that, to the best of my knowledege, the content of this Diploma Thesis is the product of my own work and there are refrences to all sources that I have used.
The opinions and conclusions contained in this Diploma Thesis are of the author and should not be interpreted as representing the official views of the School of Electrical and Computer Engineering or the National Technical University of Athens.

Myrsini Kellari, Athens 2025

.....
Μυρσίνη Κελλάρη,
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικος Υπολογιστών ΕΜΠ



Περίληψη

Η παρούσα διπλωματική εργασία εξετάζει την ανάπτυξη και αξιολόγηση αλγορίθμων συνδρομολόγησης (co-scheduling) για συστήματα High-Performance Computing (HPC), με στόχο τη βελτιστοποίηση της χρήσης των πόρων, διατηρώντας παράλληλα υψηλή απόδοση του συστήματος και ικανοποίηση των χρηστών. Η αυξανόμενη ζήτηση για υπολογιστική ισχύ σε τομείς όπως η επιστημονική έρευνα, η τεχνητή νοημοσύνη (AI) και η ανάλυση μεγάλων δεδομένων (big data) έχει καταστήσει τα συστήματα HPC απαραίτητα. Ωστόσο, αυτά τα συστήματα συχνά υποφέρουν από υποχρησιμοποίηση των πόρων, οδηγώντας σε αυξημένη κατανάλωση ενέργειας και λειτουργικά κόστη. Οι παραδοσιακοί αλγόριθμοι χρονοδρομολόγησης, όπως ο First Come First Serve (FCFS) και ο EASY, δεν παρέχουν κάποια λύση.

Για αυτό,προτείνεται η συνδρομολόγηση (co-scheduling) ως λύση. Η συνδρομολόγηση επιτρέπει σε πολλαπλές εργασίες να μοιράζονται τους υπολογιστικούς κόμβους, μειώνοντας τον ανταγωνισμό για πόρους και βελτιώνοντας την αποδοτικότητα του συστήματος. Αυτό είναι ιδιαίτερα ωφέλιμο όταν οι συνεκτελούμενες εργασίες έχουν διαφορετικές απαιτήσεις πόρων, όπως εργασίες με έντονη χρήση μνήμης και εργασίες με έντονη χρήση επεξεργαστή. Εισάγονται αλγόριθμοι συνδρομολόγησης, οι οποίοι εξετάζουν την επίδραση της συνδρομολόγησης σε κλασσικούς αλγορίθμους, όπως ο EASY Co-schedule, αλλά και νέοι όπως ο Popularity, ο Filler και ο Two Factors. Αυτοί οι αλγόριθμοι αξιολογούνται χρησιμοποιώντας τον Efficient Lightweight Scheduling Estimator (ELiSE), έναν προσομοιωτή βασισμένο σε Python που επιτρέπει τον ελεγχόμενο έλεγχο των πολιτικών προγραμματισμού. Η αξιολόγηση βασίζεται σε βασικές μετρικές απόδοσης, όπως η επιτάχυνση του χρόνου ολοκλήρωσης (απόδοση συστήματος) και η μέση επιβράδυνση εργασιών (ικανοποίηση χρηστών).

Τα πειραματικά αποτελέσματα δείχνουν ότι οι αλγόριθμοι συνδρομολόγησης, ιδιαίτερα ο SJF-Filler (μία εκδοχή του Two Factors), επιτυγχάνουν σημαντικές βελτιώσεις στην απόδοση του συστήματος και την ικανοποίηση των χρηστών, καθιστώντας τους υποσχόμενους υποψήφιους για πραγματικά συστήματα HPC. Ωστόσο, η συνδρομολόγηση μπορεί να οδηγήσει σε αύξηση του χρόνου εκτέλεσης μεμονωμένων εργασιών, υπογραμμίζοντας την ανάγκη για εξισορρόπηση μεταξύ της αποδοτικότητας του συστήματος και της εμπειρίας των χρηστών.

Τα αποτελέσματα υποδηλώνουν ότι η συνδρομολόγηση μπορεί να ενισχύσει την απόδοση των συστημάτων HPC, αλλά απαιτείται προσεκτική διαχείριση για να διασφαλιστεί η δικαιοσύνη και η ικανοποίηση των χρηστών. Μελλοντικές επεκτάσεις περιλαμβάνουν τη δοκιμή των αλγορίθμων σε πραγματικά συστήματα HPC, την εξερεύνηση εναλλακτικών στρατηγικών συνεκτέλεσης και την ενσωμάτωση τεχνικών μηχανικής μάθησης για περαιτέρω βελτιστοποίηση των αποφάσεων χρονοδρομολόγησης.

Λέξεις Κλειδιά

High Performance Computing (HPC), Συνδρομολόγηση, Αλγόριθμοι Συνδρομολόγησης, Αλγόριθμοι Χρονοδρομολόγησης, Μετρικές, Προσομοίωση

Abstract

This thesis explores the development and evaluation of co-scheduling algorithms for High-Performance Computing (HPC) systems, aiming to optimize resource utilization while maintaining high system performance and user satisfaction. The growing demand for computational power in fields such as scientific research, artificial intelligence, and big data analytics has made HPC systems essential. However, these systems often suffer from underutilization of resources, leading to increased energy consumption and operational costs. Traditional scheduling algorithms, such as First Come First Serve (FCFS) and EASY, cannot provide a solution.

To address these challenges, co-scheduling is proposed as a solution. Co-scheduling allows multiple jobs to share computational nodes, reducing resource contention and improving system efficiency. This is particularly beneficial when co-allocated jobs have different resource demands, such as memory-intensive and compute-intensive tasks, which can lead to improved system performance. However, co-scheduling also introduces challenges, such as inter-job interference and fairness issues, which must be carefully managed. The research introduces several co-scheduling algorithms, including EASY Co-schedule, Largest Area First Co-schedule (LAF-Co), Popularity, Shortest Job First Co-schedule (SJF-Co), Longest Job First Co-schedule (LJF-Co), Filler, and Two Factors. These algorithms are evaluated using the Efficient Lightweight Scheduling Estimator (ELiSE), a Python-based simulator that enables controlled testing of scheduling policies. The evaluation is based on key metrics such as makespan speedup (system performance) and mean job slowdown (user satisfaction).

Experimental results demonstrate that co-scheduling algorithms, particularly SJF-Filler (a Two Factors variant), achieve significant improvements in makespan speedup and mean job speedup, while maintaining low mean slowdown values. These algorithms effectively balance system performance and user satisfaction, making them promising candidates for real-world HPC systems. However, co-scheduling can lead to increased execution times for individual jobs, highlighting the trade-off between system efficiency and user experience.

The findings suggest that co-scheduling can enhance the performance and efficiency of HPC systems, but careful management is required to ensure fairness and user satisfaction. Future work includes testing the algorithms on real HPC systems, exploring alternative colocation strategies, and integrating machine learning techniques to further optimize scheduling decisions.

Keywords

High Performance Computing (HPC), Co-scheduling, Co-scheduling Algorithms, Scheduling Algorithms, Performance Metrics, Simulation

Acknowledgements

This work marks the successful completion of my studies at the School of Electrical and Computer Engineering of the National Technical University of Athens.

First, I would like to express my sincere gratitude to my thesis supervisor, Assistant Professor George Goumas, for his invaluable support and guidance throughout this Diploma Thesis.

Furthermore, I would like to thank PhD candidate Nikos Triantafyllis and Stratos Karapanagiotis for their encouragement, constructive advice, and continuous support. Their insights and contributions have greatly enriched this work.

Last but not least, I am deeply grateful to my family for always being there for me and supporting me with their love, and to my friends for making these years so special.

Contents

Chapter 1

Introduction

In this Chapter, the motivation behind the development of co-scheduling algorithms in High-Performance Computing (HPC) systems is presented. The Chapter concludes with an outline of the Thesis structure.

1.1 Motivation - Problem Statement

The growing demand for computational power has made supercomputers and HPC systems essential. However, these systems consume vast amounts of energy, significantly increasing operational costs, especially during the current energy crisis. As a result, optimizing the performance of HPC systems has become essential.

One promising optimization strategy is co-scheduling. In contrast with traditional scheduling, where nodes are exclusively allocated to a single job, in co-scheduling jobs can be striped (occupying twice as much nodes), leaving half of the allocated nodes empty, allowing other jobs to allocate these nodes [1]. Research has shown that when the co-allocated jobs have different demands on resources (memory intensive jobs versus compute intensive), the system performance can be improved [2, 3]. It is important to be noted that due to the reduced resource contention the overall system performance can be improved, however individual job performance may be decreased as some jobs benefit in the expence of their co-allocated job.

The scope of this Thesis is to develop and evaluate co-scheduling algorithms designed to take advantage of these benefits while ensuring fairness between jobs. These algorithms are compared against traditional scheduling methods, such as First Come First Serve (FCFS), using a simulation environment, where resource allocation process can be easily intervened. The evaluation is based on two key metrics: system makespan speedup (a measure of system performance) and mean job slowdown (a measure of user satisfaction).

1.2 Thesis Outline

Following the introductory Chapter, the Thesis is structured as follows:

- **Chapter 2**
  This Chapter provides an overview of HPC systems, focusing on their architecture and resource management. It introduces traditional scheduling approaches and highlights the limitations that motivate the need for co-scheduling. The Chapter concludes by defining the evaluation metrics used throughout this study.

- **Chapter 3**
  Co-scheduling is examined, with a focus on its benefits, such as reduced resource contention and improved performance. The challenges of node sharing, including inter-job interference and fairness issues, are also discussed. This Chapter establishes the theoretical foundation for the proposed algorithms.

- **Chapter 4**
  The HPC Simulator, developed by the Computing Systems Laboratory, is introduced. Its architecture, key components, and functionality are described. The Chapter emphasizes how the simulator enables controlled testing of scheduling algorithms, particularly in scenarios that would be impractical to replicate on real systems.

- **Chapter 5**
  Traditional scheduling algorithms, such as FCFS and Backfilling, are reviewed. The Chapter then introduces and explains the two proposed co-scheduling algorithms, describing their design principles, operation, and expected advantages.

- **Chapter 6**
  This Chapter presents the experimental results obtained using the HPC Simulator. The performance of the proposed algorithms is evaluated against traditional approaches based on system makespan speedup and mean job slowdown. Insights into algorithm behavior and trade-offs are provided through detailed analysis.

- **Chapter 7**
  The Thesis concludes with a summary of results and their implications for HPC system optimization. Limitations of the study are discussed, and recommendations for future work, such as real-world implementation and further refinement of the algorithms, are provided.

Chapter 2

HPC Scheduling Background

In this Chapter we provide the background on HPC, scheduling in HPC and the metrics that are being used to evaluate the performance of such systems.

2.1 High Performance Computing (HPC)

The term High Performance Computing (HPC) refers to the practice of utilizing supercomputers and parallel processing techniques to solve complex computational problems that require immense processing power. HPC systems are able to perform trillions of calculations per second, enabling them to tackle problems that require millions of computations and/or big data sets. HPC systems are widely used in scientific fields such as physics, chemistry, biology, and meteorology. They are used by researchers who run applications such as molecular simulations, computational fluid dynamics (CFD), and weather prediction [4]. In recent years, HPC has also found applications in artificial intelligence, financial modeling, and big data analytics.

HPC systems are typically organized into clusters. Clusters consist of many separate servers (computers), called nodes, which work together to perform computations in parallel. The nodes are connected via a fast interconnect [5], to facilitate fast data exchange. Each node in a cluster typically consists of multiple CPU cores, GPUs, and memory modules, allowing for efficient execution of large-scale computations.

The applications in clusters must be designed to take advantage of parallel processing. In parallel processing, applications are divided into hundreds or even thousands jobs that execute at the same time and solve a common problem [4]. In order to co-operate, jobs have to communicate with each other and access common data.

Based on the memory model, systems can be classified into two categories:

- Shared Memory Model
- Distributed Memory Model

In shared memory systems, each process can access the main memory, as a result memory access is fast. However, there is a risk of race conditions and synchronization techniques are required. The prototype OpenMP, is mainly used so code parallelization is not particularly complicated [6].

In distributed memory systems, processes cannot access data that belong to other processes, and as a result the communication between processes is achieved through message passing.

Message passing adds overhead on the execution, but allows scalability as there is no memory contention between the processes. In such systems the prototype MPI (Message Passing Interface) is used [7].

2.2 Scheduling

2.2.1 Scheduling intro

In systems, we refer to scheduling as the process of managing and allocating recources to tasks efficiently. Scheduling algorithms dictate the order in which jobs or tasks are executed. The goal of each scheduling algorithm may vary, but mostly fall into one of these:

- Throughput maximization
- Wait time minimization
- Response time minimization
- Ensuring fairness

According to [8] a good scheduler should have the following properties:

- General purpose: a scheduling approach should make few assumptions about the applications and apply a few restrictions to the types of applications that can be executed.
- Efficiency: the perforamnce of scheduled jobs should be improved but at the same time, the scheduling shoud not add noticable overhead in the scheduling decision.
- Fairness: sharing resources among users raises new challenges in guaranteeing that each user obtains their fair share when demand is heavy.
- Dynamic: the algorithms employed to decide where to process a task should respond to load changes, and exploit the full extent of the resources available.

Scheduling algorithms can be divided into time sharing and space sharing ones. In time sharing, time is divided into distinct intervals, where each task will take one time slot. In space sharing algorithms, every task allocates its recources until the end of the execution. The algorithms we study are space sharing and we will further discuss about them in Chapter 5.

2.2.2 Batch Scheduling Systems

In HPC systems schedulers (the components that are responsible for job scheduling), manage the execution of parallel workloads across multiple nodes in a cluster. Schedulers in HPC handle large-scaled distributed systems with usually computationally intensive workloads. In this work we study batch scheduling algorithms, meaning that the jobs are submitted as a set (batch) and are not interactive. The steps of batch scheduling are the following:

1. Job Submission – Users submit jobs to a scheduler, specifying resource requirements.
2. Queue Management – Jobs are placed in a queue and prioritized based on scheduling policies.
3. Execution – The scheduler allocates resources and runs jobs based on available compute nodes.
4. Completion & Logging – Once a job finishes, the system logs results and frees resources for the next job.

We also mention some of the tools that handle job scheduling, resource allocation, and queue management in HPC clusters in a batch scheduling way:

- Simple Linux Utility for Resource Management (SLURM)
- Maui Cluster Scheduler (Maui)
- Moab High-Performance Computing Suite (Moab)
- TORQUE
- Portable Batch System (PBS) [OpenPBS, TORQUE, PBS Pro]

Most supercomputers in TOP500 (a list of the 500 most powerful non-distributed computer systems in the world [9]) use SLURM [10]. In SLURM the sceduling algorithm that is used is the Conservative Bacfilling with backfill depth of 100 [11].

2.2.3 Basic Notation

In order to make decisions that will better achieve the schedulers' goal, some characteristics of the job must be known. Some of these characteristics are parameters that are declared when a job is submitted (i.e. t_i^{wall}) and others vary between different experiment as the come up during execution. Here we present the basic notation for the parameters that are being used:

- t_i^{run} is the time job i needs for execution
- t_i^{wait} is the time job i spends in the waiting queue before executing
- t_i^{sub} is the time of submition of job i
- t_i^{wall} is the wall time user specified for job i
- n_i^{req} is the number of cores requested for the job i
- n_i^{alloc} is the number of cores allocated by job i
- t_i^{sub} is the time of submition of job i

2.3 Metrics

In order to quantify system performance when different scheduling algorithms are utilized, the usage of metrics is required. Metrics allow researchers to detect possible system bottlenecks and direct future improvements. In systems the main objectives that need to be optimized are energy consumption, resource utilization and user satisfaction. These objectives are conflicting, thus metrics measurements are necessary in order to achieve balance according to specific requirements. We will use relative metrics [12], meaning that in our evaluation we will decide on the 'best' scheduler based on a plethora of metrics. We divide the metrics used into two categories, target metrics and explanatory. We explore different alterations of widely used metrics such area weighted as proposed in [13] or simply weighted versions of metrics.

2.3.1 Target Metrics

In this work we analyze algorithms measuring two objectives, system performance and user satisfaction.

2.3.1.1 Makespan Speedup

The system performance metric we use is makespan speedup. Makespan expresses the time elapsed from the start of execution to the end (given a specific workload). Makespan speedup is the ratio of a baseline scheduler's makespan over the examined scheduler's makespan. The aim is to increase makespan speedup, as a reduced makespan allows more workloads to be executed in a bigger time slot. In this work we use the Conservative scheduler as a reference, which will be described further in Chapter 5. We can define makespan speedup mS as:

$$mS = \frac{max_i \left(t_i^{fin}\right)_{\text{Conservative}}}{max_j \left(t_j^{fin}\right)} \tag{2.1}$$

2.3.1.2 Mean Slowdown

Considering user satisfaction, we decided to measure the mean slowdown, also known as mean flow. Mean slowdown tries to provide a measure of fairness over applications, as it is expressed as the ratio of the time a job spent in the system over its real execution time [14]. The slowdown sld_i of job J_i is defined as:

$$sld_i = \frac{t_i^{run} + t_i^{wait}}{t_i^{run}} \tag{2.2}$$

And consequantly the mean:

$$\overline{sld} = \frac{1}{J} \cdot \sum_{i=1}^{J} sld_i \tag{2.3}$$

The goal is to minimize the mean slowdown of the jobs on the workload. As the writers of [14] note, jobs with small execution heavily impact the mean slowdown (while being a small portion of the actual workload computation wise), as a small relative change in the wait time could skyrocket the value of their slowdown. One solution is to use the bounded slowdown [15],

however we decided to include the metric as is. We conduct experiments with various workloads so we expect different values and between the metrics that consider user satisfaction, slowdown is better matching user expectations that a job's response time will be proportional to its running time [16]. Additionally, we study the standard deviation of the slowdown in order to ensure that slowdown is equally distibuted across jobs.

2.3.1.3 Mean Slowdown Per Processor

Alternatively to the mean slowdown, we decided to implement the mean slowdown per processor as mentioned in [17, 18]. The slowdown per processor $sldpp_i$ of job J_i is defined as:

$$sldpp_i = \frac{t_i^{run} + t_i^{wait}}{t_i^{run}} \cdot \frac{1}{n_i^{req}} \tag{2.4}$$

and the mean:

$$\overline{sldpp} = \frac{1}{J} \cdot \sum_{i=1}^{J} sldpp_i \tag{2.5}$$

The object of this metric is to present a more fair approach of mean slowdown, as the sensitivity of mean slowdown to small jobs is tackled.

2.3.2 Explanatory Metrics

In addition to the target metrics, we include four explanatory metrics to better understand the systems' behaviour.

2.3.2.1 Utilization

Sytem utilization is a metric that showcases how efficiently are computational resources used. Low utilization means that the resources are wasted. In this work we calculate the utilization of time period $[t_1, t_2]$ as:

$$U(t_1, t_2) = \frac{\int_{t_1}^{t_2} \rho(t)dt}{N(t_2 - t_1)} \tag{2.6}$$

where $\rho(t)$ is the number of jobs under execution at time t and N the number of system CPU-cores. We try to achieve high values of utilization (it varies from 0 to 1).

2.3.2.2 Mean Job Speedup

Another metric that expresses user satisfaction is the mean job speedup. Due to node sharing the execution time of each job can vary from the traditional compact execution. The speedup S_i of job J_i is calculated as

$$S_i = \frac{(t_i^{run})_{\text{compact}}}{t_i^{run}} \tag{2.7}$$

and the mean:

$$\bar{S} = \frac{1}{J} \cdot \sum_{i=1}^{J} S_i \tag{2.8}$$

It is obvious that each job wants to increase its speedup, therefore a higher value of mean speedup is considered success.

2.3.2.3 Weighted Mean Job Speedup

The mean speedup is a measure of improvement on job performance, however the contribution of each job is considered equal. In workloads the size (execution time and number of requested cores) of jobs can strongly vary, consequently we calculate a weighted version of speedup. In the weighted version each S_i is multiplied with a factor w_i that is the product of t_i^{run} and N_i. We denote Weighted Mean Job Speedup as:

$$WMJS = \frac{1}{\sum_i w_i} \sum_i w_i \cdot S_i \tag{2.9}$$

We consider that Weighted Mean Job Speedup is a more accurate metric for job improvement.

2.3.2.4 Slowdown Counts

In order to further explore the execution speedup of applications, we decided to count the percentage of jobs that did not achieve an execution speedup bigger than 0.99. This metric should not be confused with the slowdown calculated above, as it is simply the percentage of slowed applications.

Chapter 3

Co-scheduling

In this Chapter, we describe the co-scheduling technique. As mentioned earlier, most HPC systems are underutilized, due to fragmentation and resource contention [19–21]. The chosen method for treating this problem is node sharing.

3.1 How node sharing works

In node sharing, the resources of each node are not used by one application, but from a set of applications. The resource manager is allowed to assign different number of processors to different applications, resulting in them occupying more nodes than the expected (calculated as requested cores over cores per node). The different strategies for applications deployment are basically three (as shown in Figure 3.1):

- Compact
- Spread
- Striped

The resource manager can deploy an application as compact, spread it across many more nodes, or allocate it in nodes that another application is executing. We will further discuss that this co-execution can be beneficial for the system and even the individual applications.

3.2 Advantages of node sharing

When applications are spread, they take advantage of more nodes' resources like more last level cache and perhaps some accelerator like GPU. Also, the data transfer within the node is optimized, as less instances of the application are executed within the node and consequently less data are transferred (we consider MPI applications). Node sharing has a positive impact on applications that show opposite behaviour on resource intensity. For instance, a memory intensive application (an application that its performance is bounded by bandwidth speed when scaling) benefits when co-executing with a compute intensive application, as the communication needs are reduced. Thus, the execution time of said application can be reduced. Moreover, from the system's perspective, as applications are firstly spread and then striped in the system, the

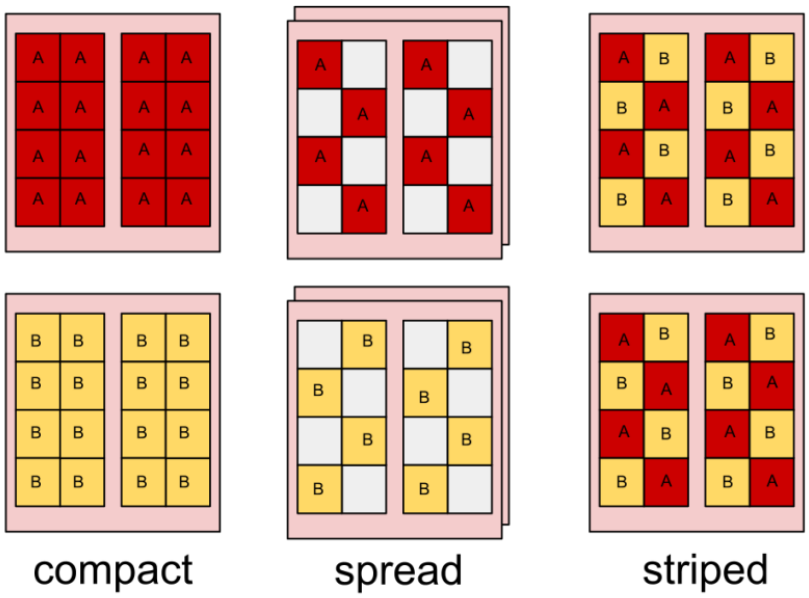


Figure 3.1: Methods for scheduling two 16 process distributed applications A and B on a supercomputer with two 8-core processor sockets [1].

fragmentation can be reduced and nodes that were previously not fully utilized can now be better used.

3.3 Disadvantages of node sharing

On the other hand, different applications have different communication patterns, so it is possible that a slowdown in execution time is observed. This may occur, due to the synchronization of application instances in different nodes, or simply that the colocated applications can be both memory intensive. It has been observed that when two jobs that run the same application but are distinct jobs (one cannot acess the data of the other) should not be co-executed. To elaborate, it is argued that if they are memory intensive there will be a slowdown in execution and if they are compute intensive, it would be more beneficial to be allocated with a memory intensive application [22, 23]. Another problem worth mentioning, is that in real systems there is a pricing scheme that values the cores used multiplied with a constant and the total execution time. Thus, it would be unfair to charge users more when their applications are slowed by the scheduling decision. Certain solutions have been proposed [24], however we will not consider this in our analysis.

3.4 Co-scheduling complexity

As we discussed above, node sharing can either have a positive or negative effect on applications' execution. Whether the load is omogenous or eterogenous heavily impacts the performance of co-execution. Especially the neighboring jobs play a crucial role in a job's execution time, as they can be the major reason for degradation [25]. Considering this, co-scheduling becomes a

really complex problem, as in order to tackle these challenges more knowledge of the applications submitted to the system is required. One could argue that the possible increment in execution time can be tolerated as it comes with a desirable decrease in wait time [26]. In this work, we will study the effect of co-scheduling and we will try to come up with co-scheduling algorithms that may not require extra knowledge about the applications and can work on every system.

Chapter 4

HPC Simulator

In this Chapter the simulation process is discussed, introducing ELiSE, a tool developed in Computing Systems Laboratory (CSLab).

