



NATIONAL TECHNICAL UNIVERSITY OF ATHENS  
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING  
DIVISION OF COMPUTER SCIENCE

# Performance Modeling for Co-Scheduling in HPC Systems

DIPLOMA THESIS

of

ATHANASIOS TSOUKLEIDIS-KARYDAKIS

**Supervisor:** Georgios Goumas  
Associate Professor NTUA

Athens, February 2025

---





NATIONAL TECHNICAL UNIVERSITY OF ATHENS  
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING  
DIVISION OF COMPUTER SCIENCE

# Performance Modeling for Co-Scheduling in HPC Systems

DIPLOMA THESIS

of

ATHANASIOS TSOUKLEIDIS-KARYDAKIS

**Supervisor:** Georgios Goumas  
Associate Professor NTUA

Approved by the examination committee on 21st February 2025.

*(Signature)*

*(Signature)*

*(Signature)*

.....  
Georgios Goumas  
Associate Professor NTUA

.....  
Nectarios Koziris  
Professor NTUA

.....  
Dionisios Pnevmatikatos  
Professor NTUA

Athens, February 2025



Copyright © – All rights reserved.

Athanasios Tsoukleidis-Karydakis, 2025.

The copying, storage and distribution of this diploma thesis, exall or part of it, is prohibited for commercial purposes. Reprinting, storage and distribution for non - profit, educational or of a research nature is allowed, provided that the source is indicated and that this message is retained.

The content of this thesis does not necessarily reflect the views of the Department, the Supervisor, or the committee that approved it.

## **DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS**

Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism.

*(Signature)*

.....

Athanasios Tsoukleidis-Karydakis

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

21st February 2025



Η συνδρομολόγηση (co-scheduling) εργασιών σε Υπολογιστικά Συστήματα Υψηλής Επίδοσης (High Performance Computing - HPC) προσφέρει σημαντικές δυνατότητες για τη βελτίωση της συνολικής ρυθμαπόδοσης (throughput) και της ενεργειακής αποδοτικότητας του συστήματος. Ωστόσο, η διεκδίκηση κοινών πόρων στους κόμβους μπορεί να οδηγήσει σε υποβάθμιση της απόδοσης, επιβραδύνοντας τις εργασίες και αναιρώντας αυτά τα οφέλη. Για την αντιμετώπιση αυτής της πρόκλησης, είναι απαραίτητη η ανάπτυξη προηγμένων αλγορίθμων συνδρομολόγησης, οι οποίοι απαιτούν μια εις βάθος κατανόηση των εφαρμογών που εκτελούνται, ώστε να λαμβάνονται τεκμηριωμένες αποφάσεις δρομολόγησης.

Στην παρούσα διπλωματική εργασία, ταξινομούμε και παρουσιάζουμε μοντέλα απόδοσης που μπορούν να αξιοποιηθούν για τη βελτίωση των στρατηγικών συνδρομολόγησης. Οι προτεινόμενες μέθοδοι επικεντρώνονται είτε στην κατηγοριοποίηση των εφαρμογών μέσω συγκεκριμένων ετικετών (tags) είτε στην πρόβλεψη της επιτάχυνσης (speedup) ή της επιβράδυνσής (slowdown) τους όταν συν-εκτελούνται με άλλα φορτία εργασίας. Για την επίτευξη αυτού του στόχου, διερευνούμε τόσο εμπειρικές προσεγγίσεις όσο και τεχνικές Μηχανικής Μάθησης, αναλύοντας τα πλεονεκτήματα και τους περιορισμούς τους. Επιπλέον, συζητούμε τις κύριες συμβιβαστικές αποφάσεις που προκύπτουν κατά την επιλογή και ανάπτυξη του καταλληλότερου μοντέλου για πρόβλεψη συν-εκτέλεσης σε περιβάλλοντα HPC. Στη συνέχεια, παρουσιάζουμε προκαταρκτικά αποτελέσματα που αποδεικνύουν την αποτελεσματικότητα κάθε μοντέλου μέσω αντιπροσωπευτικών παραδειγμάτων από διαφορετικές κατηγορίες μοντέλων. Τέλος, αξιολογούμε τη δυνατότητα της συνδρομολόγησης να βελτιώσει τον συνολικό χρόνο εκτέλεσης (makespan) ενός χρονοπρογράμματος εργασιών, καθώς και τους συμβιβασμούς που απαιτούνται για την εξισορρόπηση της απόδοσης του συστήματος και της ικανοποίησης των χρηστών σε περιβάλλοντα HPC.

## Λέξεις Κλειδιά

Co-Scheduling, High Performance Computing, Performance Analysis, Machine Learning, Profiling, perf, mpiP, MPI



Co-Scheduling jobs in High Performance Computing (HPC) systems offers significant potential to improve system throughput and energy efficiency. However, resource contention in shared node resources can introduce performance degradation, leading to job slowdowns and counteracting these benefits. To address this challenge, sophisticated co-scheduling algorithms must be developed, requiring a good understanding of the submitted applications to make informed scheduling decisions.

In this thesis, we classify and present a number of performance models that can be leveraged to support advanced co-scheduling strategies. The methods focus on either assigning specific ‘tags’ to applications or predicting their potential speedup or slowdown when co-executed with other workloads. To achieve this, we explore both empirical approaches alongside Machine Learning-based techniques, assessing their respective benefits and limitations. Furthermore, we discuss key trade-offs that arise when selecting and building the most suitable model for co-location prediction in HPC environments. We then provide preliminary results demonstrating the effectiveness of each model through representative examples across multiple model categories. Finally, we provide an initial evaluation of co-scheduling’s potential to enhance the makespan of a given schedule, as well as the trade-offs involved in balancing system performance and user satisfaction in HPC systems.

## Keywords

Co-Scheduling, High Performance Computing, Performance Analysis, Machine Learning, Profiling, perf, mpiP, MPI





---

## Acknowledgements

---

First and foremost, I would like to express my gratitude to my thesis supervisor, Mr. Giorgos Goumas, for his support and guidance throughout this diploma thesis. I also appreciate the knowledge he shared with me over the past year and during the courses he taught in the undergraduate program, which sparked my interest in this technological field. I would also like to thank PhD candidates Nikos Triantafyllis and Stratos Karapanagiotis for our excellent collaboration, both in this thesis and in our other research endeavors. Furthermore, I am grateful to my family for their support throughout both my academic and personal journey, without which reaching this milestone would not have been possible. Last but not least, I would like to thank my friends for the wonderful moments we shared during our undergraduate years.

Athens, February 2025

Athanasios Tsoukleidis-Karydakís



<b>1</b>	<b>High Performance Computing – An overview</b>	<b>12</b>
1.1	Background . . . . .	12
1.2	Scientific Applications . . . . .	12
1.3	Multiprocessing Systems . . . . .	13
1.4	Challenges in HPC . . . . .	16
<b>2</b>	<b>Infrastructure, Benchmarking and Profiling</b>	<b>18</b>
2.1	Infrastructure . . . . .	18
2.2	Benchmarks . . . . .	22
2.3	Profiling . . . . .	24
<b>3</b>	<b>Co-Scheduling in HPC</b>	<b>32</b>
3.1	Metrics and Application Types . . . . .	32
3.2	Traditional Scheduling Techniques . . . . .	34
3.3	Co-Scheduling Overview . . . . .	37
3.4	Co-Scheduling Approaches . . . . .	39
<b>4</b>	<b>Performance Prediction in co-execution mode</b>	<b>43</b>
4.1	Profiling Methodology . . . . .	43
4.2	A Tag-Based Model . . . . .	45
4.3	Empirical Heatmap Prediction . . . . .	49
4.4	Heatmap Prediction using Machine Learning . . . . .	61
<b>5</b>	<b>Co-Scheduling Simulations</b>	<b>66</b>
5.1	Simple Experiments . . . . .	66
5.2	Towards Sophisticated Co-Schedulers . . . . .	68
5.3	Need for sophistication . . . . .	70
<b>6</b>	<b>Future Research Directions</b>	<b>72</b>
<b>7</b>	<b>Εκτεταμένη Περίληψη</b>	<b>75</b>

7.1	Επισκόπηση . . . . .	75
7.2	Υποδομή και Benchmarks . . . . .	78
7.3	Profiling . . . . .	80
7.4	Μετρικές και είδη εφαρμογών . . . . .	83
7.5	Παραδοσιακή Χρονοδρομολόγηση . . . . .	85
7.6	Συνδρομολόγηση και τεχνικές . . . . .	85
7.7	Μεθοδολογία profiling . . . . .	89
7.8	Tag-based μοντέλα . . . . .	91
7.9	Pairwise μοντέλα : Εμπειρικός classifier . . . . .	93
7.10	Pairwise μοντέλα : ML classifier . . . . .	98
7.11	Προσομοιώσεις Συνεκτέλεσης . . . . .	101
7.12	Μελλοντικές Ερευνητικές Κατευθύνσεις . . . . .	103
<b>Bibliography</b>		<b>111</b>

---

## List of Figures

---

1.1	Categories of applications submitted to the Cori supercomputer system during 2019	13
1.2	Comparison of UMA and NUMA architectures [2]	15
1.3	Distributed memory architecture [2]	15
1.4	Illustration of an HPC-oriented processor: the Knights Landing (KNL), 2nd Generation Intel® Xeon Phi™ Processor. This design integrates a high-bandwidth memory layer positioned close to the processor (on-package memory) for enhanced performance [3].	17
2.1	ARIS' nodes and their technical characteristics [4]	19
2.2	An overview of SLURM's components [5]	19
2.3	All SPEChpc 2021 benchmarks alongside their workload sizes and names, their implementation language, their lines of code (LOC) and their scientific domains. [6]	23
2.4	Results of the STREAM benchmark in a node of the ARIS supercomputer	24
2.5	A diagram depicting a profiling workflow utilizing Score-P, Scalasca and CUBE. The application is first instrumented, then an effort to reduce the overheads by filtering the regions of code that will be profiled is made, and afterwards the profiling (and tracing) files are produced. Finally, the results are being analyzed and visualized in the User Interface. [7]	27
2.6	Perf output for a specific process of an MPI benchmark	29
2.7	mpiP output for a specific MPI benchmark. Visible are an overview of the time spent in the entire benchmark and the MPI time percentage as well as the top 20 most time consuming MPI directives alongside their callsites	31
3.1	Visualization of backfilling [42]	36
3.2	Visualization of the three mentioned execution modes [1]	38
3.3	Communication patterns of two different NAS benchmarks when co-located in a 16-core node [8]	39
3.4	HPC performance characterization using Intel VTune [9]	40
3.5	Heatmap of speedups from the co-execution of eight NPB benchmarks of 256 processes each in the Marconi supercomputer	41

3.6	A brief taxonomy of application models for co-scheduling . . . . .	42
4.1	Spider plot for the SP benchmark . . . . .	45
4.2	Spider Co-Plots for a good and a bad co-location scenario . . . . .	47
4.3	Benefits of our tag-based approach in comparison with the baseline case . . . . .	48
4.4	Potential application placings in a real-world HPC scenario . . . . .	48
4.5	Heatmap of the used NAS benchmarks in a co-execution environment (Average Speedup : 1.12) . . . . .	50
4.6	Heatmap of the used SPEC benchmarks in a co-execution environment (Average Speedup : 1.14) . . . . .	51
4.7	Spider Plots for each one of the 8 utilized NAS benchmarks . . . . .	52
4.8	Comparison of BT and LU . . . . .	53
4.9	Communication patterns of two instances of the same program (FT benchmark) in co-execution mode [8] . . . . .	56
4.10	Spider Plots for each one of the 7 utilized SPEC benchmarks . . . . .	57
4.11	Decision Tree made by utilizing the aforementioned empirical rules . . . . .	59
4.12	Predicted categorical heatmap for the NAS benchmarks . . . . .	59
4.13	Predicted categorical heatmap for the SPEC benchmarks . . . . .	60
4.14	Evaluation of the best-performing models from each model type . . . . .	62
4.15	Impact of using all ML models in conjunction with our empirical model on accuracy and precision . . . . .	64
4.16	Pareto Plot of all our models and all possible combinations of models . . . . .	65
5.1	Makespan improvement for two co-scheduling algorithms with regards to the EASY Scheduler and for diverse process count mixes . . . . .	67
5.2	Correlation between Makespan Improvement and various metrics related to system throughput and user satisfaction . . . . .	69
7.1	Οπτικοποίηση του backfilling [42] . . . . .	86
7.2	Οπτικοποίηση των τριών τρόπων ανάθεσης πόρων [1] . . . . .	87
7.3	Heatmap των speedups από την συνεκτέλεση οκτώ NPB benchmarks με 256 processes το καθένα στον υπερυπολογιστή Marconi . . . . .	89
7.4	Ταξινόμηση των μοντέλων performance analysis για χρήση σε αλγόριθμους συνδρομολόγησης . . . . .	90
7.5	Spider plot για το SP benchmark . . . . .	91
7.6	Spider Co-Plots για ένα σενάριο καλής και ένα σενάριο κακής συν-τοποθέτησης . . . . .	92
7.7	Τα πλεονεκτήματα της resource-centric tag-based προσέγγισής μας σε σύγκριση με την baseline περίπτωση . . . . .	93
7.8	Heatmaps των NAS και SPEChepc 2021 benchmarks στον υπερυπολογιστή ARIS . . . . .	94
7.9	Δέντρο αποφάσεων με βάση τους ανωτέρω εμπειρικούς κανόνες . . . . .	97
7.10	Αξιολόγηση των καλύτερων μοντέλων από κάθε τύπο μοντέλου . . . . .	99
7.11	Pareto Plot όλων των μοντέλων μας και όλων των πιθανών συνδυασμών μοντέλων . . . . .	101