4.1 Efficient Lightweight Scheduling Estimator (ELiSE)

Efficient Lightweight Scheduling Estimator (ELiSE), is an HPC scheduling estimator with co-location capabilities. ELiSE is a python-based framework that simulates the execution of the load on the cluster under different scheduling policies.

The required input consists of three components:

1. a simple description of a HPC cluster
2. a dynamic load of HPC jobs
3. a heatmap of pairwise speedups between the participating applications

The simulator resembles a finite state machine with jobs moving across four states as shown in Figure 4.1: The Future state contains jobs that have been generated according to user requests but their arrival time has not been reached by the simulator yet, Waiting contains jobs that have arrived and wait to be executed in the system, Executing are those that are currently running in the cluster, and Finished are those that have exited the system. The Scheduler decides which job to move from the waiting queue to the running state and which nodes/cores of the cluster to grant to the job. Further simulation logic described in the next sections orchestrates the rest of the process, i.e. handling future jobs, advancing time, calculating finishing times, etc.

The features it currently supports are:

- Workload generation
  To create a workload, users need to specify the number of applications (jobs) N, an application seed (a set of n applications with their execution times and compact allocation), and an n^2 co-execution matrix (heatmap) showing speedups for each application pair. ELiSE then generates a workload of N applications by selecting representatives from the seed randomly, by user-defined frequencies, or from a specific list. Users can also set the interarrival time of applications, choosing from constant, random, Poisson, or Weibull distributions.

- Describing the target platform
  To define the target platform, the framework only needs the number of nodes, CPUs per node, and cores per CPU.

- Available schedulers
  The framework currently supports several schedulers, including a basic FCFS scheduler, FCFS with EASY backfill, FCFS with conservative backfill, FCFS co-scheduler with EASY backfill, and a smart co-scheduler called Filler (detailed in Chapter 5). A key feature of ELiSE is its ease of extending to new schedulers.

- Output
  ELiSE generates a detailed execution log, capturing job arrival, start, and end times, among other metrics. It also offers various visualizations, such as Gantt charts, system throughput, queue size, system utilization over time, and boxplots of application speedups. Example outputs are illustrated in Figures 4.2, 4.3, 4.4 and 4.5.

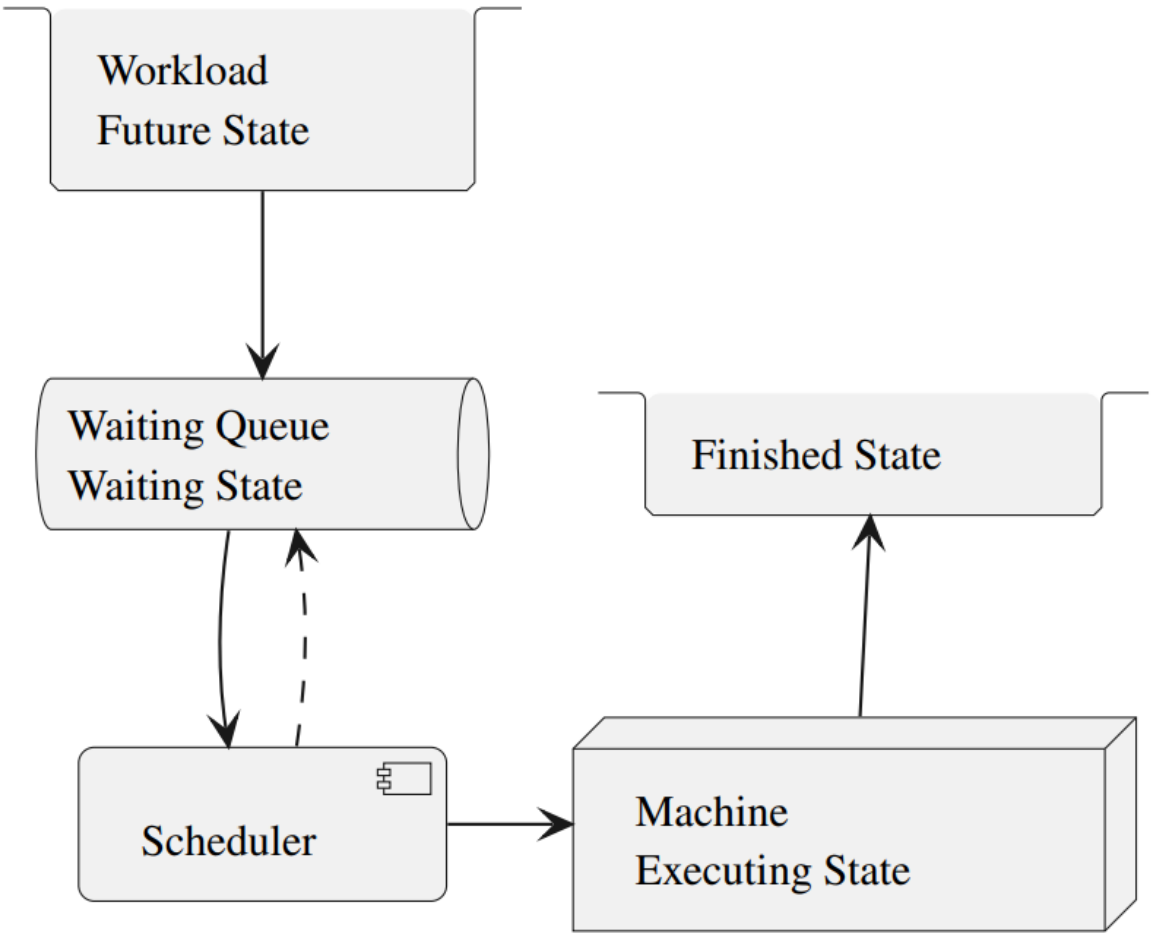


Figure 4.1: High-level design logic of the framework.

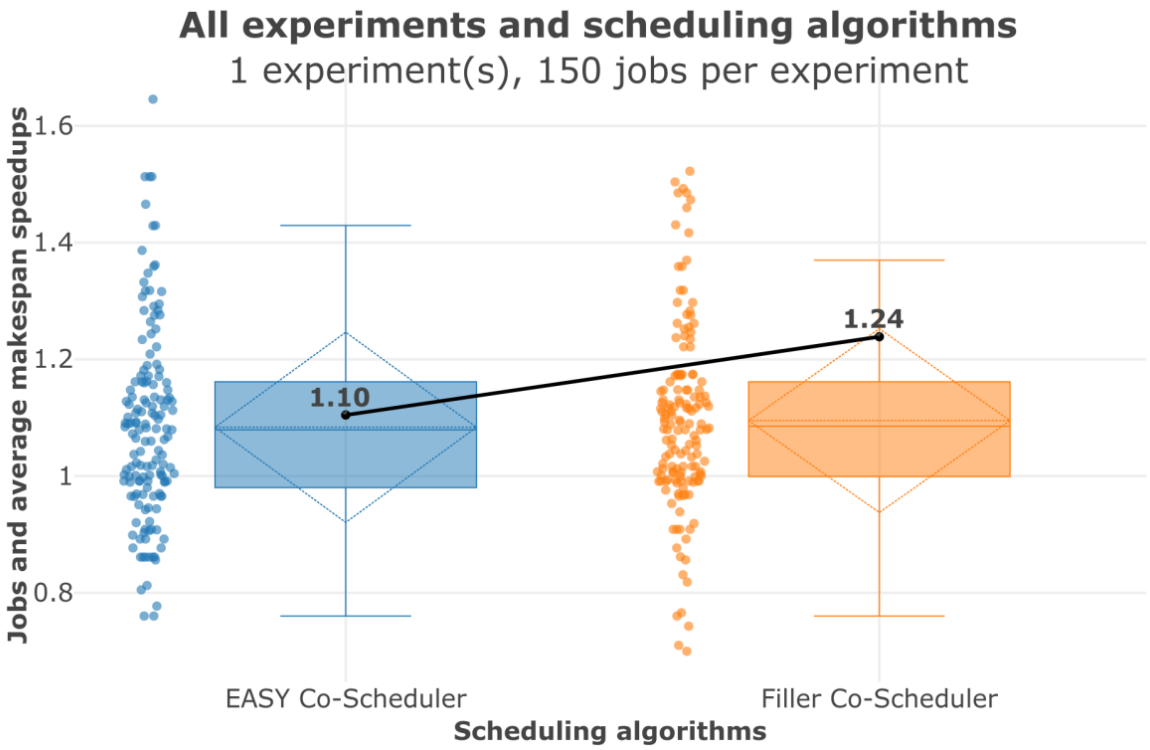


Figure 4.2: Example of job speedups and makespan speedup diagramm.

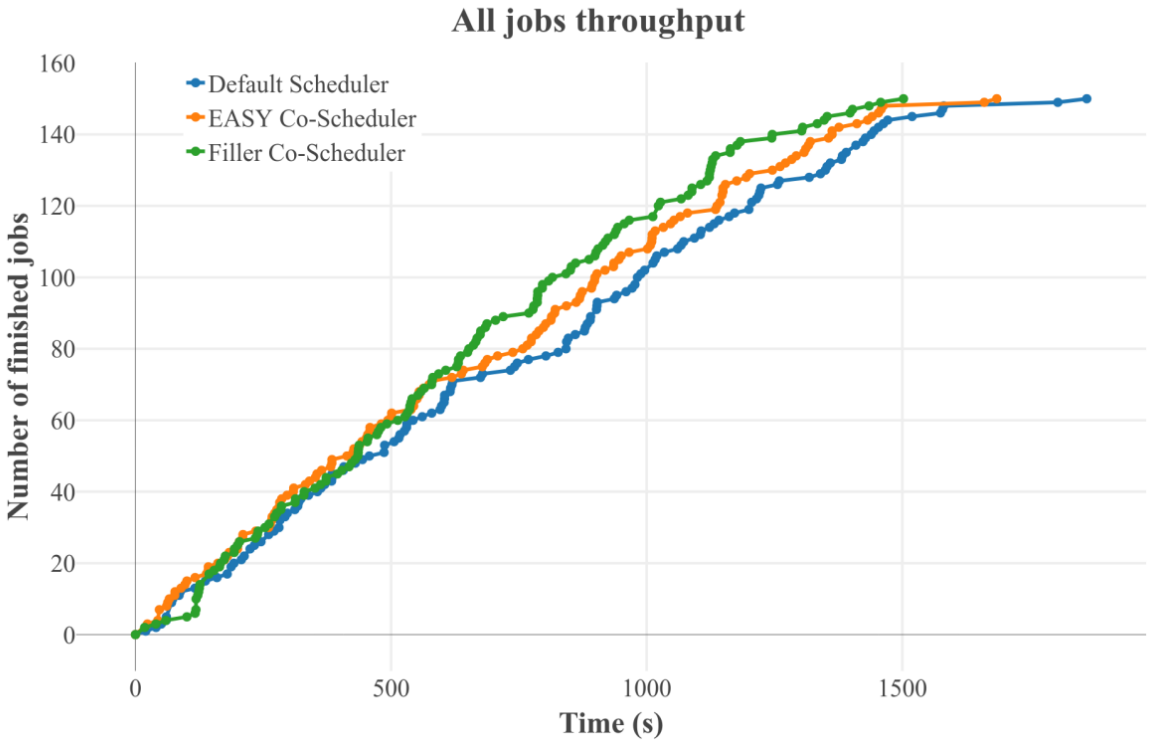


Figure 4.3: Example of jobs throughput diagramm.

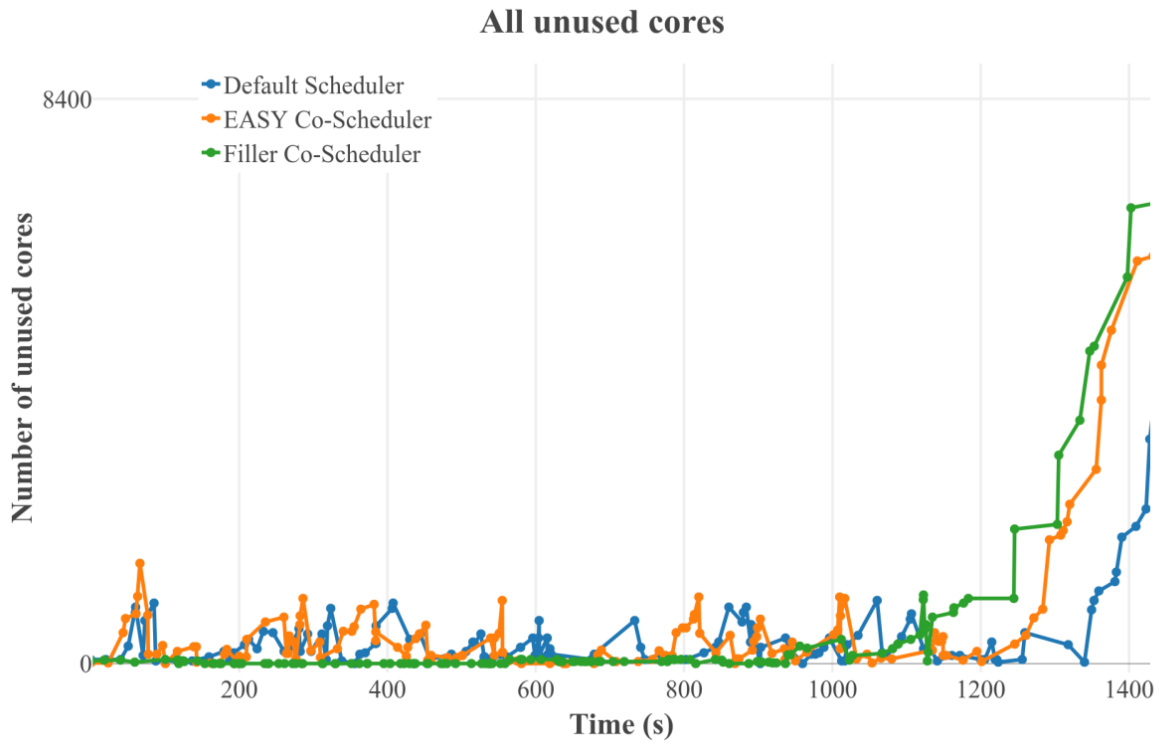


Figure 4.4: Example of unused cores diagramm.

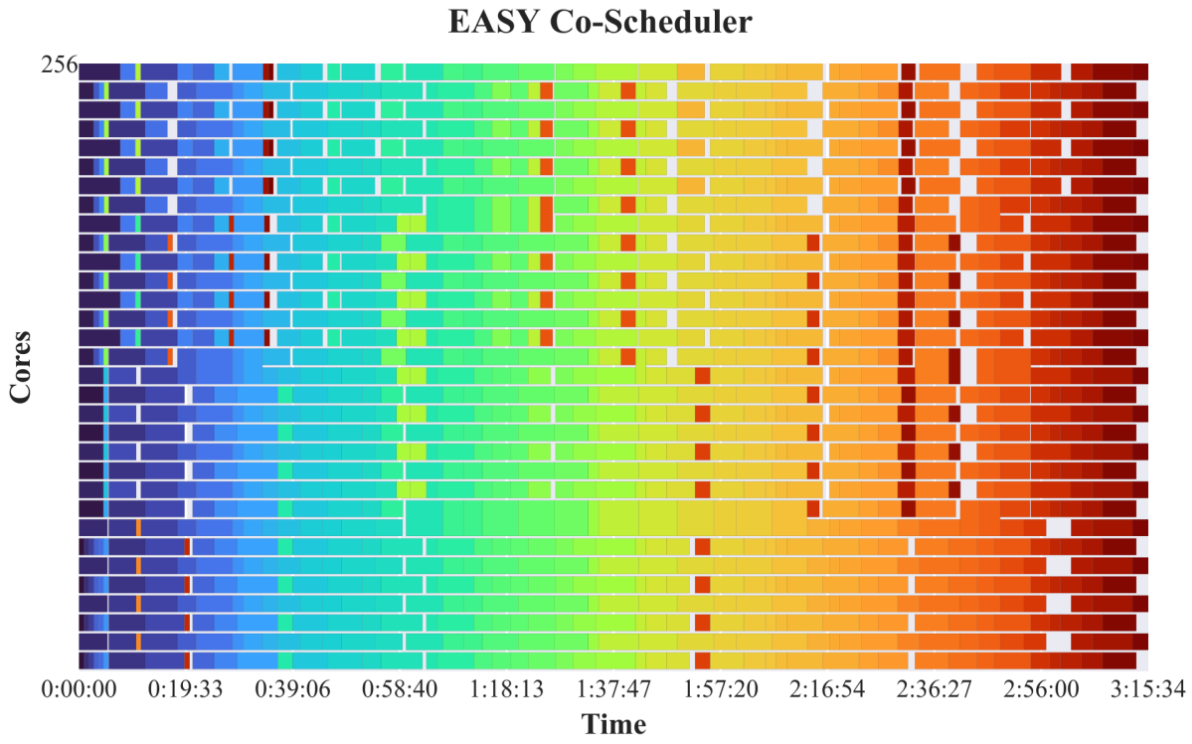


Figure 4.5: Example of a Gantt diagramm.

4.2 Heatmap

As described above, a heatmap of pairwise speedups between the applications is required. We will discuss about the creation of such heatmap and we will present the heatmap's usage. Below, a part of a heatmap file is provided, for applications run on the ARIS system (Table 4.1). In order to acquire this file, we run every co-execution scenario of the job pairs (including different process requirements). For each co-execution scenario, we paired applications on the same node and replicated this across all required nodes. Each pair ran together for 10 minutes, with any completed job being restarted. We recorded the median execution times from these repeated runs as the co-execution time for the benchmark. Speedups were calculated as the ratio of the original execution time (under compact allocation) to the co-execution time. Below, we present

Table 4.1: Part of a heatmap file from execution on the ARIS system.

name_A	procs_A	compact_A	name_B	procs_B	compact_B	co_A_B	co_B_A
bt.D.256	256	123.97	bt.D.256	256	123.97	119.62	119.40
bt.D.256	256	123.97	bt.D.512	484	60.04	121.29	57.94
bt.D.256	256	123.97	bt.D.64	64	486.25	138.96	443.08
bt.D.256	256	123.97	bt.E.1024	1024	511.22	120.39	492.30
bt.D.256	256	123.97	bt.E.2048	2025	224.97		
bt.D.256	256	123.97	cg.D.128	128	93.29	122.38	92.36
bt.D.256	256	123.97	cg.D.64	64	183.43	122.60	167.26
bt.D.256	256	123.97	cg.E.1024	1024	256.09	122.85	202.75
bt.D.256	256	123.97	cg.E.2048	2048	120.64		
bt.D.256	256	123.97	cg.E.512	512	359.30	123.81	270.04
bt.D.256	256	123.97	ep.E.256	256	145.935	114.22	145.55
bt.D.256	256	123.97	ep.E.512	512	72.89	114.25	73.43
bt.D.256	256	123.97	ft.D.256	256	55.98	126.44	41.20
bt.D.256	256	123.97	ft.D.64	64	185.72	139.93	145.63
bt.D.256	256	123.97	ft.E.1024	1024	136.65	127.82	98.73
bt.D.256	256	123.97	ft.E.2048	2048	77.31		
bt.D.256	256	123.97	ft.E.512	512	247.88	125.70	175.70
bt.D.256	256	123.97	is.E.256	256	85.21	128.49	63.40
bt.D.256	256	123.97	is.E.512	512	50.41	126.65	35.75
bt.D.256	256	123.97	lu.D.128	128	138.51	128.82	122.94
bt.D.256	256	123.97	lu.D.256	256	73.54	124.66	66.13
bt.D.256	256	123.97	lu.D.64	64	263.21	132.77	232.85
bt.D.256	256	123.97	lu.E.1024	1024	282.36	125.65	252.05
bt.D.256	256	123.97	lu.E.2048	2048	163.17		
bt.D.256	256	123.97	lu.E.512	512	542.87	126.09	482.46
bt.D.256	256	123.97	mg.E.128	128	159.37	162.00	103.03
bt.D.256	256	123.97	mg.E.256	256	87.53	156.55	54.04
bt.D.256	256	123.97	sp.C.64	64	20.12	135.31	17.00
bt.D.256	256	123.97	sp.D.128	128	318.87	158.57	210.34
bt.D.256	256	123.97	sp.D.256	256	116.32	138.40	87.07
bt.D.256	256	123.97	sp.D.512	484	58.49	133.78	48.36

the speedups of the pairs in different heatmaps, grouped by the requirements of the different processes (Figure 4.6).

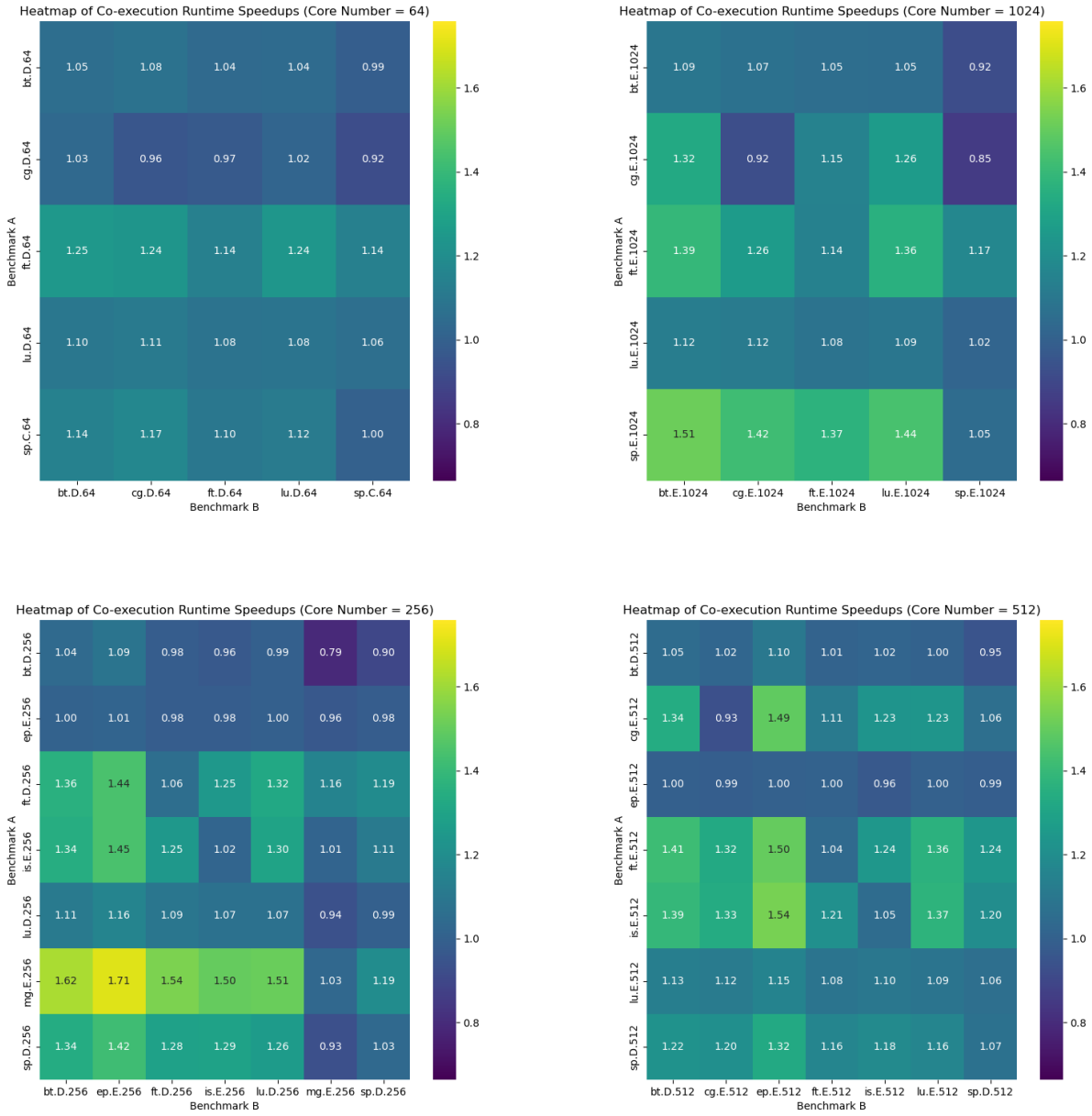


Figure 4.6: Heatmap of runtime speedups in co-execution for processes with specific core values.

The calculated runtime speedups are then used during runtime estimation. To be more specific, for the calculation of the remaining execution time of each job, we consider that the execution time of an application when concurrently co-located with other applications is determined by the slowest case, since processes in HPC applications typically interact with each other in time steps, and the slowest process determines the pace of the entire execution. The speedup for each co-execution interval of job J_i with neighbors J_n is calculated by:

$$S_i^{new} = \min_{\forall j \in J_n} getSpeedupWith(j) \tag{4.1}$$

Consequently, the new remaining execution time is calculated as:

$$\left(t_i^{run}\right)_{new} = \frac{S_i^{old} \cdot \left(t_i^{run}\right)_{old}}{S_i^{new}} \tag{4.2}$$

4.3 NAS Parallel Benchmarks (NPB)

The workloads used in the simulator are taken from the set of NAS Parallel Benchmarks (NPB). NPB were developed by the NASA Advanced Supercomputing (NAS) division to provide a standardized way to measure the performance of parallel computing systems [27]. The set consists of a series of computational kernels and pseudo-applications that mimic the workload of real-world scientific and engineering simulations, derived form CFD applications. The jobs we use consist of:

- 5 computation kernels
  - IS - Integer Sort, random memory access
  - EP - Embarrassingly Parallel
  - CG - Conjugate Gradient, irregular memory access and communication
  - MG - Multi-Grid on a sequence of meshes, long- and short-distance communication, memory intensive
  - FT - discrete 3D fast Fourier Transform, all-to-all communication
- 3 pseudo applications
  - BT - Block Tri-diagonal solver
  - SP - Scalar Penta-diagonal solver
  - LU - Lower-Upper Gauss-Seidel solver

The jobs we use are also divided in two classes D and E, that denote the size of test problems (Table 4.2). The job has another element which is the number of requested cores, that is a power of 2. To conclude, the format of a job is
`{IS,EP,CG,MG,FT,BT,SP,LU}.{D,E}.{64,128,256,512,1024,2048}`.

Table 4.2: Problem size of each class.

Benchmark	Parameter	Class D	Class E
CG	no. of rows	1500000	9000000
	no. of nonzeros	21	26
	no. of iterations	100	100
	eigenvalue shift	500	1500
EP	no. of random-number pairs	236	240
FT	grid size	2048 ×1024 ×1024	4096 ×2048 ×2048
	no. of iterations	25	25
IS	no. of keys	231	235
	key max. value	227	231
MG	grid size	1024 ×1024 ×1024	2048 ×2048 ×2048
	no. of iterations	50	50
BT	grid size	408 ×408 ×408	1020 x 1020 x 1020
	no. of iterations	250	250
	time step	0.00002	0.000004
LU	grid size	408 ×408 ×408	1020 ×1020 ×1020
	no. of iterations	300	300
	time step	1.0	0.5
SP	grid size	408 ×408 ×408	1020 ×1020 ×1020
	no. of iterations	500	500
	time step	0.0003	0.0001

Chapter 5

(Co-)scheduling Algorithms

In this Chapter we describe the functionality of the algorithms studied in this Thesis. We present both common scheduling algorithms used in clusters and systems in general such as First Come First Serve (FCFS) and algorithms supporting job striping, the so called co-scheduling algorithms. In co-scheduling algorithms apart from our own implementations, we include some traditional algorithms performing job striping like Shortest Job First (SJF).

5.1 Traditional Scheduling

In traditional scheduling after a job has been submitted a possible reordering in the waiting queue occurs and then the head of the waiting queue is selected and the required resources are allocated. The allocation we study is about computing nodes, where continous nodes are assigned to the job (the number of which is the least amount of nodes that contain the requested cores).

5.1.1 First Come First Serve (FCFS)

This might be the most common scheduling algorithm, where no reordering on the waiting queue occurs and jobs are selected based on their arrival order. FCFS is really simple and does not require any computations during execution as there is no reordering. This is considered a fair scheme as there are no starvation instances, however the system becomes prone to fragmentation. In addition, another disadvantage of FCFS is the convoy effect, where a long job delays many shorter (in clusters we can imagine a number of long jobs, or a long job that requires many nodes), resulting in grater wait times.

5.1.2 Extensible Argonne Scheduling sYstem (EASY)

The Extensible Argonne Scheduling sYstem [28] (EASY) algorithm is based on the FCFS algorithm with the improvement of backfill, an improvement that treats the convoy effect and improves the utilization.

Backfill is the procedure where one or more jobs ”skip the line” and execute before their expected time, as long as it does not affect the execution of the previous jobs. There are two main backfilling policies as described in [29], the aggressive and conservative. The difference of these backfilling policies is that in conservative backfill, every job is given a reservation when

it enters the system. A smaller job is moved forward in the queue as long as it does not delay any previously queued job. In aggressive backfilling, only the job at the head of the queue has a reservation. A small job is allowed to leap forward as long as it does not delay the job at the head of the queue.

We present an example of backfill in order to demonstrate its function. Here the waiting queue has three jobs D, E and F (Figure 5.1).

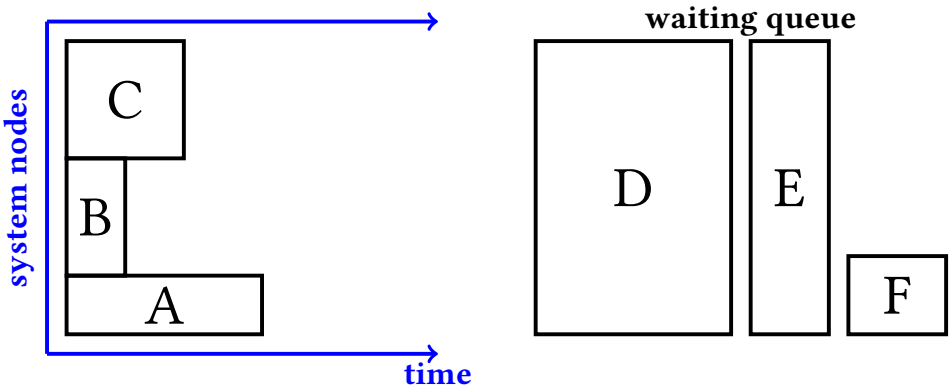


Figure 5.1: Example of backfill policy-initial waiting queue.

Job D is selected as the top job of the waiting queue. As we can observe the execution of job D must wait until job A finishes execution (Figure 5.2). Then the waiting queue is accessed again.

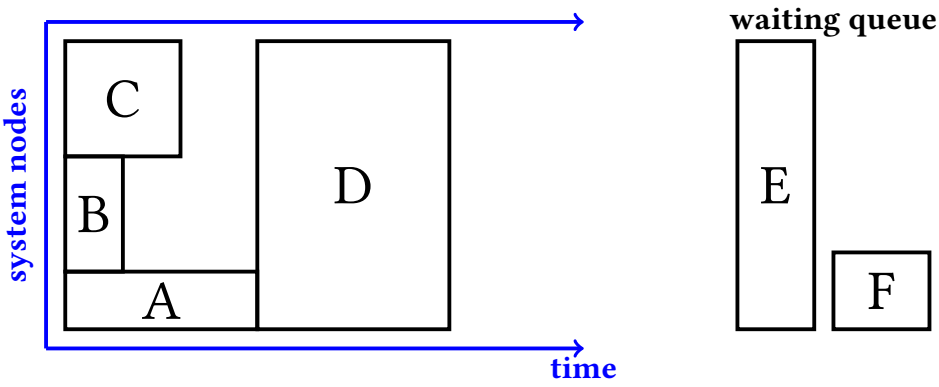


Figure 5.2: Example of backfill policy-allocation of job D.

Job E cannot execute before Job D as it requires nodes that if allocated, job D would have to delay its execution. We continue to the next Job in the waiting queue, which is Job F. If deployed Job F does not affect the execution of D so it selected and executed (Figure 5.3). The process can continue until all jobs in the waiting queue are checked, a certain number of jobs is checked or even stop at the first find. The implementation of the policy can vary according to one of the three options. In our implementation, every job is checked up to a certain index, called backfill depth.

To conclude the process of EASY scheduling is the following:

1. A job is submitted to the waiting queue
2. The head of the waiting queue is selected and set for execution

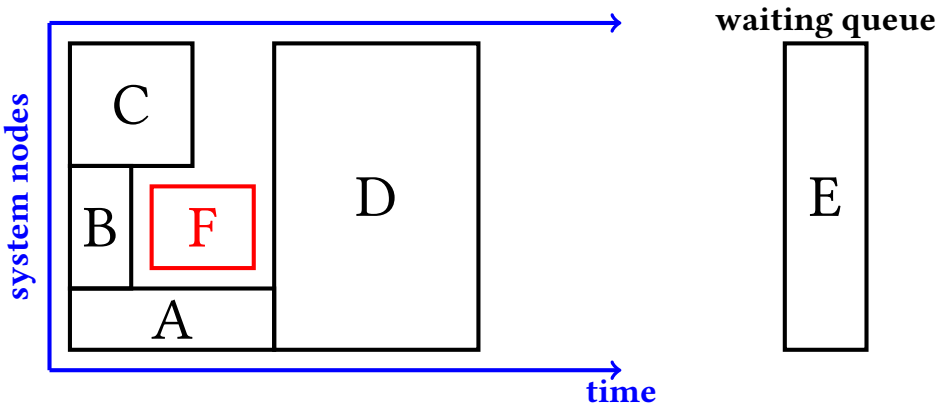


Figure 5.3: Example of backfill policy-allocation of job F (backfill).

3. The waiting queue is again accessed and every job that can be executed without messing with the beggining of head's job execution is executed

The EASY algorithm performs better than the FCFS both in terms of system efficiency and user sastisfaction, as the gaps both in time and space angle are filled. On the contrary, EASY has a higher complexity, as the waiting queue is traversed every time a job is set for execution.

Apart from the EASY algorithm there can be many variants that perform aggressive backfilling and combine other policies. For instance, we mention the FCFS-SJF as presented in [30], where the shortest job is backfilled first. In real systems, this approach may suffer from bad estimations in jobs' runtimes.

5.1.3 Conservative

Another popular scheduling policy is the Conservative Backfilling. In Conservative Backfilling, as in EASY a FCFS ordering in the waiting queue is kept. As mentioned before, in Conservative Backfilling the originally estimated start time of each job in the waiting queue must be respected, unlike EASY. This approach to backfill is considered a fairer one, preventing starvation at the same time. There are a few alterations of this policy as mentioned in [31], but we will implement the common one. Conservative algorithm can be considered more complex than EASY as more jobs have to be checked in each backfilling step.

5.1.4 Shortest Job First (SJF)

Shortest Job First (SJF) is another common scheduling algorithm. Here the jobs in the waiting queue are sorted based on their required execution time (in this work we use the remaining execution time as we assume it is known, however in a real system the wall time would replace it). As mentioned in [32], this simple policy or perhaps a similar one being Shortest Area First (SAF), can drastically outperform the EASY algorithm. We decided to only study SJF as we use this algorithm to set a baseline on slowdown values. This algorithm favors shortest jobs over longer ones and results in smaller values of slowdown and wait time in general. The disadvantages of this algorithm are that starvation instances of long jobs appear and that in real applications the

wall time may vary from the real execution time. In addition, system fragmentation may also occur as we do not consider the number of requested nodes.

5.1.5 Longest Job First (LJF)

Similarly to SJF, Longest Job First (LJF) is a scheduling algorithm that reorders the waiting queue based on the jobs' required execution time. We would expect a major decrease in mean slowdown values as the wait time of shorter jobs in increased. Considering system performance, we would expect high throughput and perhaps fragmentation that could be fixed with backfill.

5.2 Co-scheduling

The other approach scheduling algorithms follow is the one of node sharing, where as previously described, more than one jobs can allocate cores in the same node, in order to eliminate resource contention. We present algorithms that follow the same principles as some of the traditional scheduling algorithms and are modified for colocation, as well as our own implementations. All algorithms perform co-allocation in the same way, that is described below. The main focus of the algorithms presented is the reordering of the waiting queue, exactly as in the traditional scheduling schemes.

5.2.1 EASY Co-schedule

Here the same procedure as EASY is followed as the job are deployed in the order in which they arrive, however the jobs are firstly deployed as spread (as long as the required space is available) and then as the system begins to fill the system, the jobs allocate nodes that are already half full. The nodes are assigned to jobs in a serial way. We expect similar behaviour with the traditional EASY, with perhaps an improvement in utilization, as more than jobs are allowed in the same node, covering unused cores that may exist in the traditional approach.

5.2.2 Shortest Job First Co-shedule (SJF-Co)

Like sized based approaches, here we try to maximize the algorithm's performance considering only the slowdown. This algorithm reorders the waiting queue, giving priority to the shortest jobs. The same comments as the SJF apply.

5.2.3 Longest Job First Co-shedule (LJF-Co)

Similarly to SJF-Co, in Longest Job First Co-shedule (LJF-Co) we study the effect of node sharing in LJF. We expect less fragmentation and as a result higher makespan speedup. As in LJF, we expect high mean slowdown values.

5.2.4 Largest Area First Co-schedule (LAF-Co)

Next, we try an approach that reorders jobs based on their size. In particular, the waiting queue is sorted so the jobs with the grater product of requested cores and execution time are deployed

first, while executing aggresive backfill as well. We should note that this algorithms clearly discriminates against smaller jobs and consequently we expect big values in mean slowdown. Our goal with this implementation is to set some extreme values for reference, and as we evaluate algorithms on two factors we try to optimize one of them; here being system utilization. We expect high utilization values, as the bigger jobs fill the system's nodes and backfill takes care of possible gaps. In addition node sharing allows more agile allocations, thus improving utilization.

5.2.5 Popularity

This approach differs from the previous ones, as we try to take advantage of the information provided by the heatmap. Each time a scheduling desicion has to be made (a job is going to be deployed), the waiting queue is sorted, based on a popularity mesure that we call "rank". Rank is calculated for each job and it is a counter for the number of successful co-allocations the job can have with the rest of the waiting queue (according to the heatmap). To be more precise, the waiting queue is traversed in two for loops where when the average speedup of two jobs is grater than the value of a threashold, both jobs' rank is increased. The jobs with a higher rank are given a higher priority , as they are friendlier for co-execution. However, the least friendly jobs are pushed to the end of execution, with possibly negative results in execution times. In order to prevent this, we deploy the last jobs in the waiting queue as compact.

5.2.6 Filler

This is a scheduling algorithm designed by Stratos Karapanagiotis while he was part of the CSLab team, which aims to improve system fragmentation. The waiting queue is sorted like in every other algorithm, based on the ratio of two factors. The first factor (nominator) is about the gap the jobs are going to create in the system. For each job in the waiting queue the difference of system's free cores and the number of processes required by the job is calculated (as long as there are free system cores). This difference is then divided by the number of system's free cores and then is subtracted from 1 and that is the value of the first factor (unless the job cannot fit into the system, where the value is set as -1). So grater difference values result into smaller arithmetic values for the first factor. It is clear that the first factor favours the jobs that will leave the smaller gap in the system (in this particular moment according to the system's state). The second factor (denominator) is the job's id increased by one, divided by the length of the waiting queue. As the job id is increased according to the submission time of each job, meaning that a job with a higher id must have a latter or at least the same submition time than that of a job with a smaller id. This factor tries to preserve the order in which jobs arrived and insure fairness among the jobs. We expect high utilization values, as the jobs are prioritized when leaving smaller gaps in the system (without meaning that the resulting arrangement will be the optimal for the system). Simultaneously, we expect small wait times and consequently small slowdown values, as an arrival order is partially preserved.

5.2.7 Two Factors

This approach is heavily based on Filler. Again, the sorting of waiting queue is based on two factors, where the first one is exactly the same as the first factor of Filler. We decided to keep the mechanism of the first factor, because the system's current situation is considered. Here the two factors are added and currently they have the same contribution. In this implementation we create a copy of the waiting queue and we sort this queue based on a different criterion. After the second queue is sorted we locate the current job's position in the second queue and divide it with the length of the waiting queue. This value is the second factor. The complexity of this algorithm is increased as for each job there has to be a whole reordering. Based on the sorting criterion of the copy queue, we have created two variants of the algorithm:

- SJF-Filler
- Pop-Filler

Both of these variants use information about the jobs that are not provided in the job submission, but require further knowledge of the workload. We also study the case where only a part of the waiting queue is sorted in order to reduce the complexity and ensure fairness among the jobs.

5.2.7.1 Shortest Remaining Time Criterion (SJF-Filler)

This is the original variant of Two Factors algorithm. The second sorting criterion is the shortest remaining execution time. This algorithm combines the logic of SJF in order to reduce slowdown values, while at the same time the system's state is considered.

5.2.7.2 Popularity Criterion (Pop-Filler)

This variant uses as a second factor the popularity as mentioned before. In this approach we try to preserve the filling mechanism described in the Filler algorithm and at the same time increase the speedup of execution time.

Chapter 6

Evaluation

In this Chapter we will present and discuss about the experimental results of our study. We run 10 experiments of 1000 jobs each on ELiSE, simulating the ARIS system (420 nodes with 20 cores each). We chose 1000 jobs per experiment, as there co-exist several instances of the different benchmarks and the system is in full load.

6.1 Mean Values

First, we present the mean values of the metrics in these 10 experiments (Tables 6.1, 6.2).

scheduler	mean slowdown	mean slowdown pp	slowdown (%)	utilization
SJF	**20.6**	**0.101**	0.00	0.947
SJF-Co	23.6	0.118	17.51	0.875
SJF-Filler	27.7	0.109	22.94	0.964
Filler	48.1	0.189	29.70	0.965
Pop-Filler	63.5	0.265	23.78	**0.971**
Popularity	64.3	0.308	**10.10**	0.932
EASY	66.5	0.372	0.00	0.926
EASY-Co	67.5	0.368	23.82	0.904
Conservative	80.2	0.463	0.00	0.922
FCFS	82.9	0.480	0.00	0.899
LJF-Co	109.6	0.599	23.62	0.939
LAF-Co	125.7	0.745	21.59	0.937
LJF	137.2	0.812	0.00	0.960

Table 6.1: Mean values of metrics for 10 experiments (1000 jobs on ARIS) - part 1.

As we expected, SJF has the smallest mean slowdown values, followed by SJF-Co. Here we observe that co-sceduling worsens the response time of individual jobs. The mean job speedup in the co-scheduled is increased, despite the fact that about 18% of jobs are slowed down. On the other hand, the utilization and makespan speedup of the co-scheduled version is lower, possibly due to an increase in execution time of certain jobs.

The SJF-Filler has similar mean value of mean slowdown to those of the other SJF approaches, while achieving high utilization values and reaching the highest makespan speedup. The other

scheduler	mean job speedup	weighted mean job speedup	makespan speedup
SJF	1.000	1.000	1.028
SJF-Co	1.062	1.049	0.995
SJF-Filler	1.074	1.055	**1.104**
Filler	1.065	1.040	1.086
Pop-Filler	1.077	1.038	1.093
Popularity	1.047	1.020	1.029
EASY	1.000	1.000	1.006
EASY-Co	**1.097**	**1.060**	1.040
Conservative	1.000	1.000	1.000
FCFS	1.000	1.000	0.976
LJF-Co	1.083	1.042	1.063
LAF-Co	1.072	1.049	1.065
LJF	1.000	1.000	1.043

Table 6.2: Mean values of metrics for 10 experiments (1000 jobs on ARIS) - part 2.

two Filler implementations behave similarly, with the Pop-Filler combination achieving the highest utilization.

Popularity has the lowest percentage of slowed down jobs (except from the traditional schedulers), as it includes compact execution (hybrid allocation policy). Compared with the more compact approaches, the mean slowdown values are lower.

In EASY scheduling, we observe that co-scheduling has a positive effect in mean job speedup, achieving the highest value and in mean job speedup speedup as well. We also observe a small decrease in utilization.

Now comparing EASY, Conservative and FCFS, we see that utilization is decreased slightly in this order and mean slowdown is increased, again in this order. These results perfectly align with the expected effects of backfill. Moreover, in LJF-Co, LAF-Co and LJF the highest values of mean slowdown occur. These implementations also perform at high utilization and makespan speedup values.

6.2 Boxplots

Here we examine the performance on each metric seperately, using the following boxplots.

6.2.1 Makespan Speedup

Firstly, we focus on makespan speedup and mean slowdown, as these are the selected target metrics. The highest values of makespan speedup occur with SJF-Filler, followed by the other two approaches. Moreover, LJF-Co and LAF-Co have high makespan speedups as well. The schedulers with mean makespan speedup less than 1 are SJF-Co, Popularity and FCFS. Probably in SJF the co-scheduling worsens fragmentation.

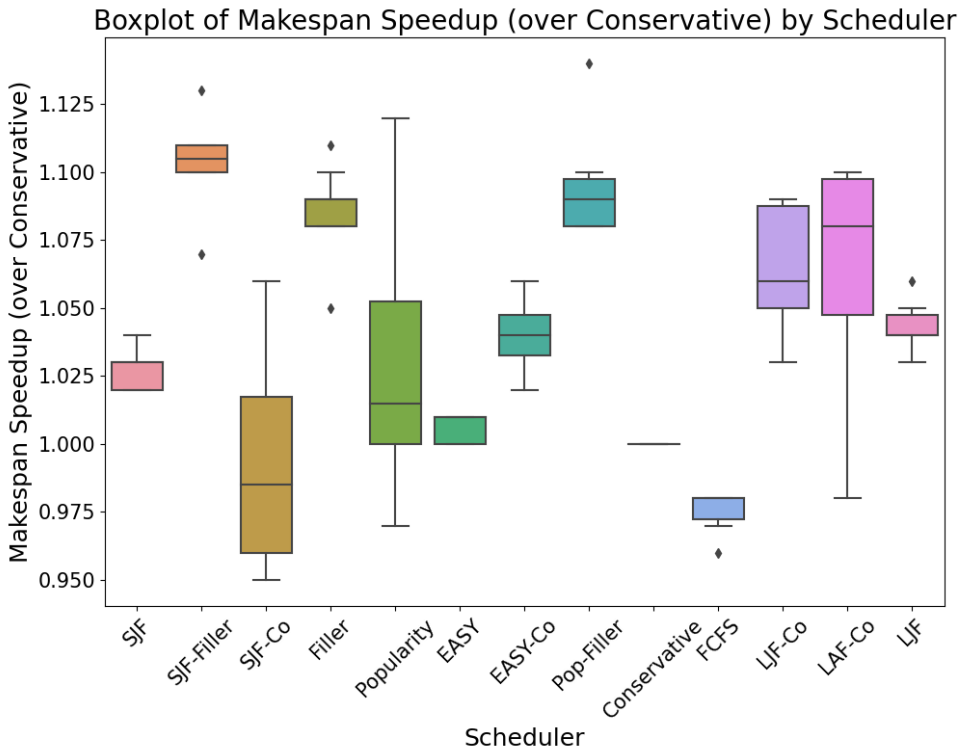


Figure 6.1: Boxplot of Makespan Speedups.

6.2.2 Mean Slowdown

With regards to mean slowdown, the three SJF implementations have considerably lower values. The Filler algorithm stands somehow in the middle between the rest. Also, the ones that value long execution time, have bigger values as expected.

6.2.3 Mean Job Speedup

The highest mean speedup values are observed in co-scheduled EASY, with a small variance. The values in the rest schedulers do not seem to differ, with perhaps the grater variance on SJF-Filler and Pop-Filler. Also, LJF-Co appears to achieve high values and Popularity lower ones. The behaviour of Popularity can be interpreted as a result of the compact deployment of certain jobs, keeping the mean lower. Here we present one example of the distribution of speedup of individual jobs during the execution of one experiment (Figure 6.4). Here the red marks represent the mean speedup considering only the jobs in each time bin and the black dotted line the cumulative mean. It is visible that the mean speedup start from high values as the first jobs are co-execution friendly and it gradually reduces as the non friendly jobs stay behind. On the other hand, the results on Fillers indicate that mean job speedup greatly varies between different executions.

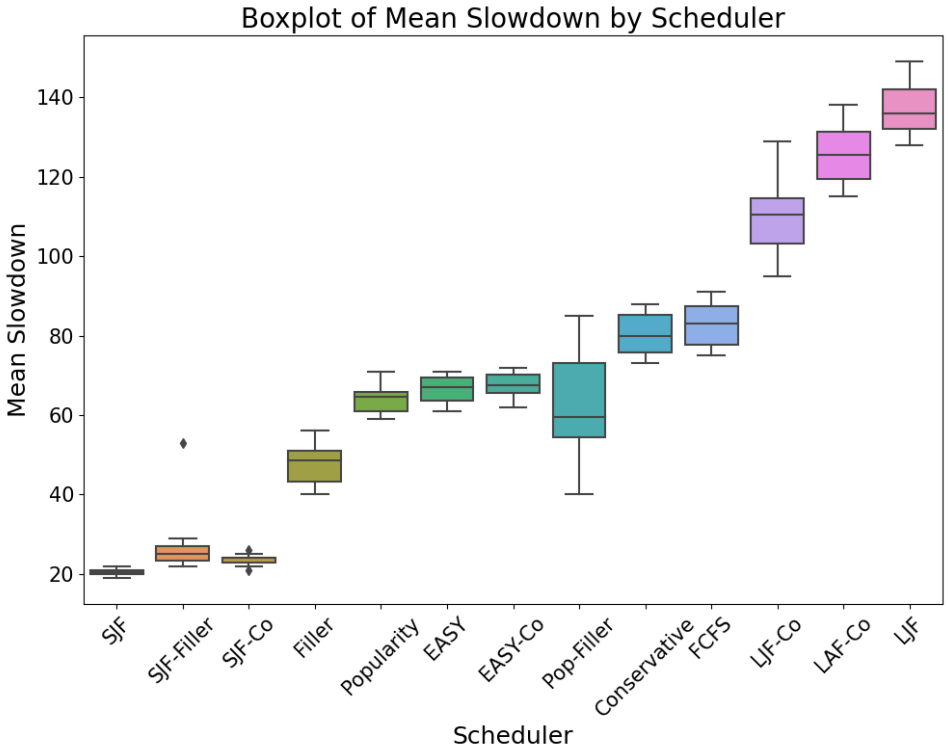


Figure 6.2: Boxplot of Mean Slowdown values.