7.12	Βελτίωση του makespan για δύο αλγόριθμους συνδρομολόγησης σε σχέση με τον απλό EASY Scheduler και για διαφορετικές ποικιλίες από πλήθη διεργασιών . . . . .	103
7.13	Συσχέτιση μεταξύ της βελτίωσης του Makespan και διάφορων μετρικών που σχετίζονται με την ικανοποίηση του συστήματος και του χρήστη . . . . .	104



---

## High Performance Computing – An overview

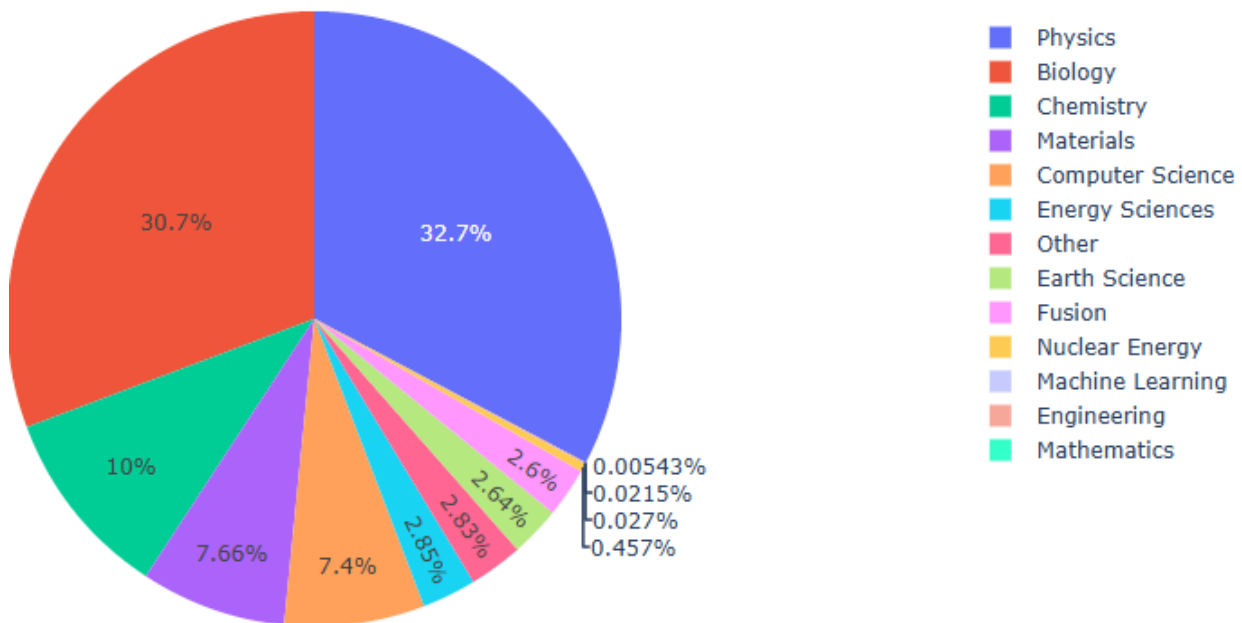
---

### 1.1 Background

In this day and age, a vast amount of applications require processing power and memory capabilities much bigger than what can be provided by a simple desktop computer. These limitations are overcome through the use of High Performance Computing (HPC) systems. HPC systems are made up of multiple servers or nodes linked together using a high-bandwidth network, to work as a single, massive computer and are able to perform millions of mathematical operations and process large volumes of data [10]. As a result, more and more countries are investing in their own supercomputers and HPC systems have emerged as a very active research field. Recent years have also seen the development of HPC as a Service (HPCaaS), that aims to provide HPC capabilities over the cloud, leveraging the existence of faster networks than before [11], as well as High Performance Edge Computing (HPEC), which extends HPC capabilities to edge hardware, enabling data processing directly at the edge and eliminating the overhead of transporting data to specialized facilities [32].

### 1.2 Scientific Applications

HPC systems have traditionally assisted in the execution of various scientific applications and simulations. Some examples include applications from the fields of biochemistry, chemistry, meteorology and computational fluid dynamics (CFD). In Figure 1.1, one can see the different categories of applications that were submitted for execution to the Cori supercomputer, located in the National Energy Research Scientific Computing Center (NERSC). The pie chart was made utilizing a dataset [33] from 2019, containing 6390408 jobs. It is evident that basic sciences applications constitute the majority of jobs submitted during that time period.



**Figure 1.1.** *Categories of applications submitted to the Cori supercomputer system during 2019*

Nowadays though, a great deal of new scientific fields that can greatly benefit from the capabilities of HPC systems have emerged. These include [11] :

- **Big Data** : The Big Data era demands the processing of a huge amount of data, originating from various sources and containing a great deal of legacy data.
- **Artificial Intelligence/Machine Learning** : AI/ML algorithms contain a learning phase, which can sometimes constitute a bottleneck in the execution. Additionally, many ML/DL strategies have recently begun being implemented in simulation problems.
- **Data Science** : Data Science's purpose is to extract knowledge and insights from large datasets. It includes data collection and storage, data processing and analysis and finally data visualization. All these tasks can be challenging without the proper infrastructure.

### 1.3 Multiprocessing Systems

A very crucial aspect of parallel processing is the types of computer architectures that are used. In 1966, Michael J. Flynn divided the available architectures into four categories, in what is known as "Flynn's Taxonomy" [12]:

- **Single Instruction, Single Data (SISD)** : A sequential computer, able to execute one instruction at a time on a given set of data stored in a single memory. This architecture was

utilized in early personal computers (PCs) where no parallelism could be exploited.

- **Single Instruction, Multiple Data (SIMD)** : A single instruction is applied at the same time on multiple different data streams, exploiting data-level parallelism. A common example of this architecture in use is in GPUs (Graphics Processing Units).
- **Multiple Instruction, Single Data (MISD)** : Multiple instruction are applied on the same data. This particular architecture is uncommon and has been primarily used for fault tolerance purposes.
- **Multiple instruction, Multiple Data (MIMD)** : Multiple units execute different instructions on different data streams at the same time. This particular architecture is the most prominent one for HPC systems.

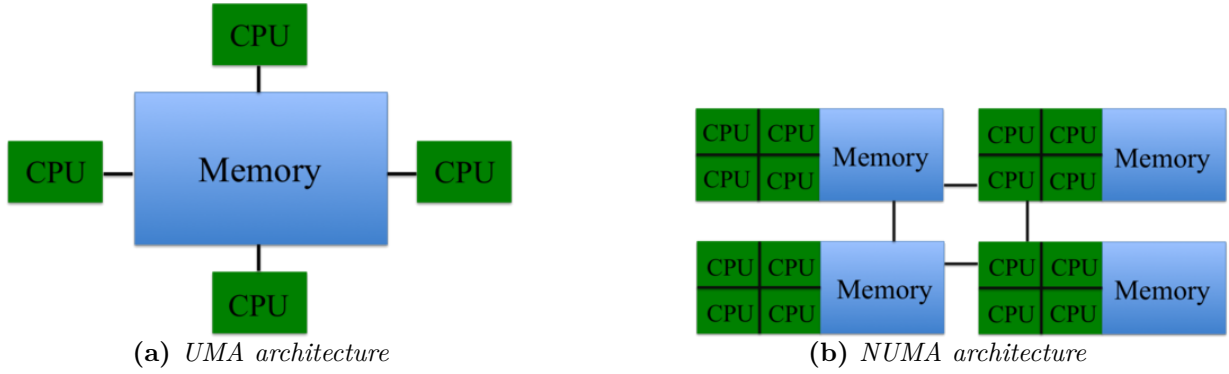
Parallel processing systems can be further categorized based on how they use the memory system into **shared memory systems** and **distributed memory systems**.

### 1.3.1 Shared Memory Systems

In a shared memory system, all processors access the same memory. Thus, any change to a particular variable stored in this memory is visible by all processors. Each processor accesses the memory using the common memory bus. Shared memory systems can be divided into two categories [2]:

- **Uniform Memory Access (UMA)** : In this case, all processors have equal access times to the common memory. These systems are also known as Symmetric Multiprocessor (SMP) machines.
- **Non-Uniform Memory Access (NUMA)** : In this case, each processor has its own local memory that can be accessed fast. At the same time, it can also access the memories of the other processors, albeit with a slower access time. A NUMA system can also be seen as multiple SMPs linked together.

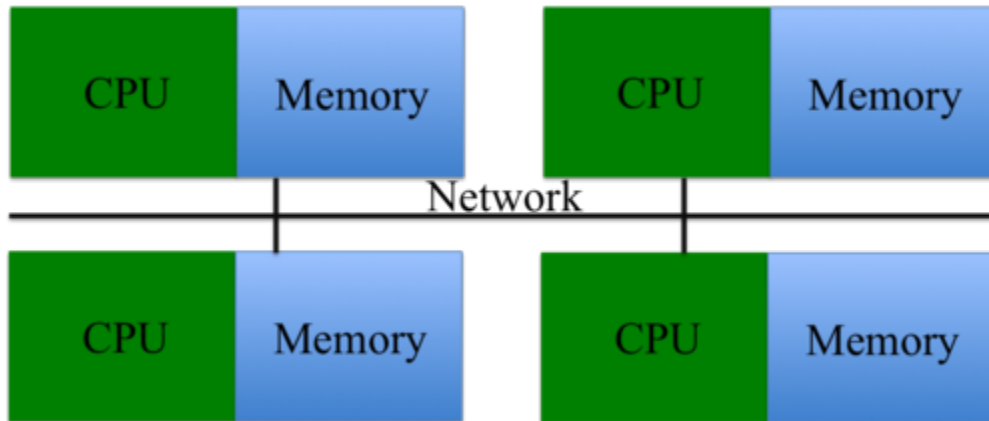
The most famous application programming interface for shared memory systems is called OpenMP. While shared memory systems enable multiple processors to communicate easily through simple load/store commands on shared data, the common memory bus and limited memory bandwidth can become a bottleneck for the performance of the system. NUMA systems mitigate this problem to a certain degree but also come with the need for better data locality so as to avoid the latency of accessing data that are present in another processor's local memory. Nevertheless, scalability may prove to be a challenge, both for UMA and for NUMA systems.



**Figure 1.2.** Comparison of UMA and NUMA architectures [2]

### 1.3.2 Distributed Memory Systems

Distributed memory systems consist of multiple nodes, with each one of them containing its own memory. A particular node can access the memory of another node only through the interconnection network. This architecture can be also called "shared-nothing" and encompasses both Massively Parallel Processors (MPP) and clusters. A cluster can be defined as a system that consists of multiple, independent nodes, also capable of independent operation [34], in contrast to MPPs, where the nodes are more tightly-coupled and are less "general-purpose". An interesting subcategory of clusters are **commodity clusters**, for which all components (compute units, network etc.) are commercial, "off the shelf" products. These clusters have a low cost, scale easily and also mostly use open-source software [34]. One such example are the Beowulf-class systems.



**Figure 1.3.** Distributed memory architecture [2]

Communication between the nodes in a distributed memory system is accomplished through the use of message-passing standards, with the most prominent one being MPI. Distributed memory systems can scale up to hundreds or thousands of nodes without the common memory bus bottleneck. Their drawback lies in the need for a high-bandwidth interconnection network and the potentially big cost of communication between the nodes. For this reason, many different network topologies have been thoroughly studied. Modern supercomputers mostly use a hybrid model, where each node uses a shared memory architecture and the nodes are connected using an interconnection

network. This way, more parallelism can be exploited.

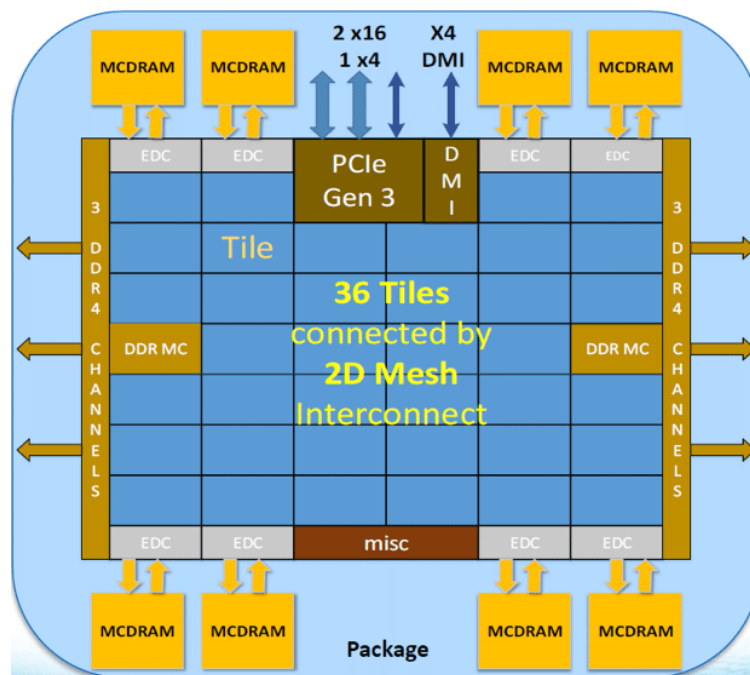
## 1.4 Challenges in HPC

The constant need for growing performance has given birth to many challenges in modern HPC systems as described in [11]. Firstly, most HPC systems nowadays are designed with a heterogeneous architecture, containing among others, CPUs, GPUs, a combination of memories such as DRAM and SRAM, FPGAs and application-specific integrated circuits (ASICs). Systems on Chip (SoC), that consist of multiple units (processors, memories, accelerators etc.) on the same chip are also widely used in Internet of Things and Edge Computing environments. Furthermore, the rapid development of Artificial Intelligence and Machine Learning and their growing needs for high performance during their training and inference phases has led to the development of AI-specialized computer hardware ("AI chips", Neural Processing Units (NPUs) and more). Last but not least, with the emergence of quantum computing, the incorporation of quantum processing units in classical HPC systems may soon become the rule rather than the exception. For now quantum operations usually come with many errors during execution. However, the remarkable speed and computational potential of quantum machines firmly establish them as a technology poised to drive the next major advancements in computing. Given all this heterogeneity, the need for new, simple programming models for software development in these environments is imperative. In this context, Intel released oneAPI in 2018, which simplifies software development across accelerators in heterogeneous environments.