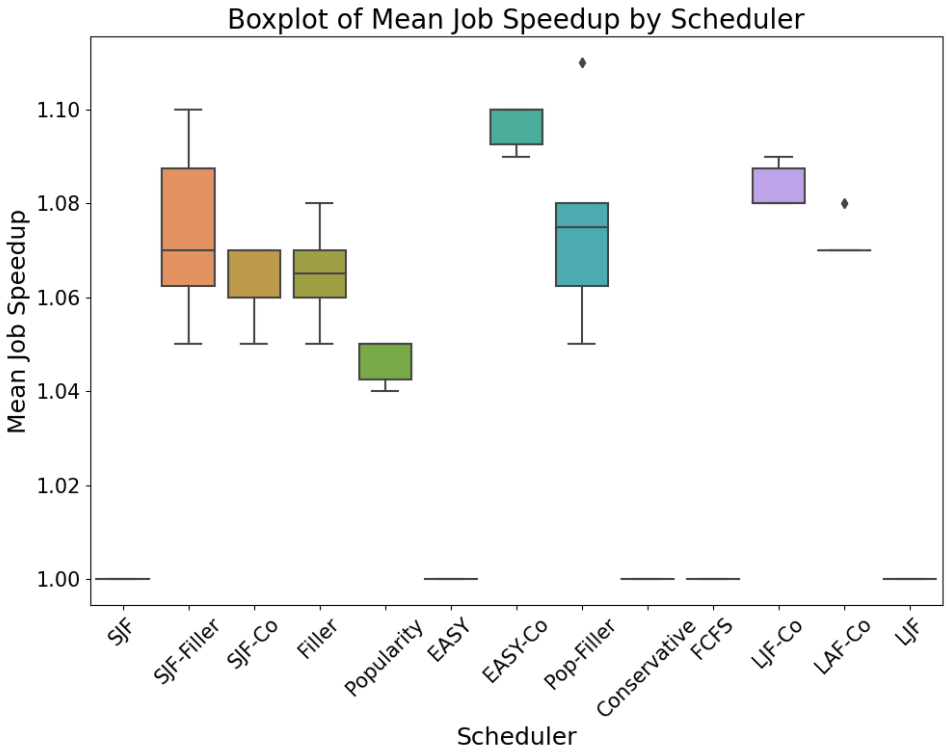


Figure 6.3: Boxplot of Mean Job Speedups.

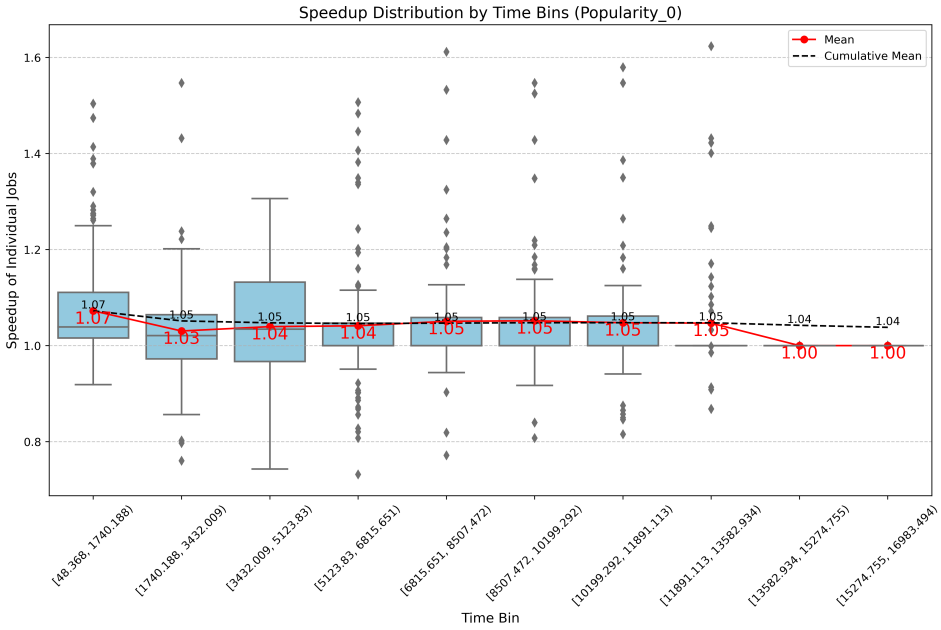


Figure 6.4: Time distribution of mean job speedup in a Popularity scheduling execution.

6.2.4 Weighted Mean Job Speedup

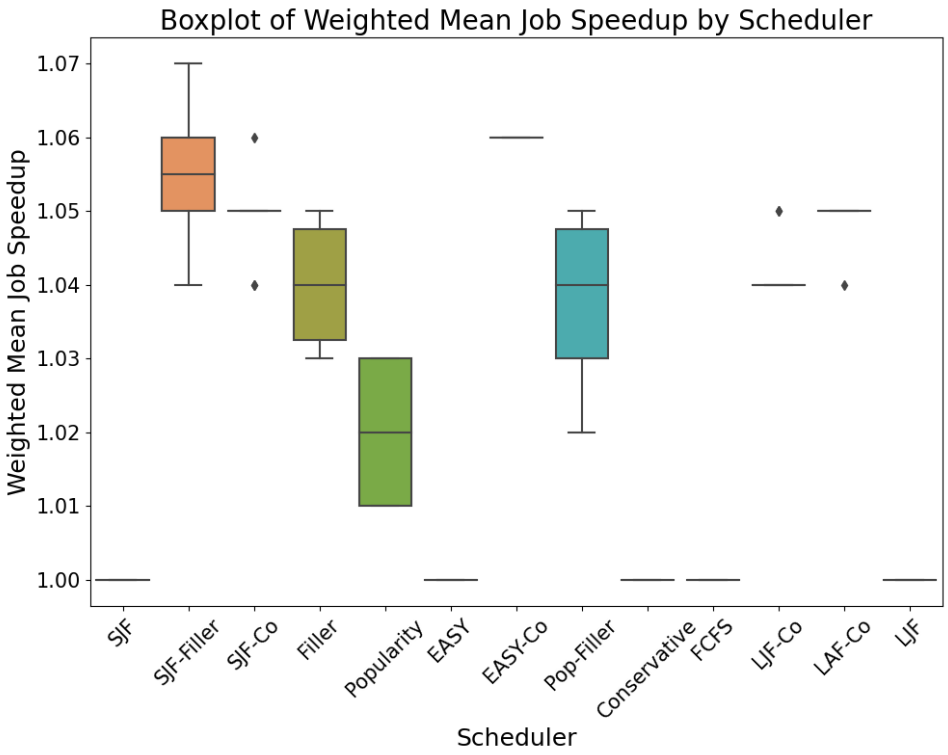


Figure 6.5: Boxplot of Weighted Mean Job Speedups.

In this plot the larger jobs have a bigger impact on the values. The values of weighted mean speedup seem to be smaller than the unweighted version, indicating that smaller jobs are favored (possibly due to backfill). We also observe that schedulers that treat large jobs in a consistent way (either push them to the front or push them back), could lead to stable results in the weighted mean.

6.2.5 Mean Slowdown per Processor

The same observations as slowdowns apply, with a slight difference in Pop-Filler, where the values are actually smaller relatively.

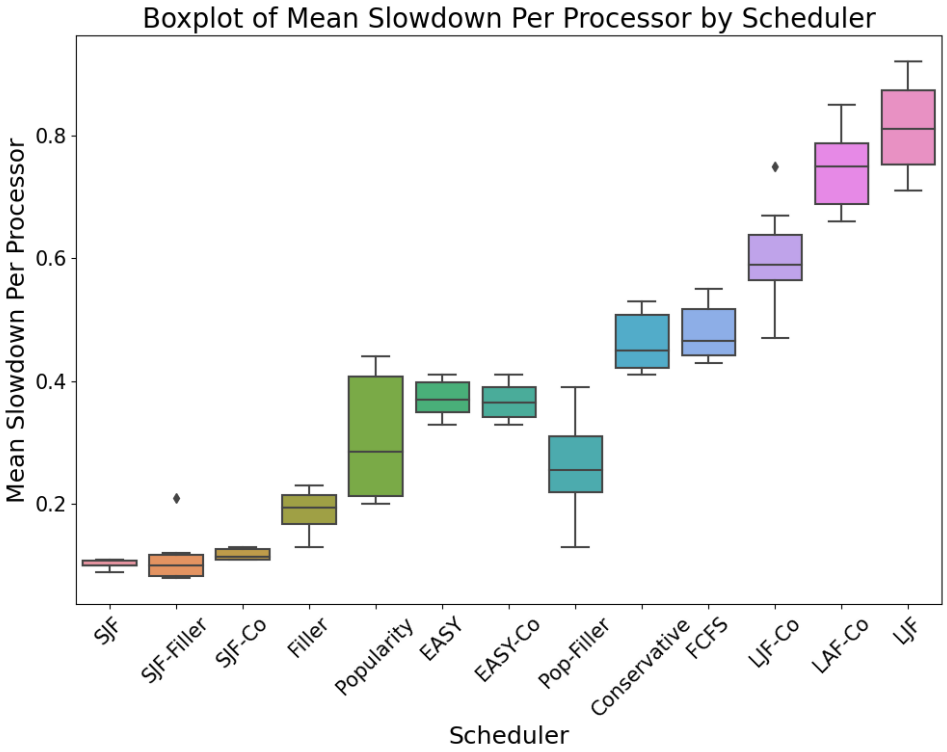


Figure 6.6: Boxplot of Mean Slowdown per Processors values.

6.2.6 Slowdown Counts Percentage

When we examine the slowdown counts percentage, we observe that most schedulers' values are around 23%, except the Filler and Popularity and of course the traditional schedulers. We can explain the lower values in Popularity, as it combines striped and spread deployment with compact as well. In the Filler case, we cannot fully interpret this behaviour, as the mean job speedup is similar to the other schedulers'.

6.2.7 Utilization

In terms of utilization, the results are mostly expected. The three Filler approaches dominate, as they actively try to reduce fragmentation on each scheduling decision. Their performance is followed by the "Large" co-schedulers (LJF, LJF-Co and LAF-Co), also expected. Interestingly, the utilization of SJF-Co and EASY-Co is worse than the one of their compact scheduling. We suspect that these approaches create more smaller gaps in the system, which cannot be filled during backfill, thus resulting in lower utilization. We present an example of an SJF execution where the makespan of the co-scheduled version (Figure 6.10) is grater than the traditional (Figure 6.9). We can see that the gaps are getting bigger as the jobs that create them expand in more nodes and backfilling cannot fill them.

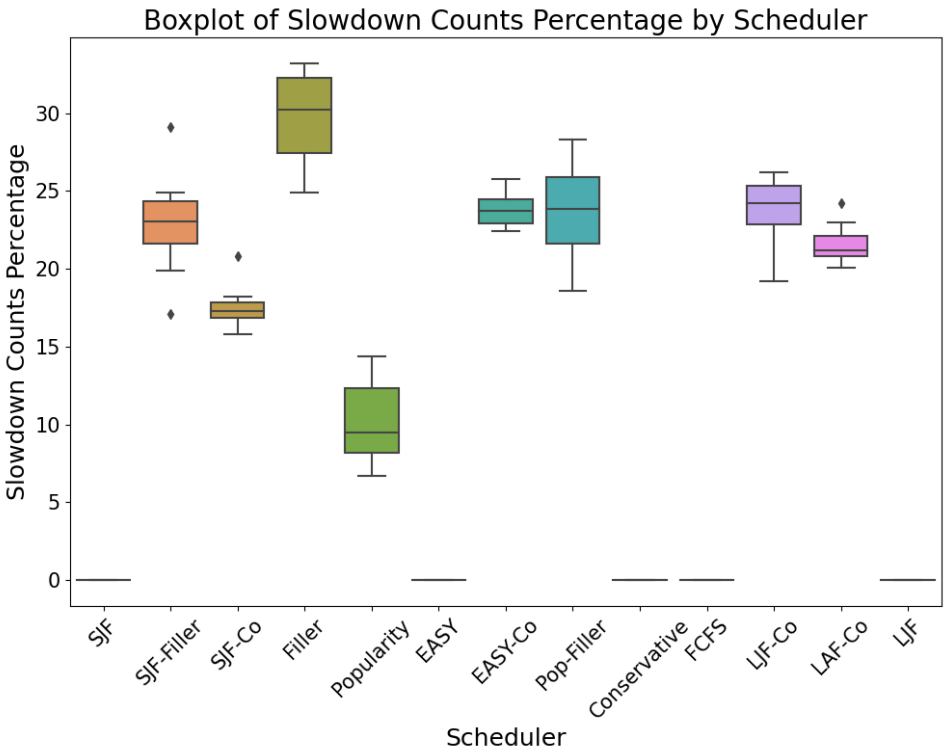


Figure 6.7: Boxplot of Slowdown Counts Percentage values.

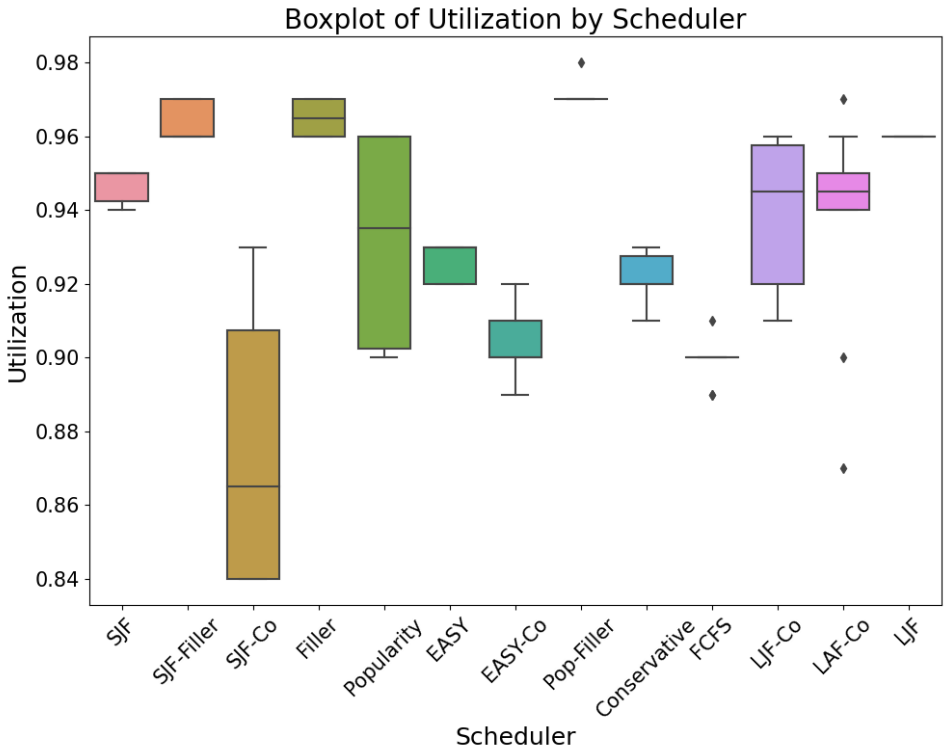


Figure 6.8: Boxplot of Utilization values.

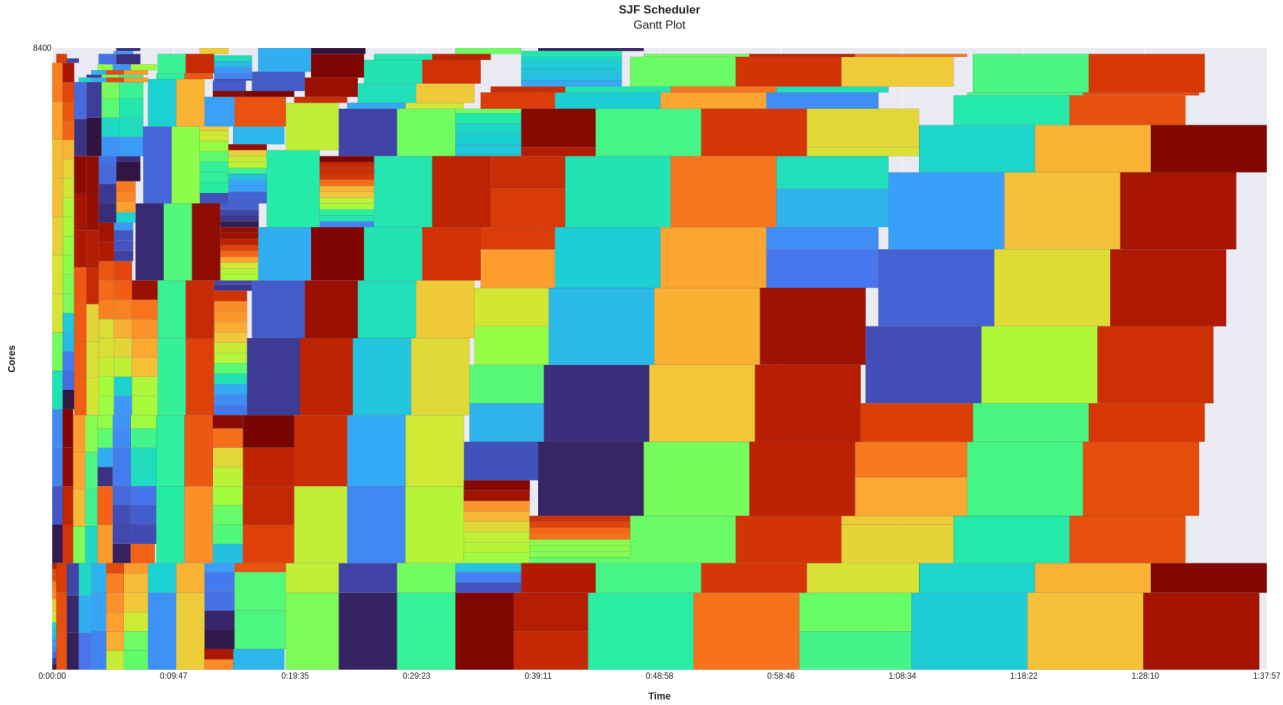


Figure 6.9: Gantt diagram of SJF in workload of 400 jobs on ARIS.

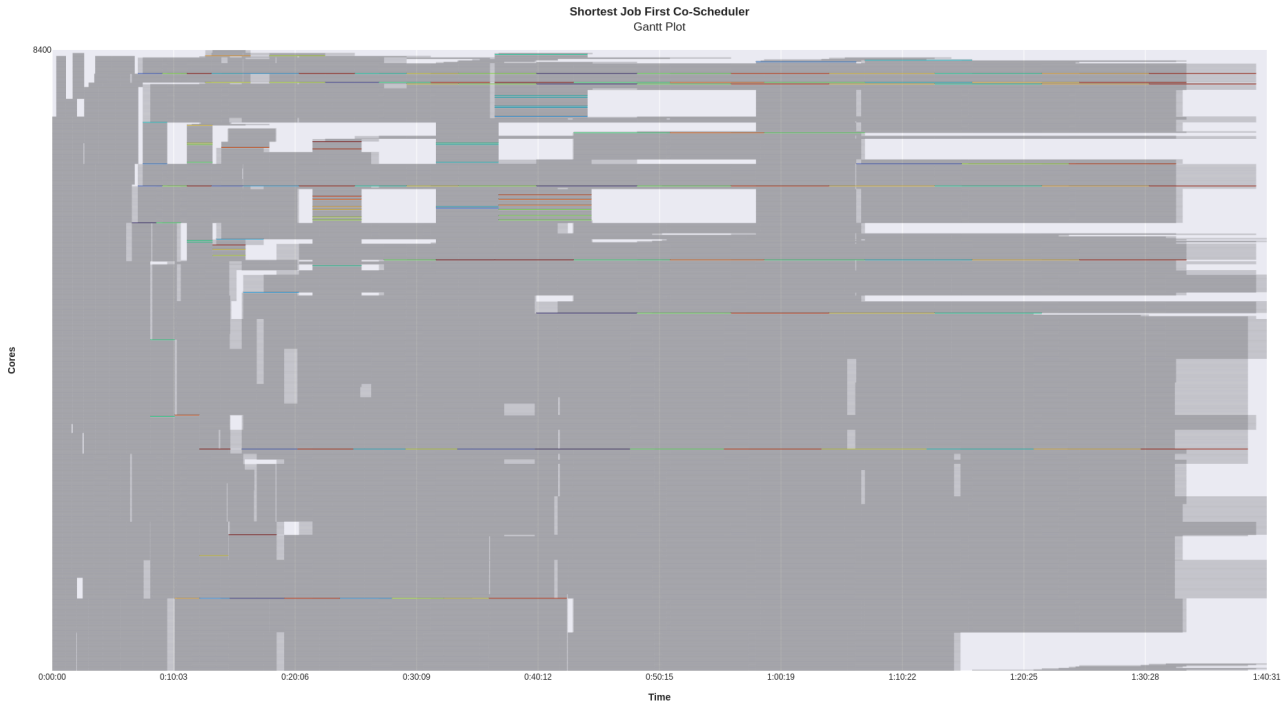


Figure 6.10: Gantt diagram of SJF-Co in workload of 400 jobs on ARIS.

6.3 Pareto Plots

In this section we combine metrics in order to further evaluate our schedulers' performance. We address this as an optimization problem with multiple criteria. As we previously mentioned, the target metrics are makespan speedup and mean slowdown. We add to these criteria the slowdown counts percentage and we seek the optimal-Pareto solution on each combination [33].

Firstly, we study the the mean slowdown values versus makespan speedup (Figure G12). The objectives are maximizing makespan speedup and minimizing mean slowdown. The SJF-Filler approach seems to dominate the solution set, with the approaches that are based on its components (meaning SJF and Filler based), achieving satisfying results as well.

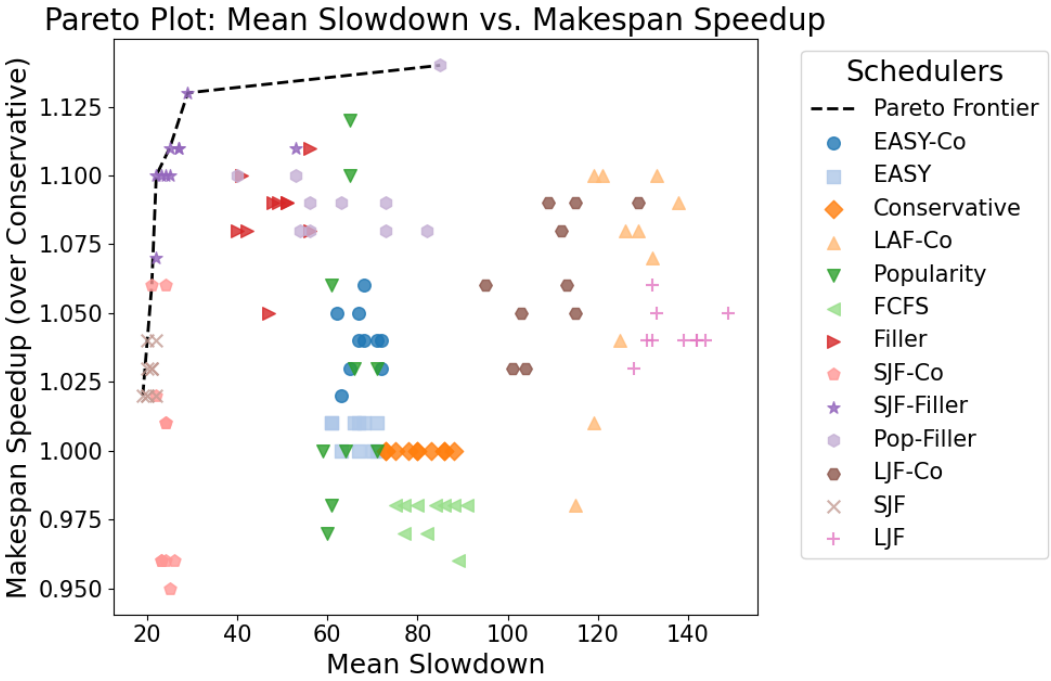


Figure 6.11: Pareto plot with mean slowdown and makespan speedup as objectives.

Next, we examine the slowdown counts percentage and makespan speedup objectives (Figure G12). Here we can see that the two composite Filler approaches have solutions on the pareto set. The original Filler does not perform that well, due to the high slowdown counts percentage values. Popularity co-scheduler also performs well, considering it has two optimal solutions. This can be considered a success as the scheduler was designed to avoid a slowdown in execution times. If we were to ignore all the schedulers that include compact deployment, we see that SJF-Co and SJF-Filler outperform the other co-schedulers.

Finally, we inspect mean slowdown and slowdown counts percentage objectives (Figure 6.13). The frontier consists of only one point, which is one instance of the SJF scheduler. SJF clearly dominates all implementations Again, the rest compact schedulers have a similar performance and from the co-scheduling ones SJF-Co, SJF-Filler and Popularity seem to perform better. Interestingly, in this scatterplot the different instances of the schedulers are more clustered.

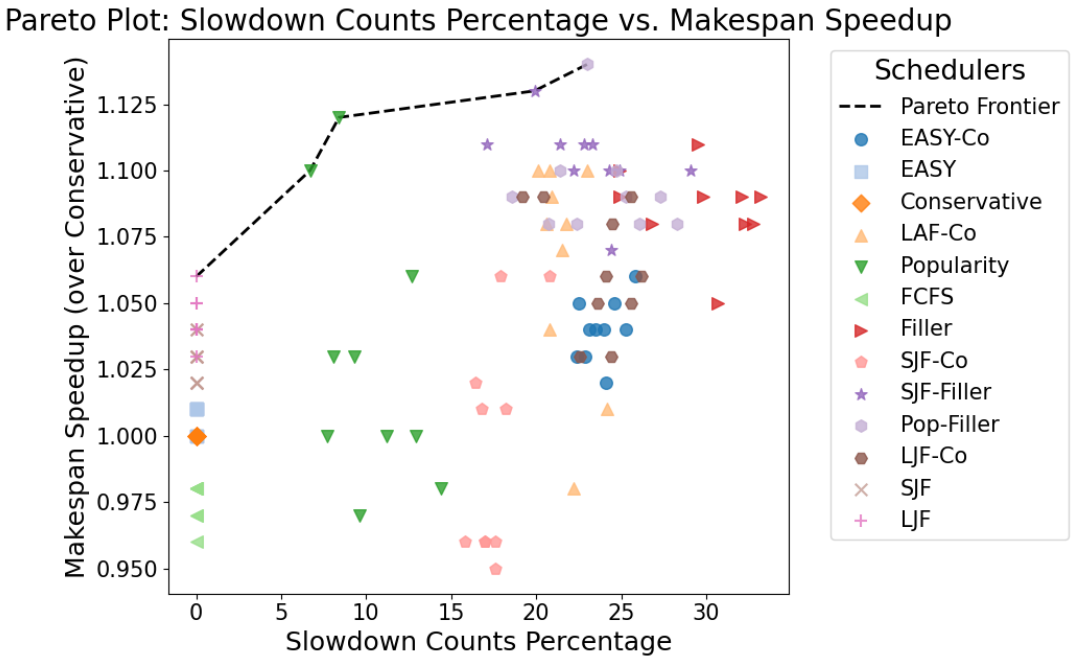


Figure 6.12: Pareto plot with slowdown counts percentage and makespan speedup as objectives.

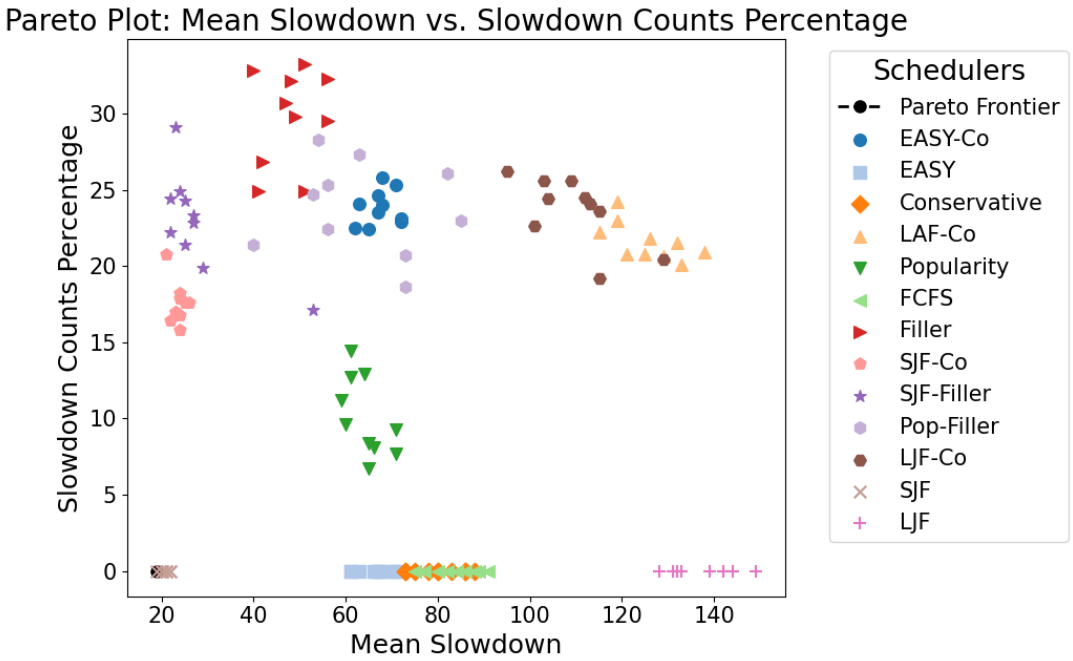


Figure 6.13: Pareto plot with mean slowdown and slowdown counts percentage as objectives.

6.4 Metrics correlation

In this section we study the correlation between the different metrics. In order to achieve this, we created three heatmaps (Figures 6.14 and 6.15) using Pearson, Kendall, and Spearman correlation coefficients [34, 35].

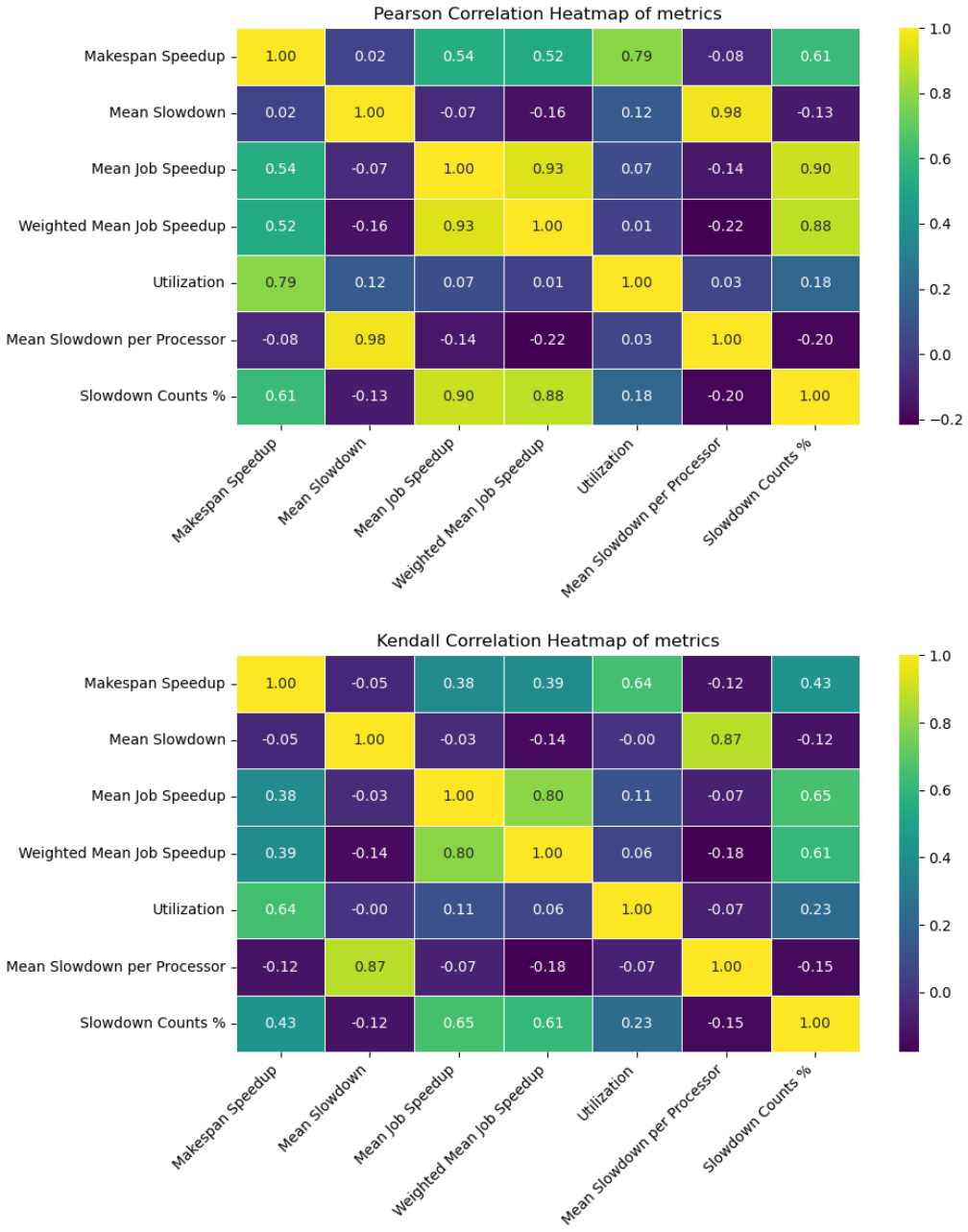


Figure 6.14: Correlation heatmaps of Pearson and Kendal coefficients.

We observe that makespan speedup shows a strong positive correlation with utilization (0.79 Pearson, 0.64 Kendall, 0.79 Spearman), indicating that as makespan speedup increases, utilization tends to increase as well. It seems obvious that mean job speedup and weighted mean job speedup and mean slowdown and mean slowdown per processor are highly correlated with each other (0.93 Pearson, 0.80 Kendall, 0.90 Spearman and 0.98 Pearson, 0.87 Kendall, 0.97 Spearman

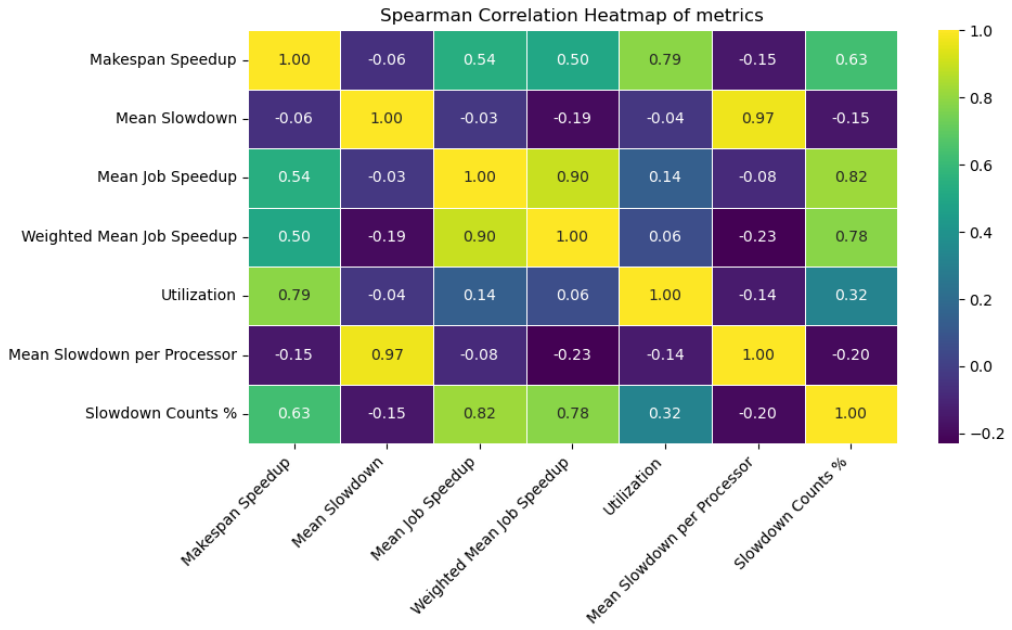


Figure 6.15: Correlation heatmap of Spearman coefficient.

respectively), as they measure similar aspects of performance.

On the other hand, makespan speedup has a very weak or negligible correlation with mean slowdown and mean slowdown per processor (0.02 and -0.06 Pearson, -0.05 and -0.12 Kendall, -0.06 and -0.15 Spearman), suggesting these metrics are largely independent of each other. This seems to verify our decision of using these two metrics as the main two optimization objectives.

Utilization shows weak correlations with mean job speedup and weighted mean job speedup (0.07 and 0.01 Pearson, 0.11 and 0.06 Kendall, 0.14 and 0.08 Spearman), indicating that utilization is not strongly influenced by these speedup metrics.

There are some negative correlations, such as between weighted mean job speedup and mean slowdown per processor (-0.22 Pearson, -0.18 Kendall, -0.23 Spearman), suggesting that as job speedup increases, slowdown per processor tends to decrease.

We also observe a correlation between mean job speedup and slowdown counts percentage (0.90 Pearson, 0.65 Kendall, 0.82 Spearman). This may seem confusing at first, but we estimate that it is caused by the slowdown some jobs suffer from when co-executed.

The Pearson, Kendall, and Spearman correlation coefficients generally show consistent patterns, though the strength of the correlations varies. This indicates that the relationships between the metrics are likely real and not due to outliers. Pearson coefficients tend to be higher, which is expected as Pearson measures linear relationships, while Kendall and Spearman measure rank correlations and are more robust to non-linear relationships.

Chapter 7

Summary and Conclusions

7.1 Summary

This Thesis explored the development and evaluation of co-scheduling algorithms for High-Performance Computing (HPC) systems, aiming to address the growing demand for computational power while optimizing resource utilization and reducing energy consumption. The primary focus was on improving system performance and user satisfaction through co-scheduling, a technique that allows multiple jobs to share computational nodes, thereby reducing resource contention and improving overall efficiency.

The research began by identifying the limitations of traditional scheduling algorithms, such as First Come First Serve (FCFS) and EASY, which often lead to underutilization of resources and increased energy costs. Co-scheduling was proposed as a solution to these challenges, with the goal of balancing system performance (measured by makespan speedup) and user satisfaction (measured by mean job slowdown).

To evaluate the proposed co-scheduling algorithms, the Efficient Lightweight Scheduling Estimator (ELiSE) simulator was used. ELiSE allowed for controlled testing of various scheduling policies, including traditional and co-scheduling algorithms, using workloads generated from the NAS Parallel Benchmarks (NPB).

Several co-scheduling algorithms were introduced and evaluated, including EASY Co-schedule, Largest Area First Co-schedule (LAF-Co), Popularity, Shortest Job First Co-schedule (SJF-Co), Longest Job First Co-schedule (LJF-Co), Filler, and Copy Queue. Among these, the SJF-Filler algorithm, one of the two Copy Queue algorithms, stood out for its ability to reduce fragmentation and improve system utilization by considering the current state of the system during scheduling decisions.

Experimental results demonstrated that co-scheduling algorithms, particularly SJF-Filler and Filler, achieved significant improvements in makespan speedup and mean job speedup, while maintaining low mean slowdown values. These algorithms effectively balanced system performance and user satisfaction, making them promising candidates for real-world HPC systems.

7.2 Conclusions

The findings of this Thesis highlight the potential of co-scheduling to enhance the performance and efficiency of HPC systems. By allowing multiple jobs to share computational nodes, co-scheduling reduces resource contention and improves utilization, leading to faster completion times for workloads and better overall system performance. The Filler algorithm, in particular, demonstrated superior performance by actively reducing fragmentation and optimizing resource allocation.

However, the research also revealed that co-scheduling can lead to increased mean slowdown for individual jobs, as some jobs may experience performance degradation when co-executed with others. This trade-off between system performance and user satisfaction must be carefully managed, especially in environments where fairness and user experience are critical.

The use of the ELiSE simulator provided valuable insights into the behavior of different scheduling algorithms under various workloads. The simulator's ability to generate detailed metrics and visualizations allowed for a thorough evaluation of each algorithm's strengths and weaknesses, guiding the development of more effective scheduling policies.

7.3 Future work

Based on the conclusion reached above and the relative discussion in the application chapters in this Thesis, the following future works are proposed:

- Experiments in real systems:
  In this Thesis all experiments were run on ELiSE, were execution time was estimated as described in Chapter 4. We recognize that in real time execution, results may vary. Thus, we consider that experiments in real systems (perhaps without the extra knowledge of remaining time) should be conducted in order to validate our observations.

- Exploration of different colocation tactics:
  In all co-scheduling algorithms, the resource allocation was made in a FCFS way, meaning that after a job was deployed it occupied the first available node (at first if enough space the jobs were deployed as spread) without futher computations. A different yet effective allocation strategy would be of a great interest.

- Execution aware scheduler implementation:
  The results of this study indicate that the co-scheduling policies which consider the system state before the scheduling decision outperform those who do not. We believe that more execution aware approaches could be studied and perhaps bring better results.

- Fairness and Pricing Models:
  Co-scheduling can lead to performance degradation for some jobs, which may raise fairness concerns, especially in environments where users are charged based on resource usage. Future research could explore pricing models that account for the impact of co-scheduling on individual job performance, ensuring fairness and transparency for users.

- Integration with Machine Learning:
  Machine learning techniques could be employed to predict the performance of co-executed jobs and optimize scheduling decisions. By leveraging historical data and real-time system metrics, machine learning models could further enhance the efficiency and effectiveness of co-scheduling algorithms.

List of Figures

List of Tables

Εκτεταμένη Ελληνική Περίληψη

1 Εισαγωγή

Η διαρκώς αυξανόμενη ζήτηση υπολογιστικής ισχύος που παρατηρείται στις μέρες μας, καθιστά την χρήση υπερυπολογιστών (supercomputers) αναγκαία. Τα υπερυπολογιστικά συστήματα μπορεί να παρέχουν μεγάλη υπολογιστική ισχύ, όμως έχουν σημαντικά μεγάλες ενεργειακές απαιτήσεις οδηγώντας σε μεγάλα κόστη λειτουργίας. Αναφέρεται επίσης στην βιβλιογραφία [19–21], οι πόροι των συστημάτων υψηλών επιδόσεων (HPC) δεν χρησιμοποιούνται πλήρως. Επομενως, κρίνουμε σκόπιμη την προσπάθεια βελτιστοποίησης της λειτουργίας των συστημάτων αυτών.

Η επιλεγμένη κατεύθυνση βελτιστιποίησης είναι η χρήση της τεχνικής διαμοιρασμού πόρων και η αξιοποίηση της στους αλγόριθμους χρονοδρομολόγησης. Σε αντίθεση με τη σύνηθη πρακτική, όπου για την εκέλεση της η κάθε εργασία καταλαμβάνει τους απαραίτητους κόμβους σύμφωνα με τις απαιτήσεις της και οι κόμβοι ”ανήκουν” σε αυτή, στον διαμοιρασμό πόρων περισσότερες από μία εργασίες μπορούν να χρησιμοποιήσουν τους πόρους ιδίων κόμβων. Έχει δειχθεί ότι εφαρμογές με διαφορετική συμπεριφορά εκτέλεσης (εφαρμογές που έχουν υψηλές απαιτήσεις σε μνήμη και εφαρμογές με υψηλές υπολογιστικές απαιτήσεις), βελτιώνουν την επίδοσή τους αν εκτελεστούν μαζί (δηλαδή αν υπάρξει επικάλυψη στους κόμβους που καταλαμβάνουν) και μπορούν να βελτιώσουν την συνολική επίδοση του συστήματος [22]. Βέβαια αξίζει να σημειωθεί ότι η επίδοση του συστήματος μπορεί να βελτιωθεί εις βάρος της επίδοσης των ξεχωριστών εργασιών.

2 Υπόβαθρο χρονοδρομολόγησης σε HPC

Παρέχουμε το υπόβαθρο στα συστήματα υψηλών επιδόσεων, την χρονοδρομολόγηση και τις μετρικές με βάση τις οποίες θα αξιολογήσουμε την επίδοση των αλγορίθμων χρονοδρομολόγησης και συνδρομολόγησης.

2.1 Συστήματα υψηλών επιδόσεων (HPC systems)

Τα συστήματα υψηλών επιδόσεων ή υψηλών επιδόσεων συστήματα παράλληλης επεξεργασίας αξιοποιούν τις δυνατότητες που προσφέρει η παράλληλη επεξεργασία, έτσι ώστε να επιλύσουν προβλήματα, τα οποία απαιτούν μεγάλο όγκο υπολογισμών και πιθανόν σε μεγάλα δεδομένα. Τα συστήματα παράλληλης επεξεργασίας μπορούν να κατηγοριοποιηθούν με βάση τον τρόπο που χρησιμοποιούν το σύστημα μνήμης σε συστήματα κοινόχρηστης μνήμης και συστήματα κατανεμημένης μνήμης: Στα συστήματα κοινής μνήμης, κάθε διεργασία μπορεί να έχει πρόσβαση

στην κύρια μνήμη, με αποτέλεσμα η πρόσβαση στη μνήμη να είναι γρήγορη. Ωστόσο, υπάρχει ο κίνδυνος συνθήκης ανταγωνισμού (race conditions) και απαιτούνται τεχνικές συγχρονισμού. Το πρότυπο OpenMP [6] χρησιμοποιείται κυρίως, έτσι ώστε η παραλληλοποίηση του κώδικα να μην είναι ιδιαίτερα περίπλοκη.

Στα συστήματα κατανεμημένης μνήμης, οι διεργασίες δεν μπορούν να έχουν πρόσβαση σε δεδομένα που ανήκουν σε άλλες διεργασίες και, ως εκ τούτου, η επικοινωνία μεταξύ των διεργασιών επιτυγχάνεται μέσω ανταλλαγής μηνυμάτων (message passing). Η ανταλλαγή μηνυμάτων προσθέτει επιπλέον φόρτο στην εκτέλεση, αλλά επιτρέπει την επεκτασιμότητα (scalability), καθώς δεν υπάρχει ανταγωνισμός για τη μνήμη μεταξύ των διεργασιών. Σε τέτοια συστήματα χρησιμοποιείται το πρότυπο MPI (Message Passing Interface) [7].

2.2 Χρονοδρομολόγηση

Με τον όρο χρονοδρομολόγηση, αναφερόμαστε στην διαδικασία επιλογής εργασιών προς εκτέλεση. Η επιλογή των εργασιών γίνεται από μία δομή που ονομάζεται ουρά αναμονής (waiting queue). Συνήθως επιλέγεται η κεφαλή της ουράς αναμονής και η ίδια η ουρά μπορεί να ταξινομηθεί με βάση μία προτεραιότητα. Η διαχείρηση της ουράς μπορεί να διαφέρει σύμφωνα με τον επιλεγόμενο αλγόριθμο χρονοδρομολόγησης. Στόχοι της χρονοδρομολόγησης είναι η ελαχιστοποίηση του χρόνου αναμονής των εργασιών και η μεγιστοποίηση των εργασιών που ολοκληρώνονται, διασφαλίζοντας συγχρόνως την δικαιοσύνη μεταξύ των εργασιών. Αφού επιλεχθεί η εργασία προς εκτέλεση, ακολουθεί η διαδικασία ανάθεσης πόρων.

Στα HPC συστήματα, σύμφωνα με την κλασσική χρονοδρομολόγηση η ανάθεση πόρων γίνεται σύμφωνα με τους απαιτούμενους υπολογιστικούς πυρήνες (εφόσον δεν έχει προσδιοριστεί κάποιος ειδικός τύπος όπως GPU). Συγκεκριμένα, ανατίθεται ο ελάχιστος αριθμός κόμβων, ο οποίος καλύπτει τον αριθμό πυρήνων που έχει ζητηθεί. Οι πιο διαδεδομένοι αλγόριθμοι χρονοδρομολόγησης είναι ο FCFS, ο EASY και ο Conservative, οι οποίοι θα περιγραφούν αναλυτικά στα επόμενα κεφάλαια.

2.3 Μετρικές αξιολόγησης

Οι κύριοι στόχοι για την βελτιστοποίηση των αλγορίθμων χρονοδρομολόγησης (αποδοτικότητα συστήματος και ικανοποίηση χρήστη) είναι αντικρουόμενοι, επομένως η αξιολόγηση τους είναι ένα περίπλοκο ζήτημα. Χρησιμοποιήσαμε λοιπόν, σχετικές μετρικές [12], δηλαδή η αξιολόγηση και η επιλογή του ”βέλτιστου” δρομολογητή σύμφωνα με πληθώρα μετρικών. Παρακάτω, ορίζονται οι μετρικές που θα χρησιμοποιηθούν για την αξιολόγηση, οι οποίες χωρίζονται σε μετρικές στόχους και επεξηγηματικές.