Another challenge in modern HPC systems is energy efficiency and conservation. HPC systems are known to consume a huge amount of energy for their operations. Astonishingly though, as mentioned in [11], only a considerably small amount of the consumed energy is actually spent on the execution of the instructions of a parallel application. Most energy seems to be spent on leakages and on memory accesses through the complex memory hierarchies. This is verified in [13], where idle power was significant and high-memory applications led to more memory consumption than non-high-memory ones. Leakages and idle power pose a challenge for processor manufacturers, while the impact of memory accesses on energy consumption showcases the need for a scheduling mechanism that maintains data locality of applications or the need for Processing In Memory (PIM).

Moreover, in order to ensure user satisfaction, the system needs to continue operating even if failures in one or more parts of the system occur (resilience). In modern HPC systems with a much greater number of nodes than before and highly complex, heterogeneous architectures, failures happen much more frequently and resilience becomes a difficult feat to achieve.

In any case, addressing all challenges at once is virtually impossible and as a result most works concentrate on improving only a few functionalities or requirements. The questions of improving



**Figure 1.4.** Illustration of an HPC-oriented processor: the Knights Landing (KNL), 2nd Generation Intel® Xeon Phi™ Processor. This design integrates a high-bandwidth memory layer positioned close to the processor (on-package memory) for enhanced performance [3].

performance and energy efficiency sometimes come down to finding a better resource management and scheduling mechanism. In this thesis, we will concentrate on enhancing application and system performance through co-scheduling applications on supercomputer nodes. We will study ways to predict the behavior of applications in a co-scheduling environment and we will evaluate co-scheduling as a means to improve performance. Additionally, we will investigate emerging trade-offs within this more complex co-scheduling framework.

---

## Infrastructure, Benchmarking and Profiling

---

Before delving into co-scheduling of applications in HPC systems, we will go through all the infrastructure and benchmarks used for the experiments that will be presented in this thesis. Furthermore, an overview of application profiling in HPC environments will be provided, along with the specific methodology employed in the experiments presented here.

### 2.1 Infrastructure

#### 2.1.1 ARIS supercomputer

The experiments for this work were conducted using the ARIS supercomputer operated by GR-NET S.A (National Infrastructures for Research and Technology S.A.) in Athens, Greece [4]. ARIS ranked 468 in the top500 list when it was first installed in June 2015. It consists of 532 computational nodes, separated in the partitions shown in Figure 2.1, alongside their characteristics. Additionally, ARIS has an x86-64 architecture with Redhat/Centos 6.7 operating system and the nodes are connected using an Infiniband FDR network with a bandwidth of 56Gb/s. The nodes are connected using a fat tree topology. Users can connect to one of the two login nodes of the system using a Secure Shell (SSH) connection and from there submit their jobs to the SLURM Workload Manager. ARIS uses the environment module approach, where users must load their modules (software, compilers, libraries) so as to use them.

For the experiments presented in this work, the thin nodes were utilized and each experiment was taken multiple times within a ten or fifteen minute interval. Afterwards, the median value from the multiple repetitions of the experiment was retained to minimize the influence of outliers, unlike the average, which is more susceptible to them. As seen in Figure 2.1, each node utilized in the

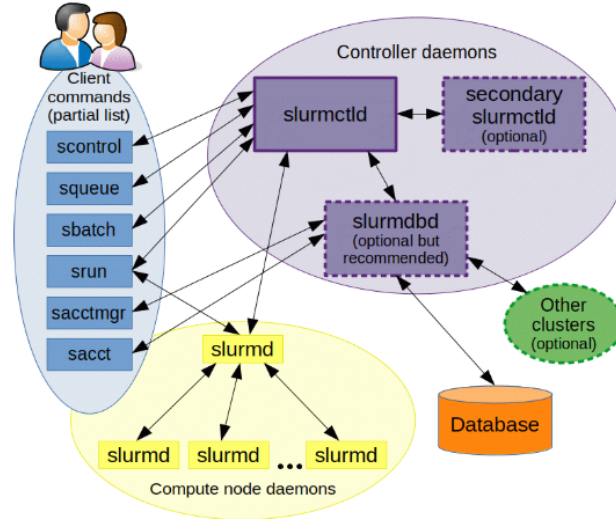
Node Type	Count	Accelerator	Memory	Cores
THIN nodes	426	w/o	64 GB	20@2.8 GHz (two sockets)
GPU nodes	44	dual tesla k40m	64 GB	20@2.6 GHz + 2 x K40
PHI nodes	18	dual xeon phi 7120p	64 GB	20@2.6 GHz + 2 x 7120p
FAT nodes	44	w/o	512 GB	40@2.4 GHz (four sockets)
ML node	1	8 volta v100	512 GB	40@2.2 GHz (two sockets)

**Figure 2.1.** *ARIS' nodes and their technical characteristics [4]*

experiments contains two sockets with 10 cores in each one of them. The 426 thin compute nodes (thin node island) have a theoretical peak performance (Rpeak) of 190,85 TFlops and a sustained performance (Rmax) of 179,73 TFlops.

### 2.1.2 SLURM Workload Manager

As mentioned before, ARIS uses SLURM Workload Manager [5] to manage and schedule the submitted jobs to the system. SLURM has the ability to allocate resources to a specific user for a specific period of time, offering exclusive or non-exclusive access. For the purpose of the experiments presented in this thesis, exclusive access to the allocated resources was used so as to correctly measure performance without interference from other submitted jobs. After a job has been submitted, SLURM monitors it until it either finishes its execution or its wallclock time (execution time estimation provided by the user) expires. At the same time, SLURM manages a queue of pending jobs that are ready to enter the system but await for resources to be freed by contending jobs.



**Figure 2.2.** *An overview of SLURM's components [5]*

SLURM's basic component is its slurmctld daemon, which is fault-tolerant, optionally has a backup daemon and is responsible for the operation of the workload manager. It stores the system's information, monitors each node's state, manages the system's partitions and schedules incoming



jobs to free resources. Each node also contains a `slurmd` daemon, which is responsible for initializing, monitoring and finalizing the jobs running on the node. Moreover, `slurmd` daemons communicate with the `slurmctld` daemon so that the latter can be aware of the running jobs' states and the availability of the nodes' resources. Finally the `slurmdbd` daemon manages a database of resources, limits and licenses and archives accounting records. Users and admins can use SLURM through a variety of command line commands, with some of them presented in Table 2.1.

Command	Explanation
<code>srun</code>	Submits a job for execution or initiates job steps in real time
<code>sbatch</code>	Submits a job script (that may contain multiple <code>srun</code> commands) for execution
<code>squeue</code>	Reports the state of all jobs currently in the system
<code>scancel</code>	Cancels the execution of a particular pending or running job

**Table 2.1.** *Four important SLURM commands*

### 2.1.3 Message Passing Interface (MPI)

Programming in distributed memory systems is less straightforward than in shared memory systems and requires a programming interface that supports message-passing between the different processes. MPI [35] is the most popular message-passing library interface specification used in most distributed memory systems. It supports programming in C, C++ and Fortran and has multiple implementations, including MPICH, Intel MPI, OpenMPI etc.

An MPI program typically starts with the **MPI\_Init** command, responsible of initializing the MPI environment. Afterwards, a common communicator, named **MPI\_COMM\_WORLD** for all processes is created, with each process being mapped to a particular rank (unique identifier). The programmer can create more communicators that contain particular processes or groups of processes if needed. After the MPI code has been written, the MPI environment is terminated using the **MPI\_Finalize** command. MPI contains a vast amount of operations, aiming to establish and enable data transfer or synchronization. Each operation consists of the following stages:

- **Initialization :** The MPI operation takes control of the argument list of the MPI command.
- **Starting :** The MPI operation takes control of the data buffers, which are the send/receive buffers that will be used for the data transfer. In this stage, the actual communication begins.
- **Completion :** Returns control of the content of the data buffers, after having completed the data transfer. At this point, the receive buffers have been updated with the data that has been sent.
- **Freeing :** Returns control of the rest of the argument list.

MPI offers both **collective** and **point-to-point** communication. In point-to-point communication,

a process communicates with another process through for example the basic commands `MPI_Send` and `MPI_Recv`. In collective communication, the communication can be One-to-Many (e.g. `MPI_Bcast`, `MPI_Scatter`), Many-to-One (e.g. `MPI_Gather`, `MPI_Reduce`) or Many-to-Many (e.g. `MPI_Alltoall`, `MPI_Allreduce`, `MPI_Allgather`). All the abovementioned commands are **blocking**, which means that they return only when the communication buffers can be re-written safely. MPI also offers **non-blocking** implementations of many of its commands (e.g. `MPI_Isend`, `MPI_Irecv`) that return immediately after being called but require from the programmer to check if the communication has been completed before re-using the communication buffers. Nevertheless, non-blocking operations are useful so as to overlap communication and computation, as well as to avoid deadlocks. Finally some synchronization directives also exist, that introduce waiting time, so as to coordinate processes efficiently. This however decreases the available parallelism of the application and has a negative effect on the application's scalability and speedup. The most important synchronization directives are the following ones:

- **MPI\_Barrier** : blocks all processes in the communicator until all of them reach the specific point in the code where the barrier is located. As it applies to all processes of a particular communicator, this directive may severely impact the available parallelism.
- **MPI\_Wait** : blocks the calling process until a specified communication operation completes. Its primary use case is to ensure that a non-blocking operation between 2 processes has finished.
- **MPI\_Waitall** : blocks the calling process until all specified communication operations have completed. Its primary use case is to ensure that a group of non-blocking operations have finished.

We must also keep in mind that many Many-to-Many communication directives may also include some synchronization overhead, as they must ensure that all data have been distributed to the processes that participate in the communication. Furthermore, blocking operations like `MPI_Send` or `MPI_Recv` may also include waiting periods if one of the two communicating processes is not ready to participate in the communication yet.

MPI programs can be compiled using different compiler wrappers. For the experiments conducted in this thesis, we used the Intel MPI compiler, otherwise known as `mpiicc` (for C programs) or `mpiifort` (for Fortran programs). Afterwards, the programs can be run using the `mpirun` or the `mpiexec` commands.

## 2.2 Benchmarks

In order to conduct the necessary experiments, we mainly used the MPI implementations of the **NAS Parallel Benchmarks (NPB)**, version 3.4.3, developed at NASA Ames Research Center. The NAS Parallel Benchmarks [14] are derived from different computational fluid dynamics (CFD) applications and have been widely used for many years now for various research purposes, including performance modeling and HPC simulations. Each benchmark can have a variety of problem sizes, which are referred to as classes. The structure of the name of each MPI benchmark is the following one:

$$\langle \text{benchmark\_name} \rangle . \langle \text{class} \rangle . \langle \text{number\_of\_processes} \rangle$$

So for example, mg.E.128 refers to the MG benchmark, with class E, running using 128 processes. The NPB benchmark suite consists of eight benchmarks and specifically five kernels and three pseudo-applications. The eight benchmarks are described below:

- **The Embarassingly Parallel Benchmark (EP)** : Collects two-dimensional statistics from a large number of generated Gaussian pseudo-random numbers.
- **The Multigrid Benchmark (MG)** : A simplified multigrid kernel that solves a 3-D Poisson partial differential equation.
- **The Conjugate Gradient Benchmark (CG)** : Calculates an approximation of the smallest eigenvalue of a large, sparse, symmetric positive definite matrix.
- **The 3-D FFT PDE Benchmark (FT)** : Solves a 3-D partial differential equation using Fast Fourier Transforms (FFTs).
- **The Integer Sort Benchmark (IS)** : Performs a sorting operation that is commonly used in “particle in cell” applications of physics.
- **The Lower-upper Diagonal Benchmark (LU)** : Uses a symmetric successive over-relaxation (SSOR) numerical scheme to solve a regular-sparse, block ( $5 \times 5$ ) lower and upper triangular system.
- **The Scalar Pentadiagonal Benchmark (SP)** : Multiple independent systems of non-diagonally dominant, scalar pentadiagonal equations are solved.
- **The Block Tridiagonal Benchmark (BT)** : Multiple independent systems of non-diagonally dominant, block tridiagonal equations with a  $5 \times 5$  block size are solved.

It is important to note that due to their structure, each benchmark cannot run with all possible process counts. In Table 2.2, the possible process counts for each benchmark are shown.

Benchmarks	Possible Process Counts
BT, SP	a square number of processes (1, 4, 9, ...)
LU	2D ( $n_1 * n_2$ ) process grid where $n_1/2 \leq n_2 \leq n_1$
CG, FT, IS, MG	a power-of-two number of processes (1, 2, 4, ...)
EP	No special requirement

**Table 2.2.** *NAS benchmarks process count requirements*

The NAS Parallel Benchmarks, apart from being popular in research circles, also represent a diverse variety of possible scientific applications, thus making them suitable for performance modeling research and simulations of real HPC systems. This is extremely important nowadays, when the heterogeneous and diversified HPC systems make it difficult to evaluate and model performance. In this context, having a trustworthy and diverse set of benchmarks can prove crucial both in performance modeling and to gain insights on how to improve the software and hardware design of large-scale systems [13].

In addition, some benchmarks from the **SPEChpc 2021** suite were utilized in this thesis. The benchmarks of this suite cover a wide range of scientific domains and as shown in [13], are representative of modern scientific HPC workloads. Just like the NAS benchmarks, each SPEC benchmark also comes in different workload sizes: tiny, small, medium and large. The structure of the name of each benchmark is the following one:

`<identifier>.<benchmark_name>.<workload_size>.<number_of_processes>`

The identifier is a number that is unique for this particular benchmark with a prefix indicating its workload size (5 for tiny, 6 for small, 7 for medium and 8 for large). So for example, 619.cvleaf\_s.1024 refers to the cvleaf benchmark, with a small workload size, running using 1024 processes. Figure 2.3 shows the SPEChpc 2021 benchmarks alongside some information about them.

Application Name	Benchmark				Language	Approximate LOC	Application Area
	Tiny	Small	Medium	Large			
LBM D2Q37	505.lbm_t	605.lbm_s	705.lbm_m	805.lbm_l	C	9000	Computational Fluid Dynamics
SOMA Offers Monte-Carlo Acceleration	513.soma_t	613.soma_s	Not included.		C	9500	Physics / Polymeric Systems
Tealeaf	518.tealeaf_t	618.tealeaf_s	718.tealeaf_m	818.tealeaf_l	C	5400	Physics / High Energy Physics
Cloverleaf	519.cloverleaf_t	619.cloverleaf_s	719.cloverleaf_m	819.cloverleaf_l	Fortran	12,500	Physics / High Energy Physics
Minisweep	521.minisweep_t	621.minisweep_s	Not included.		C	17,500	Nuclear Engineering - Radiation Transport
POT3D	528.pot3d_t	628.pot3d_s	728.pot3d_m	828.pot3d_l	Fortran	495,000 (Includes HDF5 library)	Solar Physics
SPH-EXA	532.sph_exa_t	632.sph_exa_s	Not included.		C++14	3400	Astrophysics and Cosmology
HPGMG-FV	534.hpgmgfv_t	634.hpgmgfv_s	734.hpgmgfv_m	834.hpgmgfv_l	C	16,700	Cosmology, Astrophysics, Combustion
miniWeather	535.weather_t	635.weather_s	735.weather_m	835.weather_l	Fortran	1100	Weather

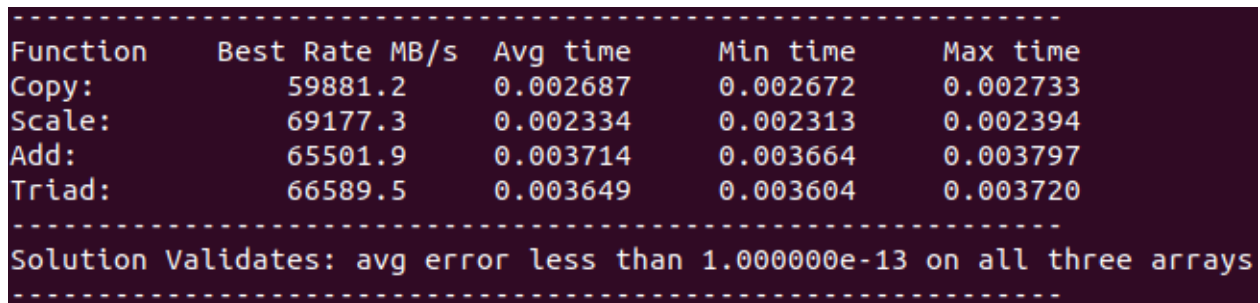
**Figure 2.3.** *All SPEChpc 2021 benchmarks alongside their workload sizes and names, their implementation language, their lines of code (LOC) and their scientific domains. [6]*

Finally, studying the memory bandwidth consumption of the applications in this thesis required

finding the upper bound of the memory bandwidth of each node of the ARIS supercomputer. To do so, the C version of the **STREAM benchmark** [15] was utilized. The STREAM benchmark measures the time it takes to execute four kernels (loops) that operate on large arrays that do not fit in the cache memory of a machine. After multiple trials, it calculates the best MB/s rate for each of these four kernels. The four kernels executed can be seen below:

- **COPY** :  $a[:] = b[:]$
- **SCALE** :  $a[:] = q * b[:]$
- **SUM** :  $a[:] = b[:] + c[:]$
- **TRIAD** :  $a[:] = b[:] + q * c[:]$

In Figure 2.4, the results of the STREAM benchmark for a random node of the ARIS supercomputer can be seen. The largest memory bandwidth is 69177.3 MB/s and is being achieved for the Scale kernel.



Function	Best Rate MB/s	Avg time	Min time	Max time
Copy:	59881.2	0.002687	0.002672	0.002733
Scale:	69177.3	0.002334	0.002313	0.002394
Add:	65501.9	0.003714	0.003664	0.003797
Triad:	66589.5	0.003649	0.003604	0.003720

Solution Validates: avg error less than 1.000000e-13 on all three arrays

**Figure 2.4.** Results of the *STREAM* benchmark in a node of the ARIS supercomputer

## 2.3 Profiling

### 2.3.1 Overview

Software Profiling is used to count the occurrences or frequencies of specific events (e.g. execution of a particular function, loop or line of code) or to measure system metrics (e.g. total executed instructions, cache misses etc.) during a program's execution. This technique is particularly useful for performance engineers, as it provides insights into a program's behavior and aids in code optimization and performance analysis or tuning. Through profiling, programmers can identify the so-called **hot spots** of their code (the most frequently executed code blocks). Profiling can be divided into categories according to when it's performed [36]:

- **Offline Profiling** : Traditionally, profiling is being performed offline, which means that its findings will be available only after one full, preparatory execution of the targeted program.

Since a significant proportion of scientific applications running in an HPC environment are executed repeatedly (e.g., weather forecasting applications that run each time a new forecast is required), this approach is generally not problematic. However, in many other use cases, the profiling results are needed much earlier than offline profilers can provide them.

- **Online Profiling :** This type of profiling leverages various techniques to gather runtime data up to a specific point in the program's execution, and then uses these insights to predict the behavior of the remaining execution. This approach is invaluable for dynamic compilation systems (just-in-time compilers), dynamic optimizers and binary translators, so as to guide them as to where their costly runtime operations must be implemented. Online profilers must balance the trade-off between completing their analysis quickly enough to allow time for implementing their predictions and being sufficiently accurate in identifying the program's hot spots.

A very common form of profiling is **path profiling**. A path can be defined as a series of sequential instructions in the control flow graph (CFG) of a program. Path profiling tries to specify which are the most commonly executed paths of a program (hot paths) to guide optimization efforts or aid in dynamic decision-making.

Profiling applications comes with many challenges. To begin with, the profiling procedures introduce time and space overheads to the original program execution. While this is of smaller importance for offline profilers, it is a crucial issue for online ones, since the user may notice a significant performance degradation. These overheads may stem from the methods employed by the profiler to collect the necessary information (which will be analyzed in detail later) and from many other program-specific characteristics. For example, in path profiling the potential paths in big applications may be of a significant amount, especially if there's a great deal of loops [37]. Additionally, they can also be large in size. Thus, profiling only a particular amount of paths and up to a specific depth in the callpath itself may prove to be necessary in order to avoid large overheads. Moreover, a program's path can be hot for the first half of the execution and cold for the second half. This constitutes a phase change and may mislead an online profiler and have an impact on its predictions.

Profilers utilize a variety of methods so as to collect information on an application's execution. The most popular out of these methods are presented below:

- **Instrumentation :** Through this method, the profiler modifies the source or binary code of an application with the intention to add specific commands in particular places in the code that will collect the necessary information during the execution. Instrumentation can be performed automatically by the compiler or by linking against pre-instrumented libraries [16]. Programmers can also manually instrument their codes by adding instrumentation

instructions where necessary. This profiling method is perhaps one of the most costly in additional overhead. For instance, if an application contains numerous code regions that individually take a small amount of time to execute but are executed very frequently, this can result in significant additional overhead. Consequently, the user may need to apply selective profiling and filter out some regions of code from the instrumentation procedure, according to various conditions so as to reduce the overhead. Furthermore, the profiling tools need to efficiently place instrumentation commands in the source or binary code so as to minimize overhead, like in [37].

- **Sampling :** With this method, the profiler collects profile samples continuously by making use of the operating system’s interrupt routines [17]. As a result, the final collected application profiles are statistical and not complete like in the previous method. Nevertheless, this method results in a much smaller overhead and can many times be sufficient or necessary in online profiling.
- **Hardware Performance Counters :** Hardware Performance Counters are special-purpose registers built within the CPU [18] [19]. They count specific hardware and software events and are directly updated by the CPU itself. As a result, they cause much less overhead than software-based profilers that require time-consuming system calls to gather their information. In addition, they don’t require additional hardware cost, as they are usually already present in the majority of modern processors. However, although there is a large number of diverse events that most processors can measure, the number of events that can be concurrently monitored by a processor is limited. Thus, the performance engineer must determine which performance events best suit the intended purpose. Apart from performance analysis, hardware performance counters are tremendously useful in a diverse range of research areas, like for example in computer security, where they’ve been used, to name some examples, so as to detect ransomware attacks in [19] or for integrity checking of applications in [18].

Although profiling can provide sufficient information to performance engineers, the information it extracts rarely incorporates temporal relations, which can be really informative in parallel applications that are characterized by synchronization and wait states. In these cases, **tracing** can be helpful, as it provides a detailed chronological record of the events that occur [16]. As expected though, it imposes a much bigger overhead to the program execution than regular profiling and produces a huge amount of data that require a lot of storage space. A detailed profiling use case that showcases many of the abovementioned profiling stages using three popular profiling tools can be found in Figure 2.5

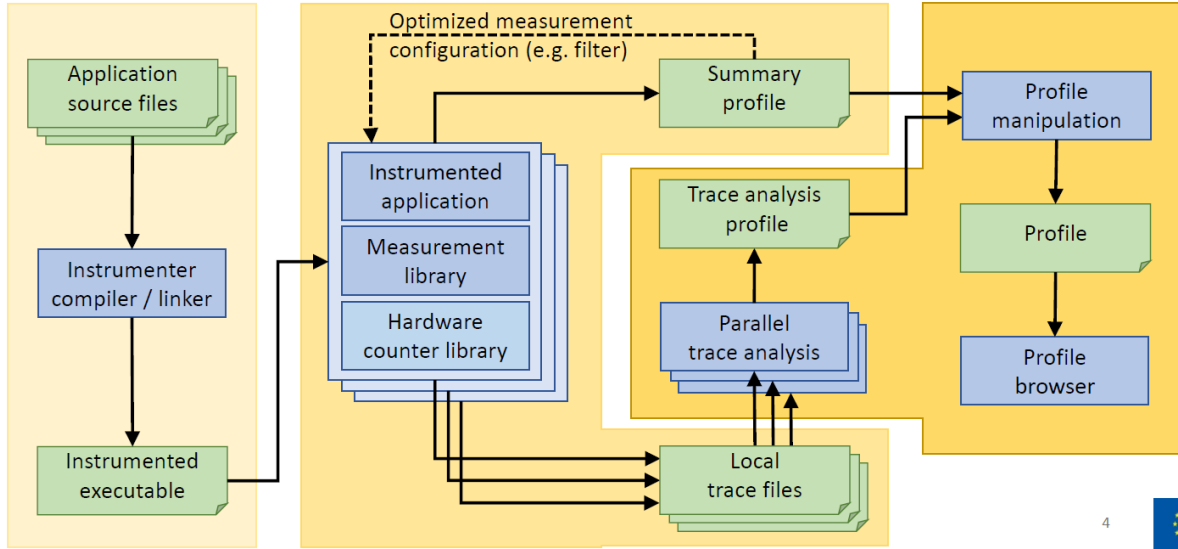
Even if the profiling overheads are kept low, there are cases where even the slightest overhead can be detrimental. Additionally, the way we instrument the code of an application depends on the programming model used in its development. As a result, sometimes passive system-level probing



### 1. Instrumentation

### 2. Measurement

### 3. Analysis



**Figure 2.5.** A diagram depicting a profiling workflow utilizing Score-P, Scalasca and CUBE. The application is first instrumented, then an effort to reduce the overheads by filtering the regions of code that will be profiled is made, and afterwards the profiling (and tracing) files are produced. Finally, the results are being analyzed and visualized in the User Interface. [7]

may be sufficient or necessary to avoid interfering with or altering the application in question. Through this method, the performance engineer monitors data on system usage during the runtime of the application without the need to instrument its code. Though this surely decreases the profiling overhead and complexity, it can only be implemented to gain insights on a particular application only if the system is relatively unoccupied. Nevertheless, this method can be employed on dedicated, controlled testbeds, set up by researchers with the aim of studying application performance like in [38].

In HPC application profiling there is one extra parameter that comes into play due to the distributed nature of execution: communication between MPI processes. Profiling communication may prove to be a challenging task, requiring new, dedicated tools. To begin with, communication in HPC environments occurs between various components of the HPC ecosystem. In MPI implementations, the MPI runtime interacts not only with the application itself, but also with the network fabric of the system and the job scheduler [39]. Being aware of the communication data of all three of these interactions is vital so as to identify bottlenecks and discover the causes for performance degradation so as to optimize an application. However, holistic communication profiling tools that evaluate the communication occurring during all these interactions are rare. Furthermore, tools that just interposition themselves between the application and the MPI library to collect information regarding the performed MPI calls may be less accurate than necessary in determining exactly the time when each message is sent, due to them not being aware of the way network operations will be executed [8]. For these reasons, depending on our purpose when studying application performance, we must be careful to choose the proper methodologies to profile communication.