2.3.1 Makespan Speedup

Παρουσιάζονται οι μετρικές στόχοι της παρούσας μελέτης. Απο πλευράς συστήματος, επιλέγουμε να εξετάσουμε το makespan speedup. Με τον όρο makespan αναφερόμαστε στον χρόνο ολοκλήρωσης ενός συνόλου εργασιών. Μετράμε την επιτάχυνση του χρόνου ολοκλήρωσης του εκάστοτε χρονοδρομολογητή ως προς αυτόν του Conservative. Στόχος μας είναι η ελάττωση

του χρόνου ολοκλήρωσης. Το makespan speedup mS υπολογίζεται ως:

$$mS = \frac{max_i \left(t_i^{fin}\right)_{\text{Conservative}}}{max_j \left(t_j^{fin}\right)} \tag{7.1}$$

2.3.2 Mean Slowdown

Μελετώντας τώρα την πλευρά των χρηστών, εισάγουμε την μετρική της μέσης επιβράδυνσης ή mean slowdown. Για κάθε εργασία, ορίζουμε το slowdown ως τον λόγο του χρόνου παραμονής της εργασίας αυτής στο σύστημα, προς τον χρόνο εκτέλεσής της [14]. Το slowdown παρέχει ένα μέτρο δικαιοσύνης μεταξύ των εφαρμογών, αφού η καθυστέρηση που δεχθεί η εκάστοτε εργασία σταθμίζεται από τον χρόνο εκτέλεσής της. Το slowdown sld_i της εργασίας J_i ορίζεται ως:

$$sld_i = \frac{t_i^{run} + t_i^{wait}}{t_i^{run}} \tag{7.2}$$

Επομένως το μέσο:

$$\overline{sld} = \frac{1}{J} \cdot \sum_{i=1}^{J} sld_i \tag{7.3}$$

2.3.3 Mean Slowdown Per Processor

Συμπληρωματικά με το mean slowdown, έχει νόημα να μελετήσουμε το mean slowdown ανά επεξεργαστή. Σε ευτή την εκδοχή της μετρικής γίνεται κανονικοποίηση σύμφωνα με την απαίτηση της εργασίας σε υπολογιστικούς πυρήνες [17]. Το slowdown ανά πυρήνα $sldpp_i$ της εργασίας J_i ορίζεται ως:

$$sldpp_i = \frac{t_i^{run} + t_i^{wait}}{t_i^{run}} \cdot \frac{1}{n_i^{req}} \tag{7.4}$$

Επομένως το μέσο:

$$\overline{sldpp} = \frac{1}{J} \cdot \sum_{i=1}^{J} sldpp_i \tag{7.5}$$

2.3.4 Utilization

Προχωράμε στις επεξηγηματικές μετρικές, ξεκινώντας από τον βαθμό χρησιμοποίησης του συστήματος. Ο βαθμός χρησιμοποίησης ποσοτικοποιεί την αποδοτικότητα του συστήματος, καθώς εκφράζει το ποσό των πόρων που χρησιμοποιούνται προς τους διαθέσιμους. Υπολογίζουμε τον βαθμό χρησιμοποίησης στο χρονικό διάστημα $[t_1, t_2]$ ως εξής:

$$U(t_1, t_2) = \frac{\int_{t_1}^{t_2} \rho(t)dt}{N(t_2 - t_1)} \tag{7.6}$$

όπου $\rho(t)$ το πλήθος εργασιών υπό εκτέλεση τη χρονική στιγμή t και N το πλήθος πυρήνων του συστήματος.

2.3.5 Mean Job Speedup

Άλλη μία μετρική που προσπαθεί να ποσοτικοποιήσει την ευχαρίστηση του χρήση είναι η μέση επιτάχυνση στον χρόνο εκτέλεσης (mean job speedup). Εξαιτίας του διαμοιρασμού πόρων, οι χρόνοι εκτέλσης δεν είναι οι ίδιοι σε κάθε χρονοδρομολογητή. Η επιτάχυνση μετριέται με βάση τον χρόνο εκτέλεσης της κλασσικής ανάθεσης πόρων. Το speedup S_i μίας εργασίας J_i υπολογίζεται ως

$$S_i = \frac{(t_i^{run})_{\text{compact}}}{t_i^{run}} \tag{7.7}$$

Επομένως το μέσο speedup υπολογίζεται:

$$\bar{S} = \frac{1}{J} \cdot \sum_{i=1}^{J} S_i \tag{7.8}$$

2.3.6 Weighted Mean Job Speedup

Η μέση σταθμισμένη επιτάχυνση (weighted mean job speedup), εξετάζεται ως εναλλακτική της μετρικής του mean job speedup, καθώς αποδίδει μεγαλύτερη βαρύτητα στις εργασίες με μεγάλο μέγεθος, οι οποίες πιθανόν έχουν μεγαλύτερο αντίκτυπο στο σύστημα. Κάθε S_i πολλαπλασιάζεται με έναν παράγοντα w_i, ο οποίος είναι το γινόμενο των t_i^{run} και N_i.

Ορίζουμε το Weighted Mean Job Speedup ως:

$$WMJS = \frac{1}{\sum_i w_i} \sum_i w_i \cdot S_i \tag{7.9}$$

2.3.7 Slowdown Counts

Στην προσπάθειά μας να συγκρίνουμε τις διαφορετικές εμπειρίες χρήστη, μετρήσαμε το ποσοστό των εργασιών στις οποίες οι χρόνοι εκτέλεσης επιδεινώθηκαν. Η μετρική αυτή δεν πρέπει να συγχέεται με το mean slowdown.

3 Συνδρομολόγηση (Co-scheduling)

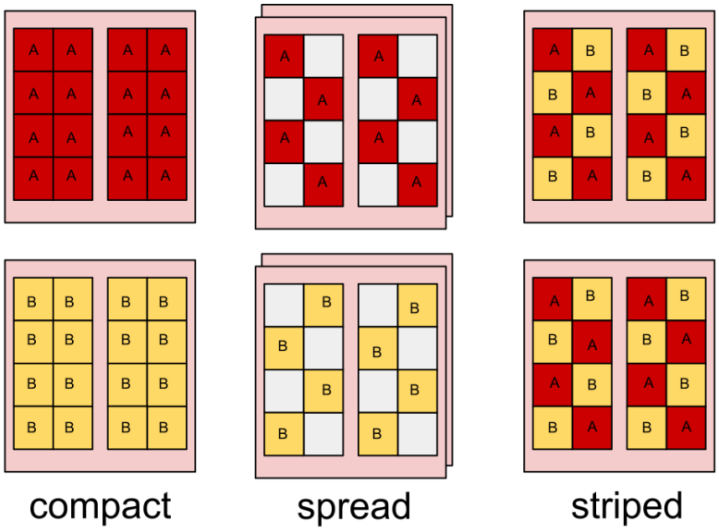
Σε αυτήν την ενότητα, περιγράφουμε την τεχνική του διαμοιρασμού πόρων (node sharing), ο οποίος αξιοποιείται στην συνδρομολόγηση (co-scheduling). Όπως αναφέρθηκε προηγουμένως, τα περισσότερα συστήματα HPC είναι υποεκμεταλλευόμενα λόγω κατακερματισμού και ανταγωνισμού για τους πόρους. Η μέθοδος που επιλέχθηκε για την αντιμετώπιση αυτού του προβλήματος είναι η κοινή χρήση κόμβων.

3.1 Διαμοιρασμός κόμβων

Στην κοινή χρήση κόμβων, οι πόροι κάθε κόμβου δεν χρησιμοποιούνται αποκλειστικά από μία εφαρμογή, αλλά από ένα σύνολο εφαρμογών. Ο διαχειριστής πόρων μπορεί να εκχωρήσει διαφορετικό αριθμό επεξεργαστών σε διαφορετικές εφαρμογές, με αποτέλεσμα αυτές να καταλαμβάνουν περισσότερους κόμβους από τον αναμενόμενο (ο οποίος υπολογίζεται ως ζητούμενοι

πυρήνες προς τους πυρήνες ανά κόμβο). Οι βασικές στρατηγικές κατανομής των εφαρμογών είναι τρεις (όπως φαίνεται στο Σχήμα G1):

- Συμπαγής κατανομή (Compact)
- Διάσπαρτη κατανομή (Spread)
- Διαμοιρασμένη κατανομή (Striped)



Σχήμα G1: Οπτικοποίηση στρατηγικών κατανομής σε κόμβους [1].

Ο διαχειριστής πόρων μπορεί να αναπτύξει μια εφαρμογή με συμπαγή κατανομή, να την κατανεμηθσει σε περισσότερους κόμβους ή να την εκτελέσει σε κόμβους όπου ήδη εκτελείται κάποια άλλη εφαρμογή. Θα συζητήσουμε περαιτέρω πώς αυτή η συνεκτέλεση μπορεί να είναι ευεργετική τόσο για το σύστημα όσο και για τις μεμονωμένες εφαρμογές.

3.2 Πλεονεκτήματα

Όταν οι εφαρμογές κατανέμονται σε περισσότερους κόμβους, εκμεταλλεύονται περισσότερους πόρους, όπως επιπλέον επίπεδα κρυφής μνήμης (last-level cache) ή επιταχυντές, όπως GPUs. Επιπλέον, η μεταφορά δεδομένων μέσα σε κάθε κόμβο βελτιστοποιείται, καθώς εκτελούνται λιγότερες διεργασίες ανά κόμβο και συνεπώς μεταφέρονται λιγότερα δεδομένα (λαμβάνοντας υπόψιν εφαρμογές που χρησιμοποιούν MPI).

Η κοινή χρήση κόμβων έχει θετικό αντίκτυπο σε εφαρμογές που εμφανίζουν αντίθετες απαιτήσεις πόρων. Για παράδειγμα, μια εφαρμογή με υψηλή κατανάλωση μνήμης (δηλαδή μια εφαρμογή της οποίας η απόδοση εξαρτάται από το εύρος ζώνης της μνήμης κατά την κλιμάκωση) ωφελείται όταν συνεκτελείται με μια υπολογιστικά εντατική εφαρμογή, καθώς μειώνονται οι ανάγκες επικοινωνίας. Αυτό μπορεί να οδηγήσει σε μείωση του χρόνου εκτέλεσης της εφαρμογής.

Επιπλέον, από τη σκοπιά του συστήματος, όταν οι εφαρμογές κατανέμονται αρχικά και έπειτα διαμοιράζονται, μειώνεται ο κατακερματισμός των πόρων, και κόμβοι που προηγουμένως δεν αξιοποιούνταν πλήρως μπορούν πλέον να χρησιμοποιούνται αποδοτικότερα.

3.3 Μειονεκτήματα

Από την άλλη πλευρά, διαφορετικές εφαρμογές έχουν διαφορετικά μοντέλα επικοινωνίας, κάτι που μπορεί να οδηγήσει σε επιβράδυνση του χρόνου εκτέλεσης. Αυτό μπορεί να συμβεί λόγω συγχρονισμού των διεργασιών μεταξύ διαφορετικών κόμβων ή απλά επειδή οι εφαρμογές που συνεκτελούνται είναι και οι δύο μνημοβόρες.

Έχει παρατηρηθεί ότι όταν δύο εργασίες που εκτελούν την ίδια εφαρμογή αλλά ανήκουν σε ξεχωριστές διεργασίες (δηλαδή η μία δεν μπορεί να έχει πρόσβαση στα δεδομένα της άλλης), δεν θα πρέπει να συνεκτελούνται. Συγκεκριμένα, εάν είναι μνημοβόρες, θα υπάρξει επιβράδυνση λόγω ανταγωνισμού για τους πόρους της μνήμης, ενώ αν είναι υπολογιστικά εντατικές, θα ήταν πιο επωφελές να συνεκτελούνται με μια εφαρμογή που απαιτεί περισσότερη μνήμη [22, 23].

3.4 Πολυπλοκότητα της συνεκτέλεσης

Όπως αναφέρθηκε παραπάνω, η κοινή χρήση κόμβων μπορεί να έχει είτε θετική είτε αρνητική επίδραση στην εκτέλεση των εφαρμογών. Ομοιογενές ή ετερογενές φορτίο επηρεάζει σημαντικά την απόδοση της συνεκτέλεσης.

Ιδιαίτερα, οι γειτονικές εργασίες παίζουν κρίσιμο ρόλο στον χρόνο εκτέλεσης μιας εργασίας, καθώς μπορούν να αποτελέσουν την κύρια αιτία υποβάθμισης της απόδοσης [25]. Λαμβάνοντας αυτό υπόψη, η συνεκτέλεση γίνεται ένα πολύπλοκο πρόβλημα, καθώς για την αντιμετώπιση αυτών των προκλήσεων απαιτείται περισσότερη γνώση σχετικά με τις εφαρμογές που εκτελούνται στο σύστημα.

Θα μπορούσε να υποστηριχθεί ότι η πιθανή αύξηση στον χρόνο εκτέλεσης μπορεί να γίνει ανεκτή, εφόσον συνοδεύεται από μια επιθυμητή μείωση του χρόνου αναμονής [26]. Στην παρούσα εργασία, θα μελετήσουμε την επίδραση της συνεκτέλεσης και θα προσπαθήσουμε να αναπτύξουμε αλγορίθμους συνεκτέλεσης που δεν απαιτούν πρόσθετες πληροφορίες σχετικά με τις εφαρμογές και μπορούν να λειτουργούν σε κάθε σύστημα.

4 Προσομοιωτής HPC

Σε αυτό το σημείο παρουσιάζουμε το εργαλείο που χρησιμοποιήθηκε στο πειραματικό μέρος, για την αξιολόγηση των δρομολογητών.

4.1 Efficient Lightweight Scheduling Estimator (ELiSE)

Για την προτυποποίηση των αλγορίθμων, έγινε χρήση του Efficient Lightweight Scheduling Estimator (ELiSE), ενός εργαλείου γραμμένου σε python.

Ως είσοδο απαιτούνται:

1. μια απλή περιγραφή του HPC cluster
2. το φορτίο με τις εργασίες
3. ένα heatmap των ανά ζεύγη επιταχύνσεων μεταξύ των εργασιών του φορτίου

Η εκτίμηση των χρόνων εκτέλεσης στην προσομοίωση γίνεται με βάση το χειρότερο speedup. Συγκεκριμένα, από όλες τις γειτονικές συνεκτελούμενες εργασίες (J_n) επιλέγεται το μικρότερο speedup.

$$S_i^{new} = \min_{\forall j \in J_n} getSpeedupWith(j) \tag{7.10}$$

Επομένως, ο νέος υπολειπόμενος χρόνος υπολογίζεται ως:

$$\left(t_i^{run}\right)_{new} = \frac{S_i^{old} \cdot \left(t_i^{run}\right)_{old}}{S_i^{new}} \tag{7.11}$$

4.2 NAS Parallel Benchmarks (NPB)

Οι εργασίες που χρησιμοποιήθηκαν προέρχονται από το σύνολο των NAS Parallel Benchmarks (NPB). Το σύνολο NPB αναπτήχθηκε από τον τομέα Advanced Supercomputing της NASA (NAS) ώατε να προσφέρεται ένας τυποποιημένος τρόπος να μετράται η επίδοση συστημάτν παράλληλης επεξεργασίας [27]. Οι εργασίες που χρησιμοποιούμε είναι οι εξής:

- 5 υπολογιστικοί πυρήνες
  - IS - Integer Sort (ταξινόμηση ακεραίων), τυχαίες προσβάσεις στη μνήμη
  - EP - Embarrassingly Parallel
  - CG - Conjugate Gradient (μέθοδος συζυγών κλισεων), μη κανονική επικοινωνία και προσβάσεις στη μνήμη
  - MG - Multi-Grid σε ακολουθία πλεγμάτων, σύντομης και μεγάλης απόστασης επικοινωνία, memory intensive
  - FT - διακριτός 3D fast Fourier Transform (γρήγορος ματασχηματισμός Fourier), επικοινωνία μεταξύ όλων
- 3 ψευδοεφαρμογές
  - BT - Block Tri-diagonal επιλύτης
  - SP - Scalar Penta-diagonal επιλύτης
  - LU - Lower-Upper Gauss-Seidel επιλύτης

Οι εργασίες είναι χωρισμένες σε δύο κλάσεις D και E, οι οποίες υποδηλώνουν το μέγεθος του προβλήματος (Πίνακας G1).

5 Αλγόριθμοι (Συν)δρομολόγησης

Σε αυτή την ενότητα παρουσιάζονται οι αλγόριθμοι χρονοδρομολόγησης που υλοποιήθηκαν. Περιλαμβάνονται κλασσικές προσεγγίσεις αλλά και προσεγγίσεις που υλοποιούν συνδρομολόγηση.

5.1 Αλγόριθμοι κλασσικής χρονοδρομολόγησης

Αρχικά παρουσιάζουμε τους αλγόριθμους χρονοδρομολόγησης που χρησιμοποιούνται ευρέως.

Πίνακας G1: Μέγεθος προβλήματος κάθε κλάσης.

Benchmark	Parameter	Class D	Class E
CG	no. of rows	1500000	9000000
	no. of nonzeros	21	26
	no. of iterations	100	100
	eigenvalue shift	500	1500
EP	no. of random-number pairs	236	240
FT	grid size	2048 ×1024 ×1024	4096 ×2048 ×2048
	no. of iterations	25	25
IS	no. of keys	231	235
	key max. value	227	231
MG	grid size	1024 ×1024 ×1024	2048 ×2048 ×2048
	no. of iterations	50	50
BT	grid size	408 ×408 ×408	1020 x 1020 x 1020
	no. of iterations	250	250
	time step	0.00002	0.000004
LU	grid size	408 ×408 ×408	1020 ×1020 ×1020
	no. of iterations	300	300
	time step	1.0	0.5
SP	grid size	408 ×408 ×408	1020 ×1020 ×1020
	no. of iterations	500	500
	time step	0.0003	0.0001

5.1.1 First Come First Serve (FCFS)

Η παρούσα προσέγγιση είναι η απλούστερη και δεν απαιτεί υπολογισμούς. Όπως αναφέρεται, στην First Come First Serve (FCFS) προσέγγιση, οι εργασίες εκτελούνται σύμφωνα με την σειρά άφιξης τους, αφού η διάταξη τους στην ουρά αναμονής είναι ταυτόσημη με την σειρά αφίξεων. Στην FCFS δρομολόγηση δεν εμφανίζονται φαινόμενα λιμοκτονίας, καθώς καμία εργασία δεν παραμελείται, μπορεί όμως να εμφανιστεί το φαινόμενο του καραβανιού (convoy effect), όπου μία εργασία η οποία έχει μεγάλες απαιτήσεις σε υπολογιστικούς πυρήνες και έχει μεγάλο χρόνο εκτέλεσης καθυστερεί την εκτέλεση πολλών μικρότερων εργασιών. Επιπλέον, το σύστημα είναι επιρρεπές σε φαινόμενα κατακερματισμού (fragmentation).

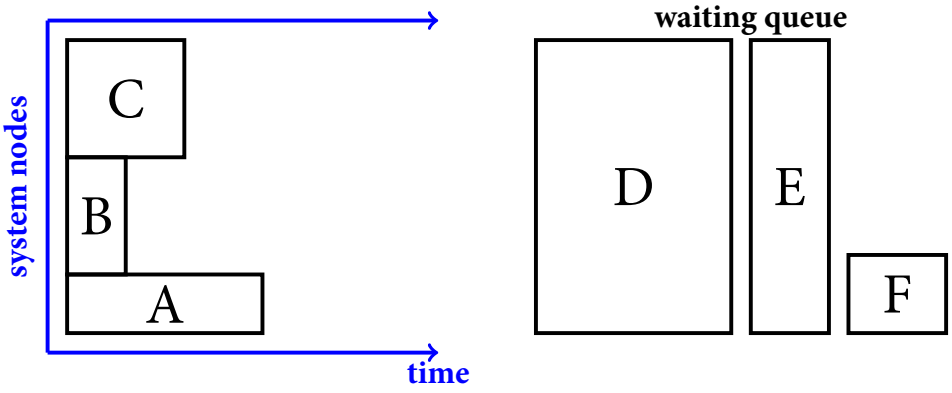
5.1.2 Extensible Argonne Scheduling sYstem (EASY)

Στη συνέχεια παρουσιάζουμε τον αλγόριθμο Extensible Argonne Scheduling sYstem (EASY). Ο EASY αλγόριθμος [28] βασίζεται στην FCFS λογική, με την προσθήκη του backfill για την αντιμετώπιση του φαινομένου του καραβανιού και την βελτίωση του βαθμού χρησιμοποίησης του συστήματος.

Το backfill είναι η διαδικασία κατά την οποία μία εργασία εκτελείται εκτός σειράς αρκεί να μην ”συγκρούεται” με την εκτέλεση των εργασιών που προηγούνται. Υπάρχουν δύο βασικές

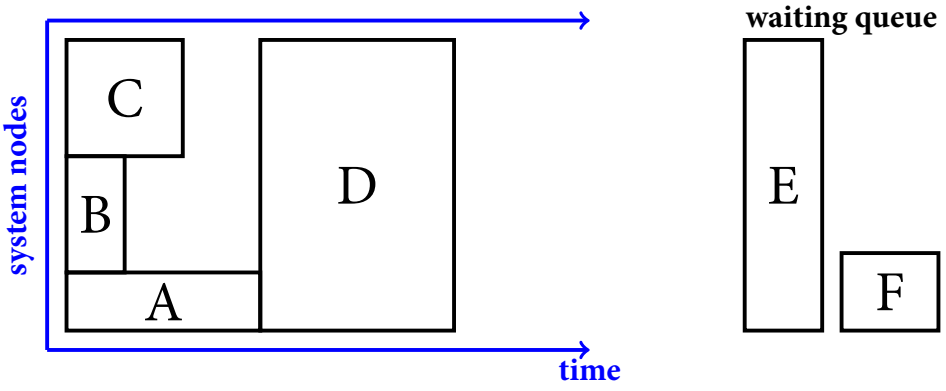
υλοποιήσεις backfilling στην βιβλιογραφία [29] (με τις παραλλαγές τους), το aggressive και conservative. Στο conservative backfilling οι εργασίες λαμβάνουν μια χρονική κράτηση (προσεγγιστικά) για το πότε θα ξεκινήσουν την εκτέλεση με βάση την FCFS λογική. Στην aggressive υλοποίηση, μόνο η πρώτη εργασία προστατεύεται με κράτηση και είναι αυτή που χρησιμοποιείται στον EASY.

Παρουσιάζεται ένα μικρό παράδειγμα του backfill. Εδώ η ουρά αναμονής περιέχει τρεις εργασίες D, E και F (Σχήμα G2).



Σχήμα G2: Κατάσταση συστήματος πριν την δρομολόγηση της D.

Η εργασία D επιλέγεται ως η πρώτη στην ουρά και για την εκτέλεση της πρέπει η εργασία A να ολοκληρώσει την εκτέλεση της, επομένως εκτελείται μόλις οι κατάλληλοι πόροι γίνουν διαθέσιμοι (Σχήμα G3). Έπειτα γίνεται έλεγχος στην ουρά αναμονής για το αν μπορεί να επιλεχθεί

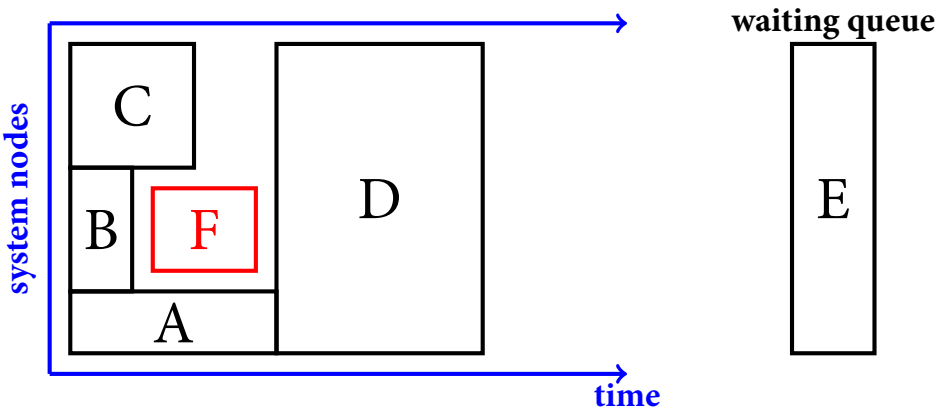


Σχήμα G3: Κατάσταση συστήματος μετά την είσοδο της D.

κάποια εργασία. Επιλέγεται η εργασία E. Η εργασία E δεν μπορεί να εκτελεστεί πριν από την D χωρίς να επηρεάσει το πότε εκτελείται η D. Στη συνέχεια επιλέγεται η εργασία F. Αν εκτελεστεί η εργασία F δεν επηρεάζει την εκτέλεση της D και για αυτό επιλέγεται και εκτελείται (Σχήμα G4). Η διαδικασία αυτλη μπορεί να συνεχιστεί και για τις υπόλοιπες εργασίες της ουράς αν υπάρχουν, ή να τεθεί ένα όριο (βάθος backfill).

5.1.3 Conservative

Σε αυτόν τον αλγόριθμο υλοποιείται το conservative backfilling, όπου όπως αναφέρθηκε προστατεύονται όλες οι εργασίες που προηγούνται από την υποψήφια προς backfill εργασία. Αυτή



Σχήμα G4: Είσοδος εργασίας F εκτός σειράς (backfill)

η προσέγγιση θεωρείται πιο δίκαιη από την EASY.

5.1.4 Shorterst Job First (SJF)

Ο Shortest Job First (SJF) είναι ένας ακόμη κοινός αλγόριθμος, όπου η ουρά αναμονής ταξινομείται με βάση τον απαιτούμενο χρόνο εκτέλεσης. Συγκεκριμένα, εκτελούνται πρώτα οι εργασίες με μικρό χρόνο εκτέλεσης (στα πραγματικά συστήματα χρησιμοποείται το wall time). Αναμένουμε μικρές τιμές στο mean slowdown, καθώς πλέον οι εργασίες με μικρούς χρόνους εκτέλεσης έχουν και τους μικρότερους χρόνους αναμονής. Τέλος, αυτή η υλοποίηση μπορεί να οδηγήσει σε φαινόμενα λιμοκτονίας.