In this thesis, the main purpose of the profiling performed is to provide general characterizations for our benchmarks. As a result, there is no immediate need for the extremely specific profiling that is required when developers try to pinpoint the causes of performance degradations so as to optimize their code. Consequently, we will focus on:

- **Performance Counters** measured by using the Linux **perf** tool. These counters will help us categorize our applications as compute or memory intensive.
- **Communication Statistics** measured by using the **mpiP** profiling tool. These statistics will help us categorize our applications as communication intensive and identify different communication behaviors so as to differentiate between communication intensive applications.

### 2.3.2 Profiling with perf

Perf [20] is a Linux command used for lightweight profiling and based on the `perf_events` interface. Perf can be utilized as an interface to the performance counters in Linux and to perform both static and dynamic tracing. The former is achieved by instrumenting a given code with tracepoints that collect information when triggered by the `perf` command, while the latter makes use of the `kprobes` (kernel space) and `uprobes` (user space) frameworks. When profiling performance counters with `perf`, there is no need for re-compilation or re-linking, as the `perf` command can just be placed in front of the executable and all measurements are taken at runtime. The most prominent command associated with `perf` is **`perf stat`**. This command is used to simply print to the standard output after the execution of a program the aggregated counts of occurrences of specified PMU (Performance Monitoring Unit) events. PMU hardware events, like cache misses, FLOPS or load instructions are mapped by the software to performance counters (which are physical registers as described previously) so as to monitor them. The `perf stat` command can also keep track of software events like context switches and all events can be measured at the user level, kernel level, or both, as well as at hypervisor level in the case of Virtual Machines. The `perf stat` command can be phrased in the two following ways:

```
perf stat -e cycles,instructions,cache-misses [executable details]
perf stat -e r1a8 [executable details]
```

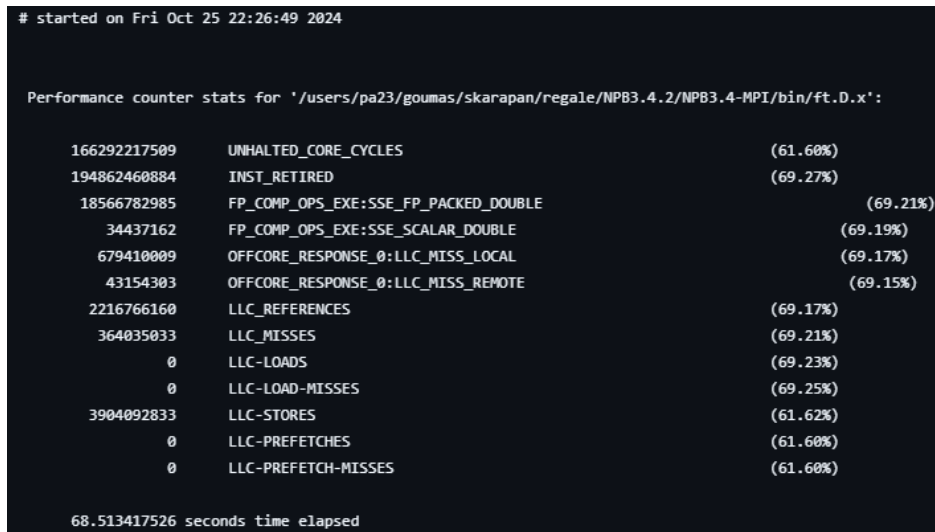
As seen above, the PMU events to be measured can be specified by the user using the `-e` option in two different ways:

- by using a set of predefined events provided by `perf` like "cycles", that are then automatically mapped onto actual events provided by the CPU
- by using hexadecimal codes that directly represent a PMU event. It's important to note that

the hexadecimal representations of PMU events vary between different CPU models and the user needs to check the architecture’s documentation to find the hexadecimal codes of the desired events.

Furthermore, starting with the Intel Nehalem microarchitecture, processors began supporting of-  
fcore events [21]. These events track memory interactions that occur outside the core, providing insights into cache misses, memory accesses, and data fetches from various levels of the memory hierarchy, in a multi-socket or multi-node system.

As mentioned previously, each CPU has a limited number of performance counters. These can be divided into generic counters that can measure every single PMU event, and fixed counters that can only measure specific events. As a result, if the user specifies in the `perf stat` command more hardware events than the available counters, the kernel will have to use multiplexing. This allows the kernel to switch events in and out of the hardware counters, meaning that not all events are measured continuously throughout the entire execution. At the end of the run, `perf` scales the event counts so as to provide an estimation of the total counts, had each event utilized the hardware counters for the entire application run. Another possible limitation is that since one file descriptor is used for each event, per thread, it’s possible to reach the maximum number of open file descriptors per process imposed by the kernel. An example of a `perf stat` output for a specific process of a benchmark is presented in Figure 2.6.



**Figure 2.6.** *Perf output for a specific process of an MPI benchmark*

Additionally, there exist some more `perf` commands that are widely used. The most prominent ones are briefly described in Table 2.3

By default, the `perf stat` command counts event occurrences for all threads of a specific process. There is also the option to count the event occurrences for each CPU. This constitutes a limitation for distributed memory applications whose execution takes place in many nodes. As a result, in our

Command	Explanation
perf record	Performs interrupt-based sampling to record events for later reporting and save them into a binary file
perf report	Reads the file produced by the perf record command and generates a concise execution profile
perf annotate	Annotates assembly or source code with event counts
perf top	Presents a live event count, similar to the Linux top tool

**Table 2.3.** *Four prominent perf commands*

experiments we added together the collected performance counters for each MPI process and kept the average value per node by dividing by the number of nodes. For each MPI process, we used the command `perf stat` along with specific events. Afterwards, multiple aggregations were performed so as to produce the abovementioned mean values of the events per node and some more complex metrics, like for example memory bandwidth of a single benchmark. Details on the events used and the metrics produced will be discussed later in this work.

### 2.3.3 Profiling with mpiP

mpiP is a lightweight MPI profiler for MPI applications [22]. It is used in order to collect statistical information about MPI calls, using the PMPI interface to view when a directive is initialized and when it completes [8]. It supports both link-time and run-time instrumentation capabilities, thus not demanding re-compilation of the profiled application. Like other previously mentioned profiling tools, it also enables profiling of only a specific user-defined section of the application. Figure 2.7 shows a fraction of the output of the profiling of a particular NAS benchmark using mpiP. The remaining output provides detailed information about each callsite, including the execution time and the sizes of the messages processed.

@--- Task Time Statistics (seconds) -----						
	AppTime	MPITime	MPI%	App Task	MPI Task	
Max	67.394348	32.828762		0	254	
Mean	67.380640	27.237899				
Min	67.310791	26.029214		1	132	
Stddev	0.023954	1.012329				
Aggregate	17249.443965	6972.902038	40.42			
@--- Aggregate Time (top twenty, descending, milliseconds) -----						
Call	Site	Time	App%	MPI%	Count	COV
Alltoall	3813	2.49e+04	0.14	0.36	25	0.00
Alltoall	3828	2.47e+04	0.14	0.35	25	0.00
Alltoall	3798	2.47e+04	0.14	0.35	25	0.00
Alltoall	3753	2.47e+04	0.14	0.35	25	0.00
Alltoall	3783	2.46e+04	0.14	0.35	25	0.00
Alltoall	3768	2.46e+04	0.14	0.35	25	0.00
Alltoall	3320	2.34e+04	0.14	0.34	25	0.00
Alltoall	835	2.32e+04	0.13	0.33	25	0.00
Alltoall	28	2.31e+04	0.13	0.33	25	0.00
Alltoall	2716	2.29e+04	0.13	0.33	25	0.00
Alltoall	13	2.29e+04	0.13	0.33	25	0.00
Alltoall	208	2.28e+04	0.13	0.33	25	0.00
Alltoall	73	2.26e+04	0.13	0.32	25	0.00
Alltoall	43	2.25e+04	0.13	0.32	25	0.00
Alltoall	3140	2.25e+04	0.13	0.32	25	0.00
Alltoall	88	2.25e+04	0.13	0.32	25	0.00
Alltoall	3439	2.24e+04	0.13	0.32	25	0.00
Alltoall	671	2.24e+04	0.13	0.32	25	0.00
Alltoall	103	2.24e+04	0.13	0.32	25	0.00
Alltoall	133	2.24e+04	0.13	0.32	25	0.00

**Figure 2.7.** *mpiP* output for a specific MPI benchmark. Visible are an overview of the time spent in the entire benchmark and the MPI time percentage as well as the top 20 most time consuming MPI directives alongside their callsites

---

## Co-Scheduling in HPC

---

The co-location of jobs on HPC nodes is a promising, emerging research sector. However, co-scheduling techniques have not yet found their way to production HPC systems. The deployment of a co-scheduling mechanism to traditional HPC systems is a rigorous task and as of now, even if its potential benefits have been highlighted in many research works, there exist no clear guidelines as to how co-scheduling should be applied in a real-world scenario. Additionally, when dealing with co-scheduling, implementing simple state-of-the-practice scheduling techniques may not be effective. We are in need of schemes that take into consideration the profile of the jobs in our application pool to make informed co-scheduling decisions. In this chapter, we will give a brief overview of scheduling and co-scheduling in HPC, focusing on key approaches to incorporate application-specific characteristics into decision-making processes.

### 3.1 Metrics and Application Types

Potential schedules or co-schedules in HPC and the algorithms utilized to produce them should be evaluated based on particular metrics. As shown in [23], the task of finding a proper set of evaluation metrics for this purpose is not straightforward. Furthermore, the necessity of categorizing applications to better label their behavior underscores the importance of identifying distinct application types.

In the realm of HPC, each metric does not hold the same importance for all stakeholders. The basic HPC stakeholders are the system administrators and the users that submit jobs. Obviously, administrators are interested more in the overall system performance and efficiency, contrary to the users who want their own submitted jobs to finish as early as possible. This antithesis can many times lead to multi-criteria optimization problems when designing a scheduling algorithm, as we

need to take both **system and user satisfaction** into consideration. The basic metrics we will use in this work are the following:

- **Makespan** : It refers to the total time needed to complete a set of jobs, starting from the submission of the first job to the completion of the final one. Since it is an indication of the throughput of the supercomputer, it is relevant only for system administrators.
- **Utilization** : It refers to the percentage of the system that is in use during an experiment. Nonetheless, as stated in [23] we must be careful to only measure the steady-state utilization of our system when we want to use this metric to evaluate an algorithm. Underutilization is bound to occur during the initialization and clean-up stages as the system gradually fills and empties afterwards. Additionally, HPC systems are not fully utilized throughout each day and time. When system utilization is low (e.g. during night time), this metric is once again not credible to evaluate potential scheduling algorithms. Utilization once again refers to the overall system performance and thus is only of interest for system administrators.
- **Turnaround time** : It refers to the sum of the waiting time and execution time of a particular job. As such, it is metric that interests the user that has submitted the job to the system.
- **Job Speedup/Slowdown** : In a co-scheduling scenario, when the two jobs are co-located, their execution times are bound to be different than when they are executed isolated (compact mode). In this context, the speedup of an application is given by the following formula:

$$\text{JobSpeedup} = \frac{\text{Compact Execution Time}}{\text{Co-located Execution Time}}$$

If JobSpeedup value is lower than 1, then we say that an application exhibits a slowdown when co-located with a particular neighbor. In a scheduling scenario, slowdown can also incorporate the waiting time of an application in the queue. For the purposes of co-scheduling, we will, for now, focus solely on the execution time change of an application to calculate its slowdown. Job Speedup/Slowdown is a metric that is of interest to the user.

In the case of metrics such as turnaround time and speedup/slowdown, there are also weighted versions that, in addition to the temporal aspect, also take into account the space an application occupies [23].

Furthermore, there exist three basic categories of MPI applications in an HPC context: **compute-bound applications** that mostly perform computations and utilize the CPUs of the nodes, **memory-bound applications** that mostly perform memory accesses and whose main bottleneck is the memory resources and **communication-bound applications** that spend most of their

time in MPI calls. In a shared-memory environment, compute-bound applications are scalable when more parallelism is added in the form of more CPU cores. On the contrary, memory-bound applications have limited scalability due to the finite memory resources. This is also proven in [24] for the case of EP (a heavily compute-bound benchmark) and MG (a heavily memory-bound benchmark) from the NAS suite. In a co-scheduling environment, an application typically spreads across twice the number of nodes compared to the isolated case, but occupies half the number of cores on each node (**spread mode**). Thus, the number of cores utilized for the application's execution stays the same as in the isolated case (**compact mode**), meaning that compute-bound applications are likely to experience neither a speedup nor a slowdown in these circumstances. Nevertheless, memory-bound applications can drastically improve their performance, as spreading to more nodes guarantees more memory resources and a higher memory bandwidth. Communication-bound applications can have a more unpredictable behavior when imposing the abovementioned co-scheduling scheme. This is because, on the one hand, more nodes provide additional memory buffers and network ports, but on the other hand, more nodes lead to increased communication overhead between them. As shown later on though, communication-bound applications also more often than not tend to have speedups in spread mode compared to their compact execution.

## 3.2 Traditional Scheduling Techniques

In order to showcase the need for co-scheduling, we will first provide a brief overview of the existing scheduling techniques and their shortcomings.

### 3.2.1 Scheduling and heuristics

Traditionally, resources are allocated to jobs using a batch scheduler, which is responsible for scheduling jobs to nodes in a fair manner [25]. It also manages access to different queues of a supercomputer, like for example a queue that contains nodes equipped with GPUs, or nodes specializing in Machine Learning. In order to make sophisticated scheduling decisions, the batch scheduler needs to have an estimation of the runtime of each submitted job at hand. For this reason, users are tasked to provide an estimation of the execution time of their submitted job, which is referred to as wallclock. As expected though, this wallclock is rarely accurate. Overestimation of the job's execution time by the user can lead to mistaken scheduling decisions by the scheduler and a longer wait time, while underestimation will lead the job to be prematurely killed and not concluded [40]. The former can have a negative effect on the utilization of the system, while the latter is detrimental for the energy efficiency of the system, as particular jobs will need to re-run. These problems have led many researchers to develop methods to predict the runtime [40] or queue time [25] of an application, so as to provide more insights to the scheduling algorithms than the ones given by the user estimation.

Given an estimation or prediction of the runtime of each job, the batch scheduler utilizes particular heuristic functions so as to schedule the jobs in the system. Traditional scheduling heuristics

include, but are not limited to [23] [26]:

- **First Come First Served (FCFS)** : Using this heuristic, the scheduler gives priority to the jobs that have arrived earlier to the waiting queue. This is perhaps the fairest heuristic towards the users and requires no knowledge of the application's execution time. However, it is bound to suffer from low utilization of the system's resources and will lead to a large idle time for particular nodes.
- **Shortest Job First (SJF)** : This technique prioritizes jobs with short runtimes over large ones. This requires knowledge of an estimate/prediction of each job's runtime and also means that the waiting queue is re-ordered and total fairness is disrupted. Additionally, starvation is possible. Another variant of this method is called **Shortest Area First (SAF)** and also takes into account the space a job occupies (number of cores/nodes) in addition to its wallclock, so as to determine the size of a job.
- **eXpansion Factor (XF)** : In this case, the waiting time of a job is also taken into consideration to minimize the danger of starvation. As a result, the priority of a certain job is given using the following formula :

$$\text{XFactor} = \frac{\text{Wait Time} + \text{Estimated Run Time}}{\text{Estimated Run Time}}$$

Moreover, recent approaches incorporate **machine learning techniques** in order to schedule the incoming jobs. A prominent example is given in [41] where the authors implemented a scheduler that made decisions using reinforcement learning techniques. However, as pointed out in [23], the main challenge in these scenarios is to find which metric(s) the scheduler must seek to optimize. Given the multi-criteria problems that often emerge when it comes to scheduling, such a task is cumbersome. Furthermore, the 'black-box' structure of these approaches hides the reasoning behind the decisions taken and may hinder the identification of potential injustices.

In any case, one of the most vital problems encountered when using traditional scheduling heuristics is the potential underutilization of the system. The state-of-the-practice method to address this limitation is through the use of **backfilling**.

### 3.2.2 Backfilling

With the use of traditional scheduling heuristics, the job with a greatest priority at a given time period may not fit in the system and may need to wait until other resources are freed so as to start its execution. This behavior will create gaps in the schedule, thus decreasing the overall system utilization. However, it may be possible for another job with a lower priority to fit into the gaps created by the stall of the job currently at the head of the queue, without causing further stalls to



the higher-priority job or disrupting the fairness of the system. This is exactly what the backfilling technique aims to accomplish. Figure 3.1 shows the basic backfilling principle. The two most prominent backfilling techniques are [26]:

- **Conservative Backfill** : A job that fits the created gaps is moved forward in the queue only if it does not delay **any** of the higher-priority jobs of the queue.
- **Aggressive (EASY) Backfill** : A job that fits the created gaps is moved forward in the queue only if it does not delay the job that is currently at **the head of the queue**. Contrary to conservative backfill, this technique may possibly disrupt fairness.



**Figure 3.1.** Visualization of backfilling [42]

Both techniques offer the system and the different types of submitted jobs both advantages and disadvantages [26]. EASY backfill has less restrictions than conservative backfill and thus has a higher chance of solving the issue of underutilization. It also enables long jobs to backfill under the right circumstances, which is increasingly difficult with the extra constraints imposed by conservative backfill. However, with EASY backfill, many jobs may see their turnaround time grow. In addition, the accuracy of the wallclocks provided by the users also impact the effectiveness of backfilling. As explained in [43], systematically overestimating the application runtimes can decrease the overall slowdown of the applications, compared to the case when user estimates are fully accurate. This is because this way bigger gaps are created in the schedule, thus increasing the backfilling opportunities. In a more realistic scenario though, where the application pool contains both accurate and overestimated user estimates, the overall slowdown seems to increase [26]. This behavior further highlights the importance of accurate user estimates.

### 3.2.3 Other techniques for handling underutilization

Although backfilling plays a vital role in improving system utilization, it does not completely eliminate the problem. As a result, more techniques are currently being used. The main adverse effect of underutilization in modern HPC systems is the fact that it wastes energy. One technique used to combat this problem is DVFS (Dynamic Voltage Frequency Scaling) [44]. Using this technique, the frequency of the cores is dynamically adapted according to the given load. This way, energy consumption is reduced if the system is idle. Nevertheless, this group of techniques are just methods to combat the adverse effects of underutilization rather than a means to increase

utilization. On the contrary, running multiple applications concurrently on a node (co-location) can both improve the system utilization and increase its throughput.

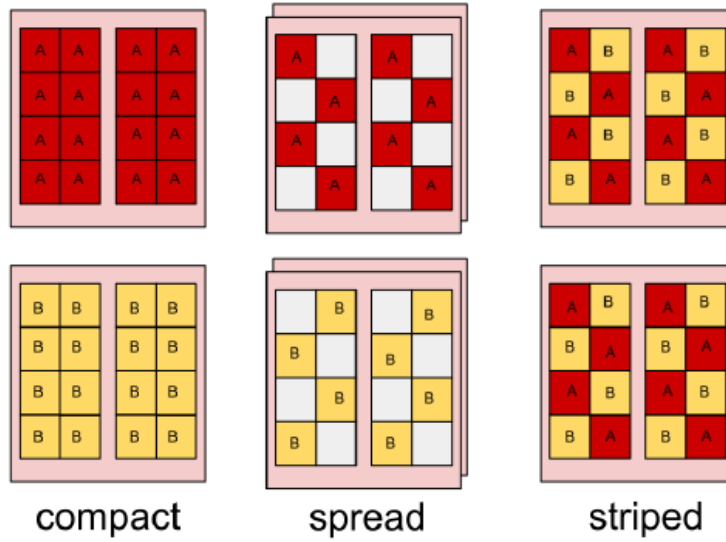
### 3.3 Co-Scheduling Overview

Executing a job using spread mode will, more often than not, lead to enhanced performance. In [1], the authors attribute this to the fact that processes of the same MPI job usually execute nearly identical tasks and thus try to simultaneously access the same resources. Reducing the number of processes of the same job in a specific node reduces contention for the said resources (e.g. caches, memory bandwidth, network ports) and our scheme can also benefit from the additional allocated resources to increase performance. However, simple spread mode entails that half the cores in each node stay idle, thus severely underutilizing the system. Moreover, executing a job in spread mode incurs higher costs for users, as in HPC environments, users are charged based on the core hours they consume. As a result, this mode of operation is particularly used in urgent computing tasks, where there is an immediate need for enhanced performance. For every day production use, the proposed co-scheduling scheme in research contains some sort of co-location of jobs on the same node. The most prominent and better performing scheme according to [1] is called **job striping**. Let's assume that each node in a supercomputer contains two sockets and each socket contains 8 cores (a total of 16 cores per node). Using the job striping scheme, half the cores of each socket (in our case four) will be assigned to the first job, while the other half will be assigned to the second job. Table 3.1 describes the aforementioned execution modes for an MPI job using 16 processes and running on the system described above. Afterwards, Figure 3.2 provides a visualization of the three execution modes.

Mode	Nodes Required	Core Utilization per Node
Compact Execution	1	16 cores (all cores used)
Spread Execution	2	8 cores (8 cores idle)
Job Striping	2	8 cores (remaining used by another job)

**Table 3.1.** *Comparison of Execution Modes*

Job striping is not the only viable co-location approach. Another approach is to assign an entire socket to each job in a node (**socket-exclusive**). According to the system configuration any one of these two approaches may be suitable. In [1] and [45] job striping yielded better results than socket-exclusive, while the opposite was true in [46]. The primary advantage of job striping is that by allocating heterogeneous processes to the same socket, we reduce the contention in the Last-Level Cache (LLC) of that particular socket in comparison to the socket-exclusive approach where multiple similar processes make similar demands on the LLC. However, processes from the first socket will need to communicate with processes of the same job from the second socket, thus creating a communication overhead, which is avoided in the socket-exclusive approach. Consequently, there exists a trade-off between LLC and memory bandwidth contention and hardware locality and the

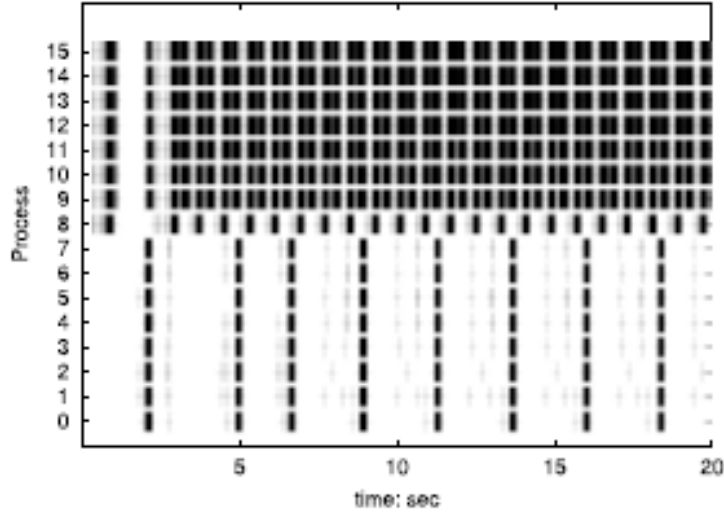


**Figure 3.2.** Visualization of the three mentioned execution modes [1]

suitable approach depends on the system in use. For the experiments conducted in this work, the job striping approach was utilized.

As mentioned before, the impact spread execution or co-execution have on MPI communication may be unpredictable. On the one hand, more memory buffers and network ports certainly accelerate communication. The time spent in MPI communication is also not necessarily useful communication as it may also include waiting and synchronization periods, as explained in Section 2.1.3. On the other hand though, spreading a job to more nodes increases the needed communication. Communication is also dependent on link contention in the entire system, something that is independent from the chosen execution mode. Nevertheless, most results up to now seem to indicate that communication-bound applications benefit from a co-execution scheme. This is corroborated in [8], where it is shown that when two applications have different communication patterns, they can both benefit from co-location. Since there is a high chance that two distinct jobs have different communication patterns, there is certainly a chance to benefit from co-location for communication-bound processes. An example of the communication patterns of two NAS benchmarks in co-execution mode is shown in Figure 3.3.

Co-Scheduling has been shown to result in an overall improvement of the makespan of an application pool and in speedups for the majority of the application types involved. However, a specific methodology or algorithm to find the optimal co-schedule given a specific application pool is still an open problem. In [27], the authors proved that optimally co-scheduling jobs in a system with more than two cores per chip is an NP-complete problem. Other works have focused on approximating the optimal co-schedule asymptotically. In any case, some jobs are bound to be slowed down, which gives birth to a question of fairness. Firstly, this leads to the abovementioned multi-criteria optimization problem, where we need to simultaneously try and keep the system and the user satisfied



**Figure 3.3.** *Communication patterns of two different NAS benchmarks when co-located in a 16-core node [8]*

as far as overall and per job performance are concerned. Secondly, as mentioned above, in most HPC systems users pay according to the core hours they consume. If a particular submitted job is slowed down due to its co-location, the user will consume more core hours and thus pay a higher cost. Since co-scheduling is yet to be implemented in a production system, this accounting issue is yet to be resolved in a real-world scenario. In this context, backfilling is not as straightforward as in traditional scheduling. Specifically, if a job is backfilled next to a neighbor that causes it to experience a slowdown it might actually finish later than if it just entered the nodes without being backfilled. Nonetheless, to develop a sophisticated decision algorithm for co-scheduling jobs that accounts for both system performance and user satisfaction, rather than relying on random co-scheduling, it is essential to understand the specific characteristics of the applications in our application pool. The next section presents some of the available approaches for extracting and processing knowledge about our applications, which can be used to make informed co-scheduling decisions.

### 3.4 Co-Scheduling Approaches

A plethora of co-scheduling approaches have been proposed. The simplest way to perform co-scheduling is by co-locating jobs as they come in a First Come First Served (FCFS) manner. This technique can be further enhanced using some sort of backfilling in order to decrease underutilization. However, with co-scheduling, contrary to simple scheduling, the placement of a job in the system will impact its runtime. The aforementioned method, even if it can possibly result in makespan improvements, does not make any effort to either further improve job speedups or decrease job slowdowns. Thus, sophisticated co-scheduling methods have been proposed. These methods can lead to improved system and user satisfaction but also require a varying degree of knowledge on the characteristics of the jobs in the waiting queue. We divide these methods into

two broad categories: **tag-based** models and **pairwise** models.

### 3.4.1 Tag-based models

Tag-based models aim to retrieve a particular standalone characterization of all applications in the application pool and afterwards make informed co-scheduling decisions taking each application's profile into consideration. This characterization can either be **resource-centric** or **co-location-centric**. In a resource-centric approach, we characterize applications according to their resource consumption patterns. This may range from a simple label (e.g. memory-bound, compute-bound or communication-bound) to a detailed analysis of the application's resource consumption to obtain a detailed profile. Figure 3.4 presents a snapshot from the HPC Performance Characterization analysis provided by the profiling tool Intel VTune. Having obtained a profile for each application in our application pool, we can then make either statically or dynamically informed decisions about where each application must be placed in our system.