5.1.5 Longest Job First (LJF)

Ο Longest Job First (LJF) αλγόριθμος ευνοεί την εκτέλεση των μεγάλων σε χρόνο εκτέλεσης εργασιών. Σε αυτή την υλοποίηση αναμένουμε σημαντικά υψηλότερες τιμές του mean slowdown, καθώς η εμφανής μεροληψία προς τις σύντομες εργασίες αυξάνει σημαντικά τους χρόνους αναμονής. Από την άλλη περιμένουμε υψηλές τιμές utilization.

5.2 Αλγόριθμοι συνδρομολόγησης

Σε αυτή την υποενότητα παρουσιάζουμε τους αλγορίθμους συνδρομολόγησης που υλοποιήθηκαν. Αποτελούνται από τις εκδοχές των κλασσικών υλοποιήσεων με την προσθήκη της τεχνικής διαμοιρασμού πόρων, και κάποιες δικές μας υλοποιήσεις.

5.2.1 EASY Co-schedule (EASY-Co)

Η Co-schedule εκδοχή του EASY αλγορίθμου διαφοροποιείται από την κλασσική εκδοχή στον τρόπο με τον οποίο γίνεται η ανάθεση κόμβων. Συγκεκριμένα, οι εργασίες εκτελούνται πρώτα ως spread και μόλις το σύστημα γεμίσει, εκτελούνται ως striped. Αναμένουμε την ίδια συμπεριφορά με τον EASY, με πιθανόν μια βελτίωση στον βαθμό χρησιμοποίησης.

5.2.2 Shorterst Job First Co-schedule (SJF-Co)

Σε αυτή την προσέγγιση, προσπαθούμε να βελτιστοποιήσουμε την επίδοση, λαμβάνοντας υπόψην το slowdown. Γίνεται ταξινόμηση της ουράς αναμονής σύμφωνα με τους χρόνους εκτέλεσης, ευνοώντας τις σύντομες εργασίες. Αναμένουμε αντίστοιχη συμπεριφορά με τον SJF και προσθέτουμε aggresive backfilling και προφανώς διαμοιρασμό πόρων.

5.2.3 Longest Job First Co-schedule (LJF-Co)

Ομοίως με τον SJF-Co, στον Longest Job First Co-shedule (LJF-Co) μελετούμε την επίδραση του διαμοιρασμού πόρων στον LJF. Αναμένουμε ακόμη μικρότερο κατακερματισμό στο σύστημα και κατά συνέπεια μεγαλύτερο makespan speedup. Όπως και στον LJF, αναμένουμε υψηλές τιμές στο mean slowdown.

5.2.4 Largest Area First Co-schedule (LAF-Co)

Σε αυτή την υλοποίηση η ουρά αναμονής ταξινομείται έτσι ώστε οι εργασίες με το μεγαλύτερο γινόμενο χρόνου εκτέλεσης και αιτούμενων πυρήνων να τοποθετούνται πρώτες. Εκτελείται επίσης και aggresive backfilling. Προφανώς σε αυτή την υλοποίηση αναμένουμε υψηλές τιμές στο mean slowdown, καθώς υπάρχει ξεκάθαρη μεροληψία κατά των μικρών εργασιών.

5.2.5 Popularity

Ακολουθεί μία υλοποίηση που αναπτύχθηκε για την διπλωματική. Αυτή η προσέγγιση διαφέρει από τις προηγούμενες καθώς προσπαθεί να αξιοποιήσει την επιπλέον παρεχόμενη πληροφορία από το heatmap. Η ουρά αναμονής ταξινομείται με βάση ένα μέτρο δημοφιλίας το οποίο ονομάζουμε ”rank”. Το rank είναι ένας μετρητής ο οποίος εκφράζει το πόσα πετυχημένα ζεύγη συνεκτέλεσης μπορεί να σχηματίσει η κάθε εργασία, με βάση την τρέχουσα κατάσταση της ουράς αναμονής. Συγκεκριμένα, εκτελούντι πρώτα οι εργασίες με υψηλά ranks. Σημειώνουμε ότι ως πετυχημένα ζεύγη θεωρούμε αυτά, στα οποία το μέσο speedup είναι μεγαλύτερο από 1, επομένως μπορεί μία εργασία με υψηλό rank να έχει μεγαλύτερο χρόνο εκτέλεσης, λόγω του διαμοιρασμού πόρων. Επιπλέον, αναφέρουμε ότι καθώς οι λιγότερο δημοφιλείς εργασίες μένουν προς το τέλος της εκτέλεσης, μετά από ένα σημείο το αποτέλεσμα της συνεκτέλεσης είναι αρνητικό. Προσπαθώντας να το αντιμετωπίσουμε αυτό επιλέγουμε να μην κάνουμε διαμοιρασμό πόρων μόλις οι τιμές του rank πέσουν κάτω από 3.

5.2.6 Filler

Αυτός ο αλγόριθμος αποσκοπεί στην αντιμετώπιση του κατακερματισμού, καθώς σε κάθε βήμα η ταξινόμηση της ουράς προωθεί τις εργασίες που θα αφήσουν το μικρότερο κενό στο σύστημα (με βάση την τρέχουσα κατάσταση του). Γίνεται επίσης μία κανονικοποίηση με το αναγνωριστικό της κάθε εργασίας ώστε να τηρηθεί η σειρά αφίξεων και κατά συνέπεια να αποφευχθούν φαινόμενα λιμοκτονίας. Αναμένουμε υψηλό utilization και makespan speedup.

5.2.7 SJF-Filler (Two Factors)

Ακολουθεί μία δική μας υλοποίηση η οποία βασίζεται στον Filler. Σε αυτή την υλοποίηση γίνονται δύο ταξινομήσεις, με την χρήση ενός αντιγράφου της ουράς αναμονής. Συγκεκριμένα, εδω αυτό το κριτήριο είναι ο χρόνος εκτέλεσης, όπου οι εργασίες με τους μικρότερους χρόνους λαμβάνουν την μεγαλύτερη προτεραιότητα. Αφού οι εργασίες ταξινομηθούν επιστρέφεται η θέση τους στην ταξινόμηση προς το πλήθος των εργασιών. Αυτός ο λόγος αποτελεί τον δεύτερο παράγοντα. Ο πρώτος παράγοντας, είναι το τι κενό θα αφήσει η εργασία αν τοποθετηθεί στο σύστημα, όμοια με τον Filler. Η τελική ταξινόμηση προκύπτει από το άθροισμα των δύο παραγόντων. Προσπαθούμε με αυτόν τον συνδυασμό να αυξήσουμε το utilization και makespan speedup, και συγχρόνως να μειώσουμε το mean slowdown.

5.2.8 Pop-Filler (Two Factors)

Ακόμη μία δική μας υλοποίηση, μία διαφορετική υλοποίηση των δύο παραγόντων. Σε αυτόν τον αλγόριθμο συνδρομολόγησης, το κριτήριο ταξινόμησης είναι το rank όπως χρησιμοποιείται στον Popularity. Αναμένουμε αντίστοιχη συμπεριφορά με αυτή του SJF-Filler.

6 Αξιολόγηση

Παρουσιάζουμε τα πειραματικά αποτελέσματα. Εκτελέσαμε 10 πειράματα των 1000 εργασιών στο ELiSE, προσομοιώνοντας το σύστημα ARIS (420 κόμβοι των 20 πυρήνων έκαστος). Επιλέγουμε 1000 εργασίες ανά πείραμα, καθώς συνυπάρχουν πολλές εκδοχές των εργασιών και το σύστημα είναι γεμάτο.

6.1 Μέσες τιμές

Αρχικά παρουσιάζουμε τις μέσες τιμές των μετρικών στα 10 αυτά πειράματα (Πίνακες G2, G3).

scheduler	mean slowdown	mean slowdown pp	slowdown (%)	utilization
SJF	**20.6**	**0.101**	0.00	0.947
SJF-Co	23.6	0.118	17.51	0.875
SJF-Filler	27.7	0.109	22.94	0.964
Filler	48.1	0.189	29.70	0.965
Pop-Filler	63.5	0.265	23.78	**0.971**
Popularity	64.3	0.308	**10.10**	0.932
EASY	66.5	0.372	0.00	0.926
EASY-Co	67.5	0.368	23.82	0.904
Conservative	80.2	0.463	0.00	0.922
FCFS	82.9	0.480	0.00	0.899
LJF-Co	109.6	0.599	23.62	0.939
LAF-Co	125.7	0.745	21.59	0.937
LJF	137.2	0.812	0.00	0.960

Πίνακας G2: Μέσες τιμές μετρικών 10 πειραμάτων (1000 εργασίες στο ARIS) - μέρος 1.

scheduler	mean job speedup	weighted mean job speedup	makespan speedup
SJF	1.000	1.000	1.028
SJF-Co	1.062	1.049	0.995
SJF-Filler	1.074	1.055	**1.104**
Filler	1.065	1.040	1.086
Pop-Filler	1.077	1.038	1.093
Popularity	1.047	1.020	1.029
EASY	1.000	1.000	1.006
EASY-Co	**1.097**	**1.060**	1.040
Conservative	1.000	1.000	1.000
FCFS	1.000	1.000	0.976
LJF-Co	1.083	1.042	1.063
LAF-Co	1.072	1.049	1.065
LJF	1.000	1.000	1.043

Πίνακας G3: Μέσες τιμές μετρικών 10 πειραμάτων (1000 εργασίες στο ARIS) - μέρος 2.

Όπως ήταν αναμενόμενο, ο SJF έχει τις μικρότερες τιμές mean slowdown, ακολουθήμενο από τον SJF-Co. Παρατηρούμε ότι η συνδρομολόγηση επιδεινώνει του χρόνους απόκρισης των εργασιών. Το mean job speedup αυξάνεται παρόλο που περίπου το 18% των εργασιών έχει επιβράδυνση. Από την άλλη, το utilization και makespan speedup της συνεκτελούμενης εκδοχής είναι χαμηλότερο.

Ο SJF-Filler έχει αντίστοιχη μέση τιμή στο mean slowdown με τις υπόλοιπες SJF προσεγγίσεις, επιτυγχάνοντας συγχρόνως υψηλές τιμές utilization και το μέγιστο makespan speedup. Οι άλλες δύο Filler υλοποιήσεις συμπεριφέρονται αντίστοιχα, με τον Pop-Filler συνδυασμό να επιτυγχάνει το μέγιστο utilization.

Ο Popularity επιτυγχάνει το μικρότερο ποσοστό επιβραδυνόμενων εργασιών (εξαιρουμένων των κλασσικών δρομολογητών), ως υβριδική υλοποίηση.

Στον EASY, παρατηρούμε ότι η συνεκτέλεση έχει θετική επίδραση στο mean job speedup, επιτυγχάνοντας επίσης το υψηλότερο mean job speedup. Παρατηρούμε επίσης μία μικρή μείωση στο utilization.

Ακόμη συγκρίνοντας τους FCFS, Conservative και EASY, παρατηρούμε ότι το utilization βελτιώνεται, άμεση επίδραση του backfill.

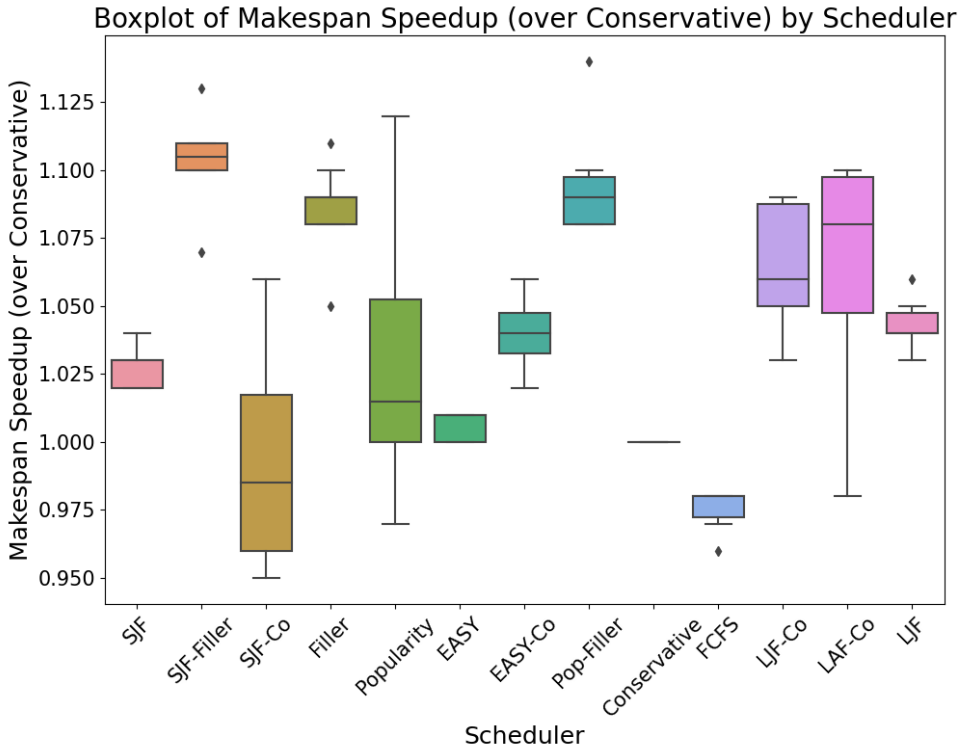
Τέλος, στους LJF-Co, LAF-Co και LJF παρατηρούνται οι μεγαλύτερες τιμές mean slowdown. Αυτές οι υλοποιήσεις βρίσκονται στις υψηλότερες επιδόσεις αναφορικά με το utilization και makespan speedup.

6.2 Boxplots

Παρουσιάζουμε την επίδοση των δρομολογητών σε όλα τα πειράματα, δημιουργώντας τα παακάτω boxplots.

6.2.1 Makespan Speedup

Αρχικά εξετάζουμε το makespan speedup ως μία από τις μετρικές στόχους (Σχήμα G5). Οι υψηλότερες τιμές παρατηρούνται στον SJF-Filler, με τις άλλες δύο προσεγγίσεις να ακολουθούν. Επιπλέον, οι LJF-Co και LAF-Co παρουσιάζουν υψηλές τιμές makespan speedup. Οι δρομολογητές με μέσο makespan speedup μικρότερο του 1 είναι SJF-Co και FCFS. Πιθανόν στον SJF ο διαμοιρασμός πόρων να επιδεινώνει τον κατακερματισμό. Σημειώνουμε όμως ότι οι περισσότερες εκδοχές ξεπερνούν το 1, διατηρώντας τον βασικό μας στόχο, ο οποίος είναι η αποδοτικότερη χρήση του συστήματος.



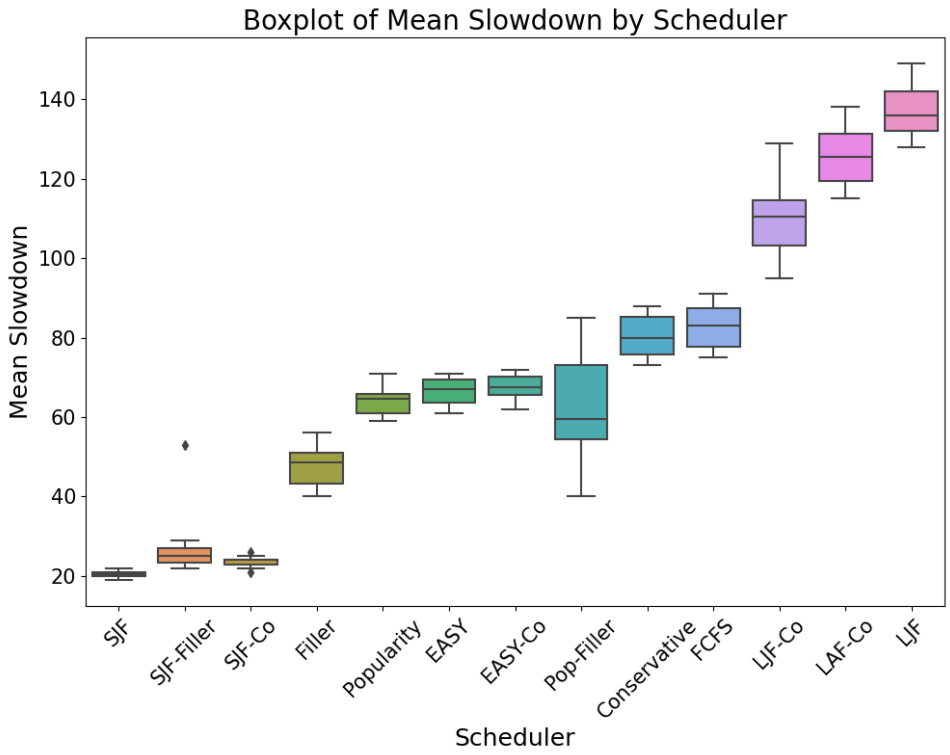
Σχήμα G5: Boxplot of Makespan Speedups.

6.2.2 Mean Slowdown

Αναφορικά με το mean slowdown, οι τρεις SJF υλοποιήσεις έχουν σημαντικά χαμηλότερες τιμές (Σχήμα G6). Παρατηρούμε έναν διαχωρισμό σε SJF και LJF περιοχές με τις δεύτερες να εμφανίζουν μεγάλες τιμές όπως ήταν αναμενόμενο. Οι υπόλοιποι δρομολογητές κυμαίνονται στα ίδια επίπεδα περίπου.

6.2.3 Mean Job Speedup

Οι μεγαλύτερες τιμές παρατηρούνται στην co-scheduled εκδοχή του EASY, με μικρή διακύμανση (Σχήμα G7). The biggest mean speedup values are observed in co-scheduled EASY, with a small variance. Οι τιμές των υπόλοιπων δρομολογητών δεν φαίνεται να διαφέρουν, με τις μεγαλύτερες διακυμάνσεις να παρατηρούνται στους SJF-Filler και Pop-Filler. Επίσης, ο LJF-Co φαίνεται να επιτυγχάνει υψηλές τιμές και ο Popularity χαμηλότερες.



Σχήμα G6: Boxplot των Mean Slowdown.

6.2.4 Weighted Mean Job Speedup

Σε αυτό το σχήμα (Σχήμα G8) αποδίδεται το σταθμισμένο mean job speedup, όπου οι μεγάλες εργασίες έχουν μεγαλύτερη επιρροή. Οι τιμές του σταθμισμένου mean speedup είναι μικρότερες από το αστάθμιστο, γεγονός που υποδεικνύει ότι οι μικρότερες εργασίες ευνοούνται (πιθανόν λόγω backfill). Παρατηρούμε επίσης ότι οι δρομολογητές που αντιμετωπίζουν τις μεγάλες εργασίες με συνεπή τρόπο έχουν και αντίστοιχα μικρή διακύμανση στις τιμές.

6.2.5 Mean Slowdown per Processor

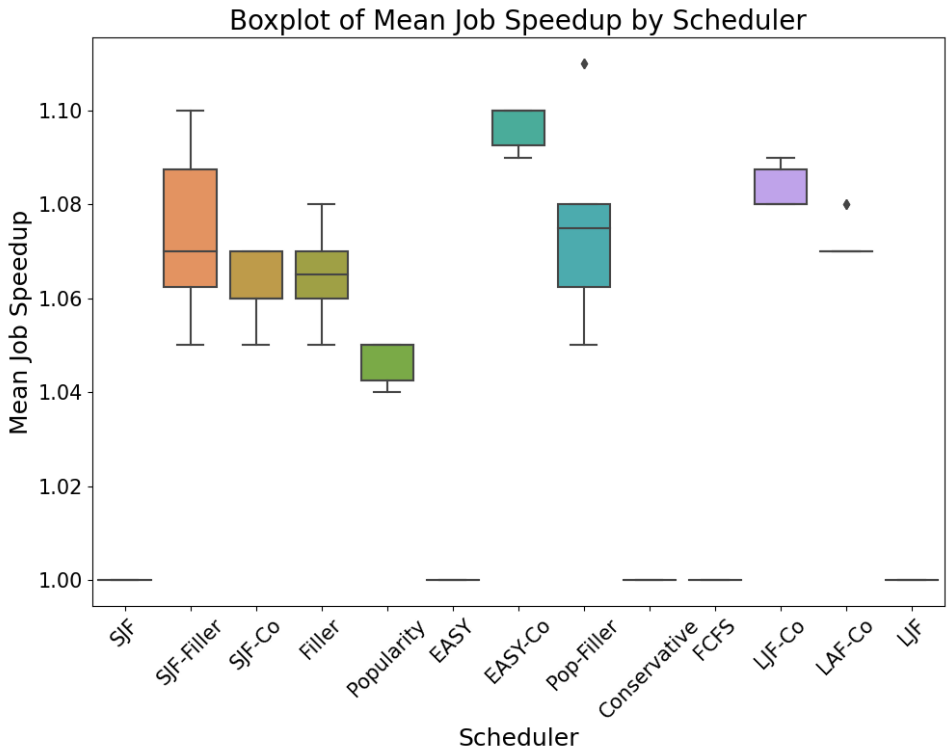
Ισχύουν οι ίδιες παρατηρήσεις με το mean slowdown, με μία ελαφρά διαφοροποίηση στον Pop-Filler, όπου οι τιμές είναι σχετικά μικρότερες (Σχήμα G9).

6.2.6 Slowdown Counts Percentage

Μελετώντας το ποσοστό των επιβραδυνόμενων εργασιών (Σχήμα G10), παρατηρούμε ότι οι περισσότεροι δρομολογητές παρουσιάσουν ποσοστά γύρω από το 23%, εκτός φυσικά από τον Popularity, που ως υβριδική υλοποίηση το ποσοστότου είναι περίπου το μισό.

6.2.7 Utilization

Αναφορικά με το utilization, τα αποτελέσματα είναι γενικώς αναμενόμενα (Σχήμα G10). Οι τρεις Filler εκδοχές κυριαρχούν, καθώς προσπαθούν ενεργά να ελαττώσουν τον κατακερματισμό. Την επίδοση τους ακολουθούν οι ”Large” δρομολογητές (LJF, LJF-Co and LAF-Co). Ενδιαφέρον παρουσιάζει το γεγονός ότι το utilization των SJF-Co και EASY-Co είναι χειρότερο από αυτό



Σχήμα G7: Boxplot των Mean Job Speedup.

των κλασσικών εκδοχών τους. Υποπτευόμαστε ότι αυτό οφείλεται στα κενά πυ δημιουργούνται στο σύστημα, τα οποία μεγαλώνουν με τον διαμοιρασμό των πόρων συν το ότι στον SJF-Co, το backfill δεν προσφέρει σημαντική βετίωση, καθώς οι επόμενες εργασίες στην ουρά αναμονής θα έχουν μεγαλύτερο χρόνο εκτέλεσης.

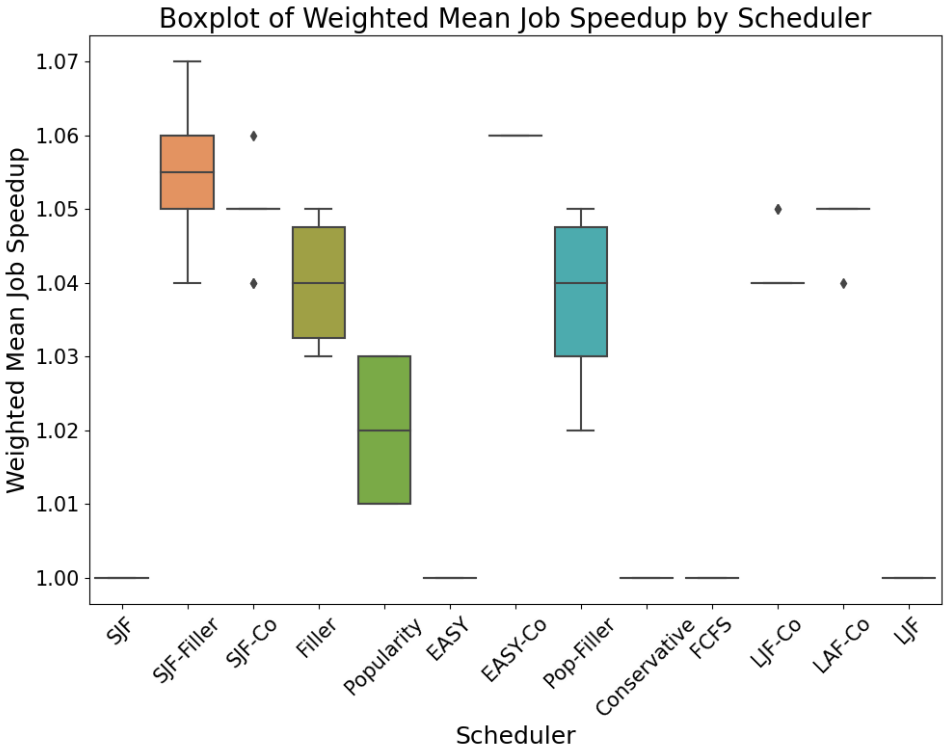
6.3 Pareto Plot

Παρουσιάζουμε το βασικό διάγραμμα Pareto (Σχήμα G12) όπου περιέχει τα δεδομένα από όλα τα πειράματα και οι δύο άξονες αποτελούνται από το mean slowdown και το makespan speedup [33]. Η διακεκομμένη γραμή αναπαριστά το μέτωπο Pareto, το οποίο αποτελείται από τις βέλτιστες λύσεις. Είναι εμφανής η κυριαρχία του SJF-Filler, στο σύνολο βέλτιστων λύσεων. Ακόμη, οι δύο υλοποίησιες στις οποίες βασίζεται ο SJF-Filler δηλαδή ο SJF και ο Filler επιτυγχάνουν καλές επιδόσεις.