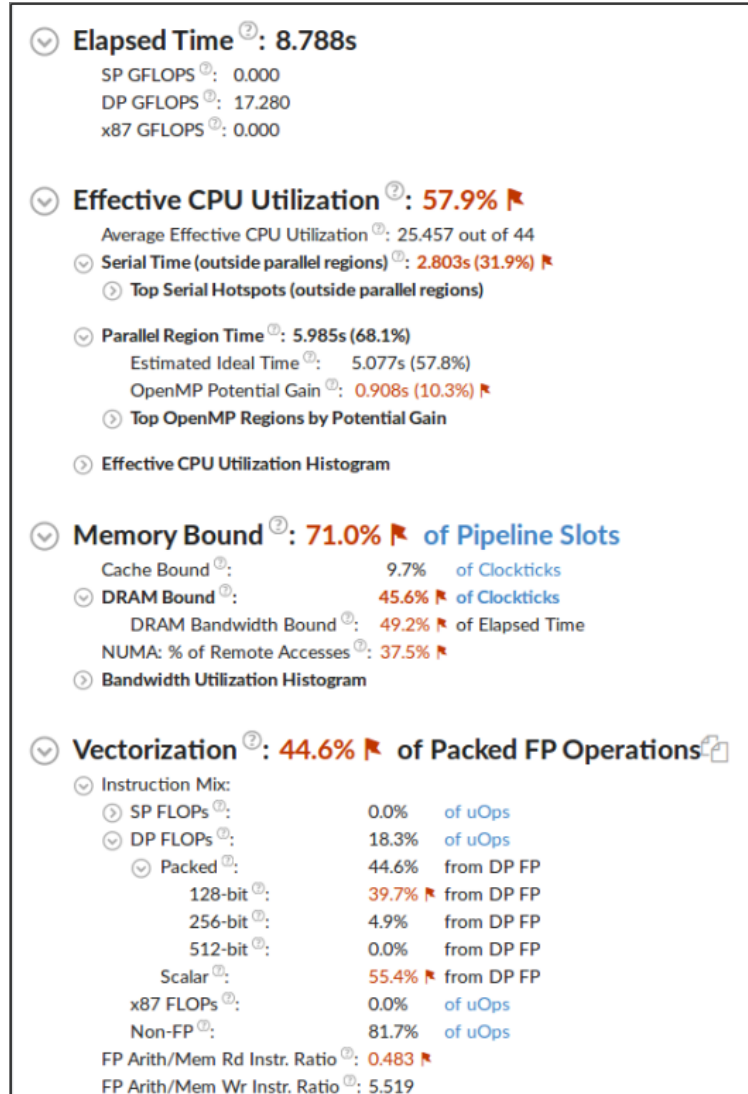
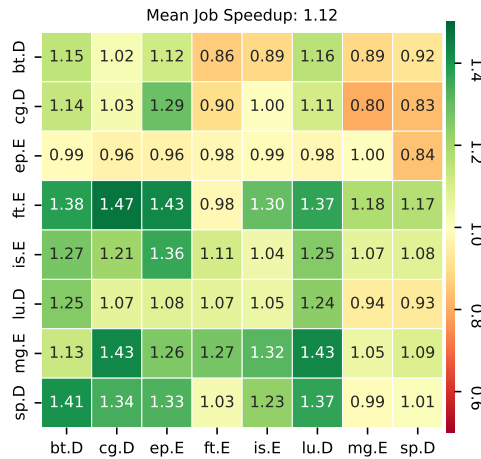


Figure 3.4. HPC performance characterization using Intel VTune [9]

In a co-location-centric approach, each application must be labeled as either co-location friendly or co-location unfriendly. This approach allows us to either utilize this information to determine which pairs of applications will be co-located or to select a subset of applications for a co-scheduling dedicated queue, with the remaining applications assigned to the regular scheduling queue. The authors of [1] proposed that the performance of an application when executed in spread mode can provide us with insights about its behavior in a co-scheduling environment. Specifically, they argue that applications that gained a higher than 40% performance boost when executed in spread mode are also, more often than not, expected to benefit from co-location. They then generalize this conclusion by noting that when an application that significantly benefits from spread execution is co-executed with another application that gains less from it, the former application experiences improved performance. By dividing our applications into High, Medium and Low spread performers we can then decide the optimal way to co-schedule them. This approach requires two runs for each application in order to gain the necessary insights. Specifically, each application needs to be executed in both spread mode and compact mode to measure its speedup when run in spread mode. Resource-centric approaches require one run for each application in compact mode, while simultaneously profiling it so as to determine its resource consumption patterns. This constitutes a simple pre-processing step in most HPC cases, since the majority of HPC applications are run multiple times by users (e.g. weather prediction models) and thus profiling only their first run adds minimal overhead. Co-location-centric approaches demand a more sophisticated way to characterize the applications.

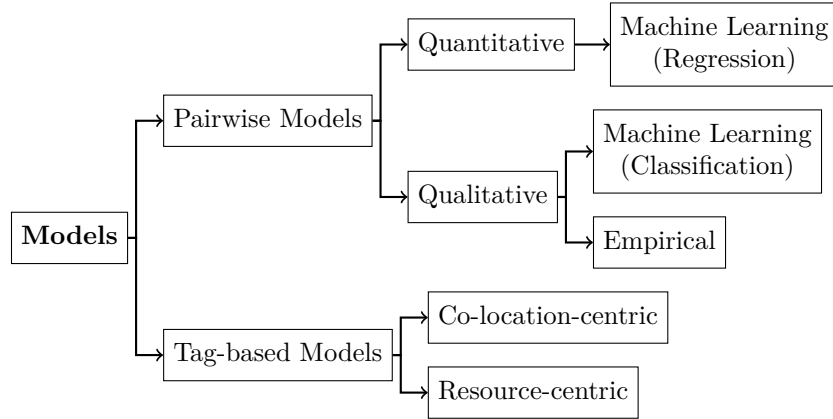
### 3.4.2 Pairwise models

Pairwise models necessitate knowledge or estimation of the co-scheduling behavior for all potential pairings of applications within the application pool. Heatmaps, such as the one shown in Figure 3.5, offer a visualization of these data, where each row represents the speedups of the corresponding application when co-located with the applications represented in each column.



**Figure 3.5.** Heatmap of speedups from the co-execution of eight NPB benchmarks of 256 processes each in the Marconi supercomputer

The main idea here is that the heatmap of the applications of our application pool is used by an optimization algorithm to make the co-scheduling decisions. The heatmap can be produced either by exhaustively running all possible pairings and measuring their performance compared to the compact mode execution of each application (an impractical method in a production environment), or by predicting the heatmap qualitatively or quantitatively. A **qualitative** approach could use labels (e.g. good, bad, stationary) to characterize the behavior of each application when paired with each one of the rest of the applications. These labels can be produced either **empirically** or by using **Machine Learning classification algorithms**. Such methods will be proposed later on in Chapter 4. On the other hand, a **quantitative** approach aims to predict the exact numerical value of the speedups in the heatmap. This can be accomplished by using **Machine Learning regression algorithms**. It is important to note that apart from the necessary sophistication so as to conceive the aforementioned prediction models, pairwise models also require pre-existing data (performance counters and heatmaps of applications), so as to either make empirical rules, or train the Machine Learning models. In the next chapter, we will propose and explore some methods to gain insights into the performance of applications in co-execution mode so as to produce the necessary data that will be used by sophisticated co-scheduling algorithms to make informed co-scheduling decisions. We also note, that our experiments will be conducted using benchmarks that represent popular HPC scientific applications (e.g. NAS and SPEC benchmarks). Future research around co-scheduling will also need to evaluate it using modern Big Data or Machine Learning applications like in [45]. The taxonomy described above is also briefly illustrated in Figure 3.6.



**Figure 3.6.** *A brief taxonomy of application models for co-scheduling*

---

## Performance Prediction in co-execution mode

---

In this section we will present some performance analysis methods that can be used to gain valuable insights regarding the behavior of applications in a co-scheduling scenario. The knowledge extracted from these methods can be utilized by co-schedulers to make informed co-scheduling decisions. The common denominator of all presented models is their usage of performance counters. For this reason, we will firstly explain the methodology used to obtain these counters for all applications with the use of profiling. Afterwards, we will use these profiles in the form of a tag-based model and provide an algorithm that leverages them to improve the results of random co-location. We will then discuss some qualitative approaches in predicting the heatmap of a specific application set: one approach relies on empirical rules derived from the manual analysis of previous co-scheduling results, while the other approaches employ traditional Machine Learning algorithms trained on the same co-scheduling data. In the end, we will also evaluate a way to use combinations of the empirical and machine learning approaches to enhance precision.

### 4.1 Profiling Methodology

As mentioned before, we profiled the benchmarks used in our experiments using `perf` and `mpiP`. Using `perf`, we aimed to extract information regarding the computation and memory characteristics of an application's runtime. With `mpiP`, we tried to gather insights on its communication behavior. The computational aspect of an application will be represented in this work by the **FLOPS** metric, which measures the amount of floating points operations occurring per second. This metric is widely used as a measure of computation for scientific applications, due to the high-precision nature of these workloads that render integer operation more rare. We calculate this metric by dividing the total number of floating point operations during each application's runtime by its total execution time. In order to calculate the total floating point operations taking place during an application's



runtime, we utilize two perf events : **FP\_COMP\_OPS\_EXE:SSE\_FP\_PACKED\_DOUBLE** and **FP\_COMP\_OPS\_EXE:SSE\_SCALAR\_DOUBLE**. These events measure two types of SSE (Streaming SIMD Extensions) instructions: **packed** and **scalar** double-precision floating-point operations. Packed SSE instructions divide their operands into multiple floating-point segments that can be processed in parallel, while scalar SSE instructions execute a single operation, typically on data located in the lower bits of their register arguments [47]. SSE is a SIMD instruction set extension to the x86 architecture, designed by Intel. We are interested in the per-node value of the FLOPS metric and as a result we profile each MPI process using perf, we add the floating point instructions performed throughout all processes using the sum of the two above perf events and then divide the result by the total execution time of the application and the number of nodes the processes run on. The above methodology is depicted in the following formula:

$$\text{FLOPS} = \frac{\sum_{i=1}^{np} (\text{SSE\_FP\_PACKED\_DOUBLE}_i + \text{SSE\_SCALAR\_DOUBLE}_i)}{\text{Number of Nodes} \cdot \text{Execution Time}}$$

where np stands for the number of processes of the MPI job.

Moreover, we want to gain insights into the memory accesses of an application. To accomplish this, we will calculate an indication of the **memory bandwidth** of each application by utilizing two perf offcore events: **OFFCORE\_RESPONSE\_0:LLC\_MISS\_LOCAL** and **OFFCORE\_RESPONSE\_0:LLC\_MISS\_REMOTE**. Both offcore events track the number of last-level cache (LLC) misses, which are typically resolved by the main memory system and thus provide an indication of main memory accesses. The former event counts the LLC misses that were handled locally (i.e., by the core's own memory controller), while the latter counts the cases where the data must be fetched from a remote core or node (in a multi-socket or multi-node system). Afterwards, we multiply the sum of these two events by the cache line size (64 bytes in the ARIS system) to see how many bytes were transferred. The cache line size defines the amount of data that can be retrieved from the main memory in a single operation. Making use of the same methodology as for the FLOPS metric, the per-node memory bandwidth estimation is given by the following formula:

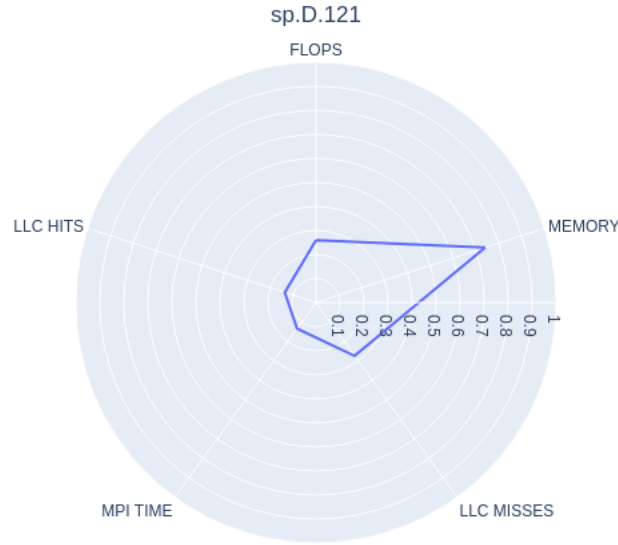
$$\text{Memory Bandwidth} = \frac{\sum_{i=1}^{np} ((\text{LLC\_MISS\_LOCAL}_i + \text{LLC\_MISS\_REMOTE}_i) \times \text{Cache Line Size})}{\text{Number of Nodes} \cdot \text{Execution Time}}$$

LLC misses within the core can be determined using the perf event **LLC\_MISSES** when necessary, while LLC hits can be calculated by subtracting the LLC misses from the perf event **LLC\_REFERENCES**. The values of both events are then divided by the number of nodes the application utilized and its execution time, just like previously.

At the same time, the communication behavior of each application was profiled using mpiP. First of all, from the **Task Time Statistics** section of the generated mpiP report, we extract the aggregate MPI% value, which represents the proportion of the application's runtime that was spent on MPI operations. This value will indicate the degree to which the application is communication-bound.

Additionally, utilizing the **Callsite Time statistics** section of the report, we will calculate the percentage of MPI time consumed by each specific type of MPI directive (e.g. MPI\_Wait, MPI\_Send etc.). This will help us determine the percentage of productive communication versus idle waiting times.

FLOPS, Memory Bandwidth, MPI Time, LLC Hits and LLC Misses that characterize a given application were then grouped together and visualized using **spider plots**. Figure 4.1 shows an example of a spider plot for the SP benchmark running on 121 processes.