6.4 Συσχέτιση Μετρικών

Ακόμη παρουσιάζουμε την συσχέτιση των μετρικών (Σχήμα G13) [34]. Παρατηρούμε ότι το makespan speedup εμφανίζει υψήλη συσχέτιση με το utilization, δηλαδή με την βελτίωση του utilization πετυχαίνουμε μείωση στο makespan. Επιπλέον, τα mean job speedup και weighted mean job speedup και αντιστοίχως τα mean slowdown και mean slowdown per processor έχουν πολύ υψηλές τιμές συσχέτισης, καθώς μετρούν ίδια μεγέθη.

Από την άλλη, το makespan speedup έχει πολύ μικρή τιμή συσχέτισης με το mean slowdown, γεγονός που επιβεβαιώνει την επιλογή μας να θέσουμε αυτές τις μετρικές ως μετρικές στόχους,



Σχήμα G8: Boxplot των Weighted Mean Job Speedup.

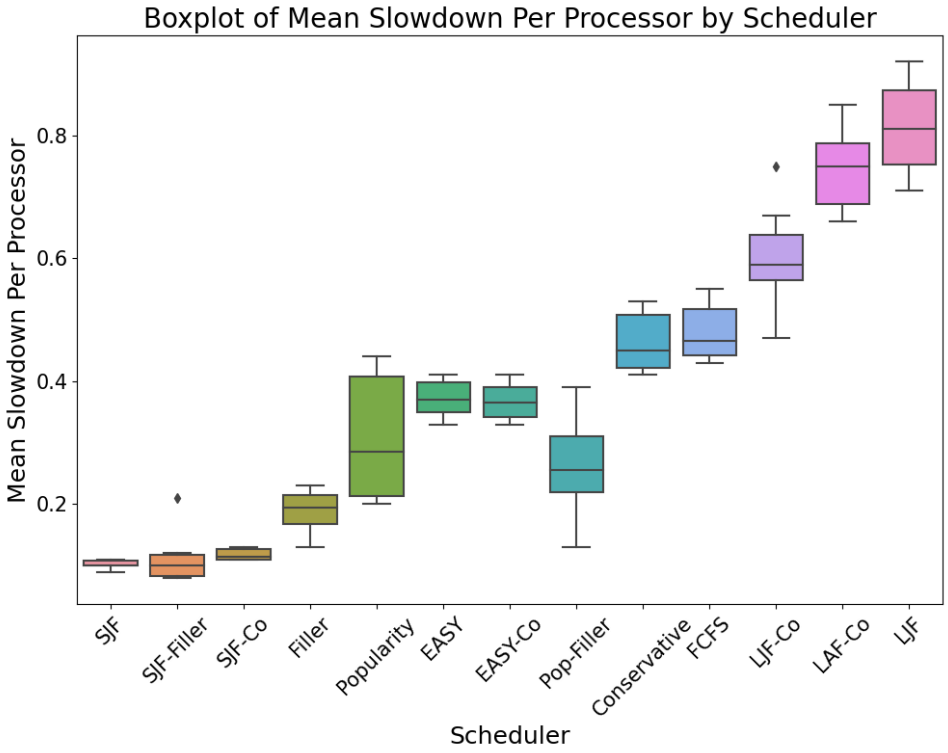
καθώς είναι ανεξάρτητες.

Τέλος, παρατηρούμε υψηλή συσχέτιση του slowdown counts percentage με το mean job speedup, το οποίο στην αρχή μπορεί να μας φαίνεται παράδοξο. Μία ερμηνεία που μπορούμε να δώσουμε σε αυτό είναι ότι όταν σε ένα ζεύγος συνεκτέλεσης η μία εργασία επιταχύνει την εκτέλεση της, το ζεύγος της πιθανόν να επιβραδύνει, με την επιτάχυνση ωστόσο να είναι μεγαλύτερη.

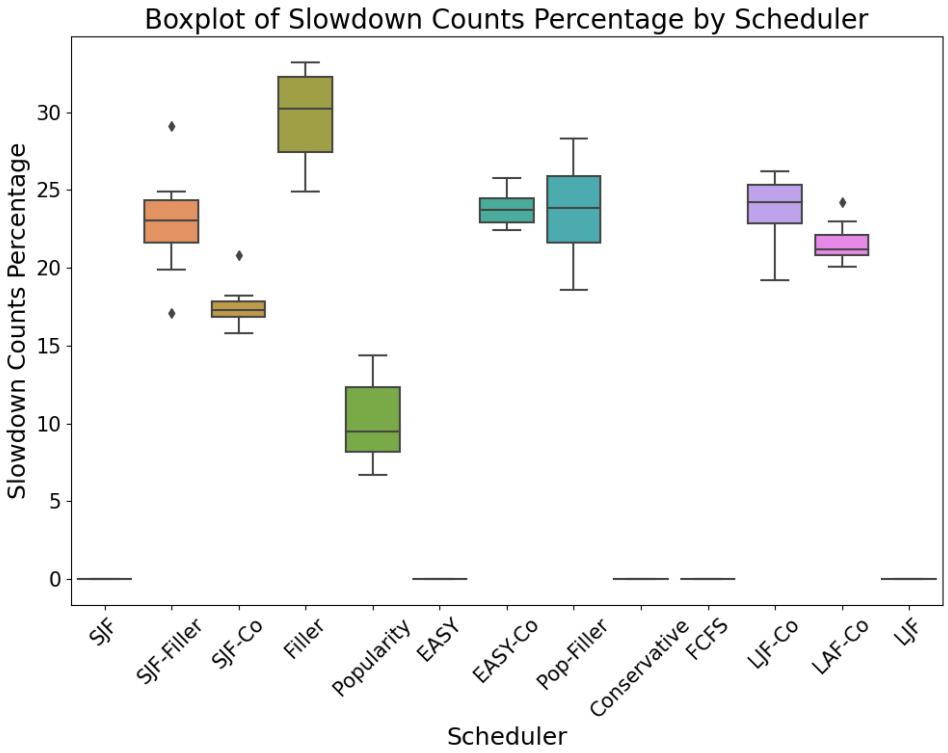
7 Συμπεράσματα

Συνοψίζοντας, στην παρούσα διπλωματική εργασία εξετάστηκε η επίδραση της συνδρομολόγησης σε υπάρχοντες αλγορίθμους χρονοδρομολόγησης, όπου φάνηκε πως ήταν ιδιαίτερα θετική για το σύστημα. Συγκεκριμένα, παρατηρήθηκε μία ελάττωση στον συνολικό χρόνος εκτέλεσης του φορτίου εργασιών, λόγω της καλύτερης αξιοποίησης των πόρων και της μείωσης του κατακερματισμού. Από την άλλη, η μέση επιτάχυνση του χρόνου εκτέλεσης φαίνεται να είναι μεγαλύτερητης μονάδας, επομένως για πλήθος εργασιών έχουμε επιτάχυνση, συχρόνως όμως παρατηρείται και μία επιβράδυνση (μεγαλύτερος χρόνος εκτέλεσης) στις εργασίες της τάξης περίπου του 23%.

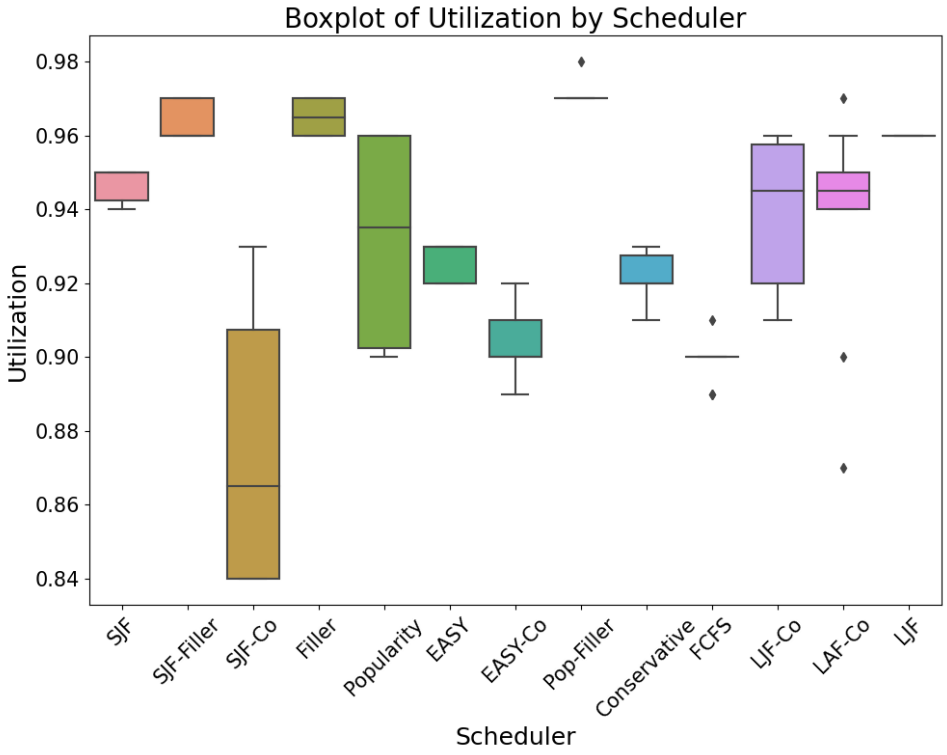
Ακόμη, αναπτύχθηκαν νέοι αλγόριθμοι συνδρομολόγησης, οι οποίοι φάνηκε να εξισορροπούν καλύτερα στους δύο άξονες βελτιστιποίησης (αποδοτικότητα συστήματος και ικανοποίηση χρηστών). Συγκεκριμένα, οι υλοποιήσεις μας δύο παραγόντων (Two Factors), πέτυχαν τις καλύτερες επιδόσεις, με τον SJF-Filler να πρωταγωνιστεί.



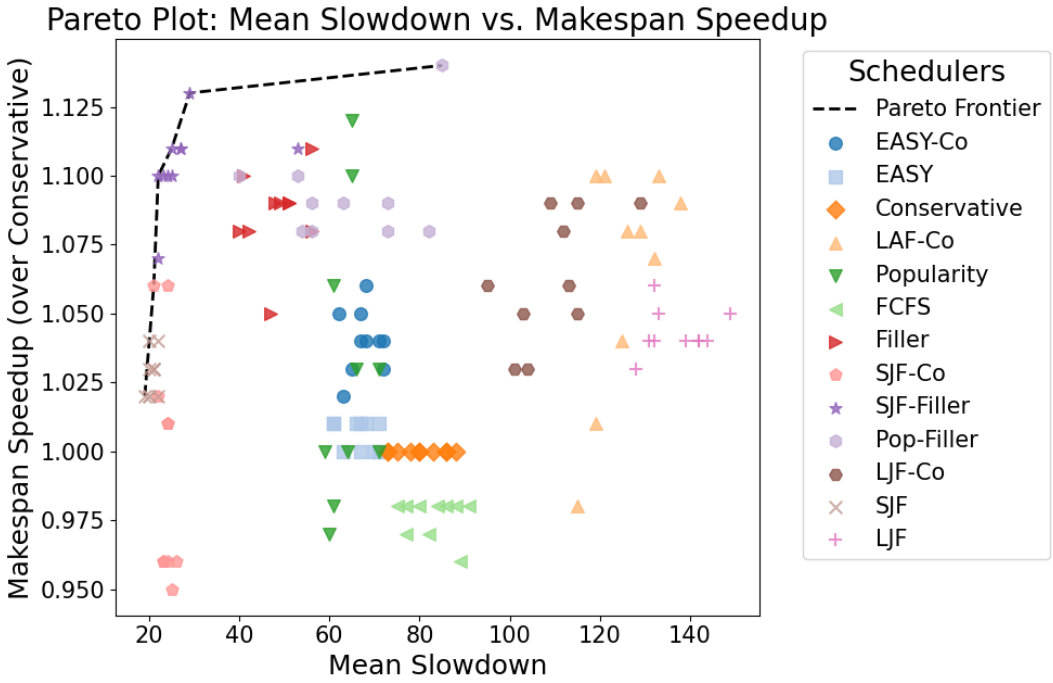
Σχήμα G9: Boxplot του Mean Slowdown ανά πυρήνα.



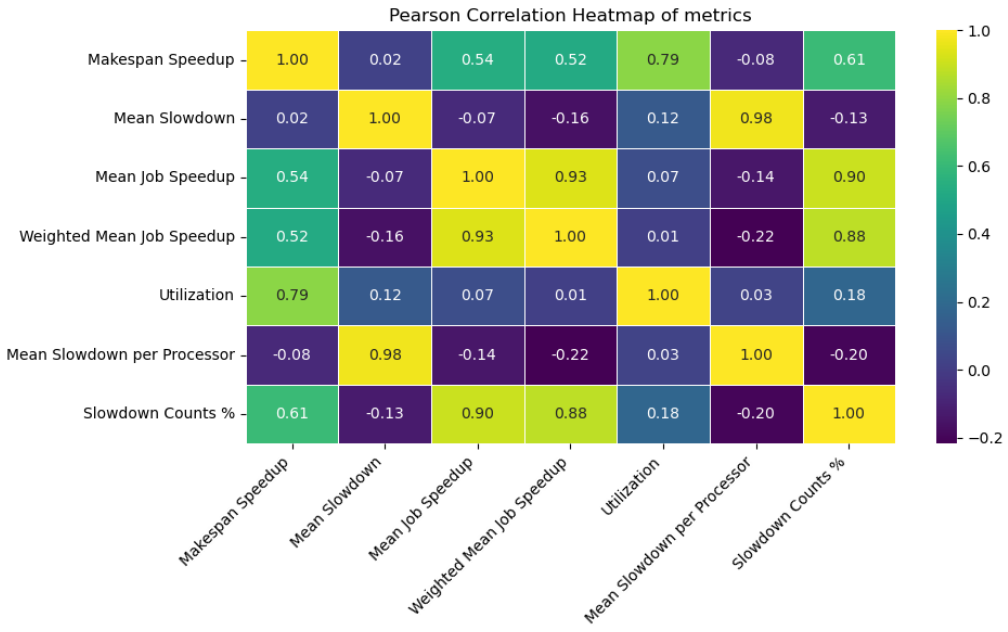
Σχήμα G10: Boxplot του ποσοστού των Slowdown Counts.



Σχήμα G11: Boxplot του Utilization.



Σχήμα G12: Διάγραμμα Pareto με το mean slowdown και makespan speedup ως στόχους.



Σχήμα G13: Heatmap συσχέτισης μετρικών.

Bibliography

[1] 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, Dec. 2013. DOI: https://doi.org/10.1002/cpe.3187.

[2] J. Breitbart, J. Weidendorfer, and C. Trinitis, "Case study on co-scheduling for hpc applications", in *2015 44th International Conference on Parallel Processing Workshops*, 2015, pp. 277–285. DOI: 10.1109/ICPPW.2015.38.

[3] 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: https://doi.org/10.14459/2018md1428535. [Online]. Available: https://www.researchgate.net/publication/322662796_Node_Sharing_for_Increased_Throughput_and_Shorter_Runtimes_-_an_Industrial_Co-Scheduling_Case_Study.

[4] GRNET, *Introduction to high performance computing systems and aris system*, 2015. [Online]. Available: https://www.hpc.grnet.gr/supercomputer/.

[5] I. S. U. of Science and Technology, *What is an hpc cluster | high performance computing*. [Online]. Available: https://www.hpc.iastate.edu/guides/introduction-to-hpc-clusters/what-is-an-hpc-cluster.

[6] OpenMP, *Home*. [Online]. Available: https://www.openmp.org/.

[7] Jul. 2021. [Online]. Available: https://en.wikipedia.org/wiki/Message_Passing_Interface.

[8] D. Liang, P.-J. Ho, and B. Liu, "Scheduling in distributed systems", 2000.

[9] 2013. [Online]. Available: https://top500.org/lists/top500/.

[10] W. Contributors, *Slurm workload manager*, Nov. 2019. [Online]. Available: https://en.wikipedia.org/wiki/Slurm_Workload_Manager.

[11] SLURM, 2024. [Online]. Available: https://slurm.schedmd.com/sched_config.html.

[12] A. Burkimsher, I. Bate, and L. S. Indrusiak, "A survey of scheduling metrics and an improved ordering policy for list schedulers operating on workloads with dependencies and a wide variation in execution times", *Future Generation Computer Systems*, vol. 29, no. 8, pp. 2009–2025, 2013, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2012.12.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X12002257.

[13] A. V. Goponenko, K. Lamar, C. Peterson, B. A. Allan, J. M. Brandt, and D. Dechev, "Metrics for packing efficiency and fairness of hpc cluster batch job scheduling", in *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2022, pp. 241–252. DOI: 10.1109/SBAC-PAD55451.2022.00035.

[14] R. Boëzennec, F. Dufossé, and G. Pallez, "Qualitatively Analyzing Optimization Objectives in the Design of HPC Resource Manager", working paper or preprint, Aug. 2023. [Online]. Available: https://hal.science/hal-04187517.

[15] D. Carastan-Santos and R. Y. de Camargo, "Obtaining dynamic scheduling policies with simulation and machine learning", in *SC17: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–13.

[16] D. Feitelson, "Metrics for parallel job scheduling and their convergence", vol. 2221, Jul. 2001, ISBN: 978-3-540-42817-6. DOI: 10.1007/3-540-45540-X_11.

[17] J. Weinberg, "Job scheduling on parallel systems", in *Job Scheduling Strategies for Parallel Processing*, vol. 5, 2002, pp. 67–73.

[18] D. Feitelson, "Metric and workload effects on computer systems evaluation", *Computer*, vol. 36, no. 9, pp. 18–25, 2003. DOI: 10.1109/MC.2003.1231190.

[19] J. Li, G. Michelogiannakis, B. Cook, D. Cooray, and Y. Chen, *Analyzing Resource Utilization in an HPC System: A Case Study of NERSC Perlmutter*, 2023. arXiv: 2301.05145 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2301.05145.

[20] S. Maloney, E. Suarez, N. Eicker, F. Guimarães, and W. Frings, "Analyzing hpc monitoring data with a view towards efficient resource utilization", in *2024 IEEE 36th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2024, pp. 170–181. DOI: 10.1109/SBAC-PAD63648.2024.00023.

[21] F. V. Zacarias, V. Petrucci, R. Nishtala, P. M. Carpenter, and D. Mossé, "Intelligent colocation of HPC workloads", *CoRR*, vol. abs/2103.09019, 2021. arXiv: 2103.09019. [Online]. Available: https://arxiv.org/abs/2103.09019.

[22] A. Blanche and T. Lundqvist, "Terrible twins: A simple scheme to avoid bad co-schedules", Jan. 2016. DOI: 10.14459/2016md1286952.

[23] A. Blanche and T. Lundqvist, "Disallowing same-program co-schedules to improve efficiency in quad-core servers", Jan. 2017.

[24] A. D. Breslow, A. Tiwari, M. Schulz, L. Carrington, L. Tang, and J. Mars, "Enabling fair pricing on hpc systems with node sharing", in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12. DOI: 10.1145/2503210.2503256.

[25] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There goes the neighborhood: Performance degradation due to nearby jobs", in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12. DOI: 10.1145/2503210.2503247.

[26] J. Hall, A. Lathi, D. Lowenthal, and P. Tapasya, "Evaluating the potential of coscheduling on high-performance computing systems", in Sep. 2023, pp. 155–172, ISBN: 978-3-031-43942-1. DOI: 10.1007/978-3-031-43943-8_8.

[27] D. Bailey *et al.*, "The nas parallel benchmarks", *The International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991. DOI: 10.1177/109434209100500306. eprint: https://doi.org/10.1177/109434209100500306. [Online]. Available: https://doi.org/10.1177/109434209100500306.

[28] J. Weinberg, "Job scheduling on parallel systems", Jan. 2006.

[29] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan, "Characterization of backfilling strategies for parallel job scheduling", Feb. 2002, pp. 514–519, ISBN: 0-7695-1680-7. DOI: 10.1109/ICPPW.2002.1039773.

[30] D. Tsafrir, Y. Etsion, and D. Feitelson, "Backfilling using system-generated predictions rather than user runtime estimates", *Parallel and Distributed Systems, IEEE Transactions on*, vol. 18, pp. 789–803, Jul. 2007. DOI: 10.1109/TPDS.2007.70606.

[31] A. Rajbhandary, D. Bunde, and V. Leung, "Variations of conservative backfilling to improve fairness", vol. 8429, Jun. 2014, pp. 177–191, ISBN: 978-3-662-43778-0. DOI: 10.1007/978-3-662-43779-7_10.

[32] D. Carastan-Santos, R. Y. De Camargo, D. Trystram, and S. Zrigui, "One can only gain by replacing easy backfilling: A simple scheduling policies case study", in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019, pp. 1–10. DOI: 10.1109/CCGRID.2019.00010.

[33] J. Hakanen and R. Allmendinger, "Multiobjective optimization and decision making in engineering sciences", *Optimization and Engineering*, vol. 22, Jun. 2021. DOI: 10.1007/s11081-021-09627-x.

[34] J. Hauke and T. Kossowski, "Comparison of values of pearson's and spearman's correlation coefficients on the same sets of data", *Quaestiones Geographicae*, vol. 30, no. 2, pp. 87–93, 2011. DOI: 10.2478/v10117-011-0021-1. [Online]. Available: https://doi.org/10.2478/v10117-011-0021-1.

[35] W. H. Kruskal, "Ordinal measures of association", *Journal of the American Statistical Association*, vol. 53, no. 284, pp. 814–861, 1958. DOI: 10.1080/01621459.1958.10501481. eprint: https://www.tandfonline.com/doi/pdf/10.1080/01621459.1958.10501481. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/01621459.1958.10501481.

Συναρτήσεις ταξινόμησης ουράς αναμονής

SJF/SJF-Co

```
def waiting_queue_reorder(self, job: Job) -> float:
    return -float(job.remaining_time)
```

LJF/LJF-Co

```
def waiting_queue_reorder(self, job: Job) -> float:
    return float(job.remaining_time)
```

LAF-Co

```
def waiting_queue_reorder(self, job: Job) -> float:
    return float(job.num_of_processes * job.remaining_time)
```

Popularity

```
def waiting_queue_reorder(self, job: Job) -> float:
    return self.ranks[job.job_id] / len(self.cluster.waiting_queue)
```

Filler

```
def waiting_queue_reorder(self, job: Job) -> float:
    # The job that is closer to cover the gaps is more preferrable
    sys_free_cores = self.cluster.get_idle_cores()
    if sys_free_cores > 0:
        diff = sys_free_cores - job.num_of_processes
        if diff > 0:
            factor0 = 1 - (diff/sys_free_cores)
        elif diff == 0:
            factor0 = 1
        else:
            factor0 = -1
    else:
        factor0 = 1
    factor1 = ((job.job_id + 1) / len(self.cluster.waiting_queue))
    return factor0 / factor1
```

SJF-Filler

```
def get_index(self, job: Job) -> int:
    positions = [index for index, value in
     ↪  enumerate(self.cluster.second_queue) if value.job_id == job.job_id]
    return positions[0]

def waiting_queue_reorder(self, job: Job) -> float:
    # The job that is closer to cover the gaps is more preferrable
    sys_free_cores = self.cluster.get_idle_cores()
    if sys_free_cores > 0:
        diff = sys_free_cores - job.num_of_processes
        if diff > 0:
            factor0 = 1 - (diff/sys_free_cores)
        elif diff == 0:
            factor0 = 1
        else:
            factor0 = -1
    else:
        factor0 = 1
    factor1 = self.get_index(job) / len(self.cluster.waiting_queue)
    return factor0 + factor1

def second_queue_reorder(self, job: Job) -> float:
    return job.remaining_time
```

Pop-Filler

```
def get_index(self, job: Job) -> int:
    positions = [index for index, value in
     ↪  enumerate(self.cluster.second_queue) if value.job_id == job.job_id]
    return positions[0]

def waiting_queue_reorder(self, job: Job) -> float:
    # The job that is closer to cover the gaps is more preferrable
    sys_free_cores = self.cluster.get_idle_cores()
    if sys_free_cores > 0:
        diff = sys_free_cores - job.num_of_processes
        if diff > 0:
            factor0 = 1 - (diff/sys_free_cores)
        elif diff == 0:
            factor0 = 1
        else:
            factor0 = -1
    else:
        factor0 = 1
    factor1 = self.get_index(job) / len(self.cluster.waiting_queue)
    return factor0 + factor1

def second_queue_reorder(self, job: Job) -> float:
    return self.ranks[job.job_id]
```