**Figure 4.1.** Spider plot for the SP benchmark

## 4.2 A Tag-Based Model

Tag-based models aim to enhance the generated co-schedules by relying solely on a specific characterization of each application within the application pool. This characterization can vary from a basic tag provided by profiling tools such as Intel VTune to a comprehensive set of related performance counters. Our approach involved gathering the previously mentioned performance counters related to memory and computation, along with communication-related data, and utilizing them as application tags to guide decision-making in co-location scenarios.

We also conceived a strategy to utilize the created tag of the applications so as to improve the mean job speedup in a co-location scenario. We have already discussed in Section 3.1, that executing memory-bound applications in spread mode can be beneficial for them due to the higher number of memory resources. At the same time, compute-bound applications are likely to exhibit a stationary behavior when executed in spread mode. As proven in [28], one can reduce the average slowdown experienced by co-located applications by simply preventing instances of the same program from

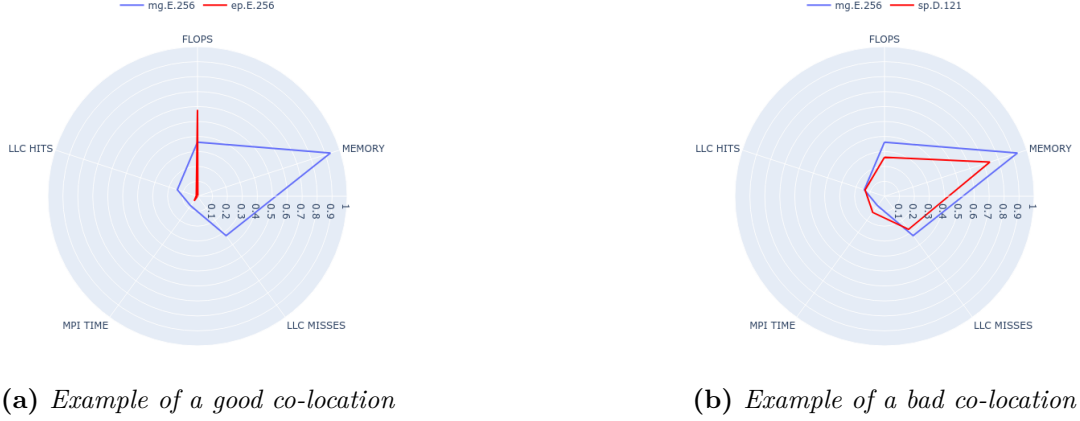
being co-located together. This is intuitively expected, as applications that are memory intensive will hinder each other's performance if forced to share the same memory resources. This way, their vast potential to have large speedups due to the bigger amount of memory resources will not be exploited. On the other hand, co-locating two compute-intensive applications may prevent either from experiencing a slowdown. However, as will be shown later on, compute-intensive applications usually facilitate the speedup of their neighbors and as a result co-locating them together eliminates the potential benefits that could be gained by reserving them for other applications. The authors then go on to prove their hypothesis, using both a statistical theory, as well as simulated static co-location experiments. However, the frequency with which we encounter use cases involving multiple instances of the same program running simultaneously in a real-world HPC system is a matter of debate. This may occur in ensemble simulations, parameter sweeps, or multi-replica runs, where the same application is executed with different input parameters or configurations. Consequently, we extend and generalize this reasoning by examining a co-location scenario where the goal is to create pairs of applications whose spider plots are as different as possible.

We explore the simple static scenario, where we have an application pool of 300 applications and we want to create 150 pairs. We conducted 100 experiments, each involving a randomly selected mix of 300 applications in the application pool. In each experiment, we compared the performance of a baseline approach against a more sophisticated method for the given application mix. In the simple baseline case, applications are paired sequentially, with the first paired with the second, the third with the fourth, and so forth. Afterwards we implement a sophisticated case, where we exploit the abovementioned reasoning. Specifically, for each potential application pair in the pool, we calculate a score that indicates the degree of overlap between the spider plots of the two applications. This score is calculated as follows:

$$\begin{aligned}\text{FLOPSdiff} &= |\text{FLOPS}_1 - \text{FLOPS}_2| \\ \text{MEMORYdiff} &= |\text{Memory}_1 - \text{Memory}_2| \\ \text{COMMdiff} &= |\text{MPI\_time}_1 - \text{MPI\_time}_2| \\ \text{score} &= \sqrt{(\text{FLOPSdiff})^2 + (\text{MEMORYdiff})^2 + (\text{COMMdiff})^2}\end{aligned}$$

The bigger this score is for a particular pair, the more different from each other these applications are, thus indicating a better co-scheduling outcome. Figure 4.2a presents two applications with minimal overlap in their spider plots. This pair of applications is thus considered to be a good co-location, and indeed under this co-location, mg.E.256 achieves a speedup of 1.71 which is impressive for co-execution mode, while ep.E.256 experiences a speedup of 0.96 (actually a slight slowdown of 4%), indicating a stationary behavior. This outcome is expected, as the EP benchmark typically neither benefits nor is adversely affected from co-execution, as we will demonstrate later. In contrast, Figure 4.2b illustrates two applications with significant overlap in their spider plots. This pair of applications is thus considered to be a bad co-location, which is corroborated by the

fact that mg.E.256 achieves a speedup of 1.01 and sp.D.121 achieves a speedup of 1.03 under this pairing, both actually being fully stationary. Given the potential of MG for significant speedups in co-execution mode, as demonstrated in the previous example (a behavior also observed with SP), this pairing is justifiably regarded as a poor co-location choice.



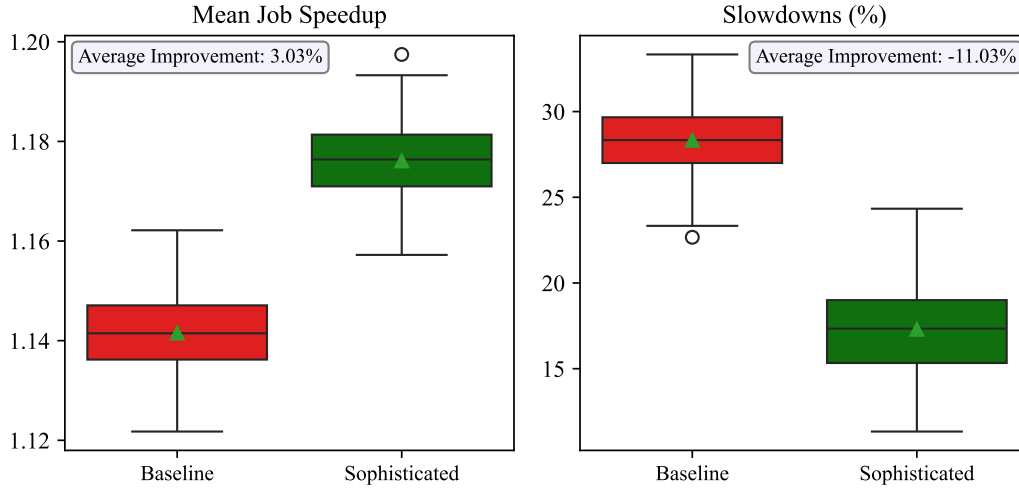
**Figure 4.2.** Spider Co-Plots for a good and a bad co-location scenario

Our approach will try to co-locate applications in a greedy way. Specifically, as we move sequentially through our application pool, we will co-locate each application with the one that results in the highest possible score. This approach heavily favors the first few applications of the application pool that will be paired with optimal neighbors. Nonetheless, it results in a considerable improvement for the mean job speedup and a significant decrease for the percentage of applications that experience a slowdown when co-located. Figure 4.3 demonstrates the benefits of our approach, compared to the baseline case described before. We observe an improvement of the average mean job speedup by 3.03% and a decrease of the average slowdowns percentage by 11.03%. We define the average slowdowns percentage as the percentage of the applications in each experiment that experienced a slowdown (i.e. their speedup value was below 1). These statistical results are encouraging, as intuition suggests that a higher mean job speedup in this static scenario could translate to a shorter makespan in a dynamic setting. However it is important to note that this statistical experimentation makes the following simplifying assumptions:

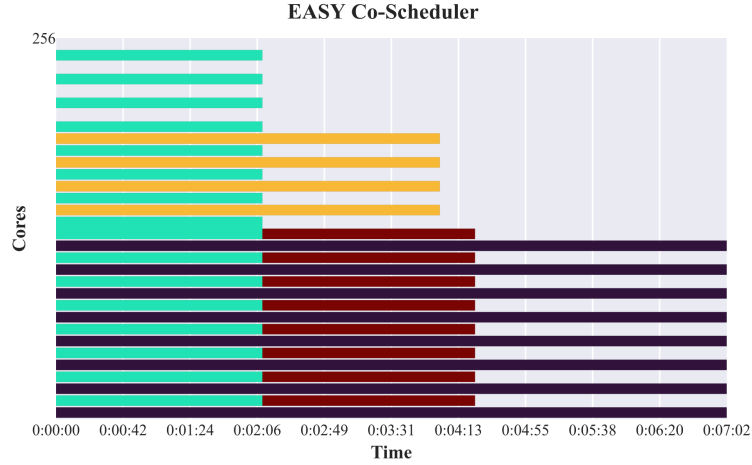
- All applications are assumed to have the exact same execution time.
- Each application is paired with exactly one other application for its entire runtime.

These assumptions are usually not valid in real HPC systems, as applications can have significant variations in execution times, meaning that each process of larger applications may be co-located with more than one other application during its runtime. Moreover, due to the diversity of processes each application runs on, some processes of an application may be co-located with one application, while others are placed with a different one. Additionally, there is a possibility that some processes of an application may run alone on their node due to utilization constraints, where the next

application cannot be accommodated in the remaining cores. These scenarios that can arise in real-world HPC systems are depicted in Figure 4.4.



**Figure 4.3.** Benefits of our tag-based approach in comparison with the baseline case



**Figure 4.4.** Potential application placings in a real-world HPC scenario

In particular:

- The light green application runs on 128 processes and thus some of its processes are co-located with the dark blue application, some others with the yellow application and some run alone on their nodes because the dark red application did not fit in the created gap.
- The dark blue application has more than one neighbors throughout its runtime (the light green and the dark red applications), due to its large execution time. It is also practically executed in spread mode after the dark red application has finished running, as there is no other application left in the waiting queue.

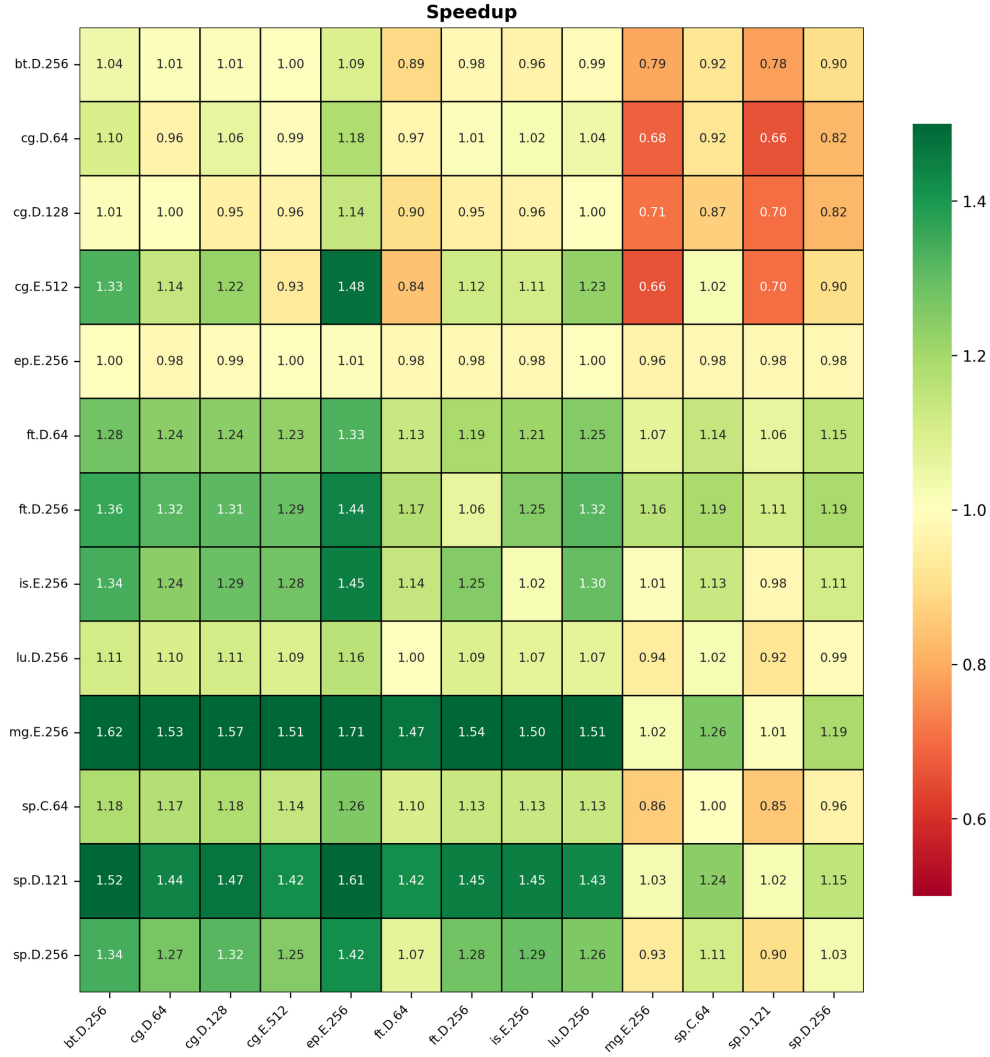
Consequently, a plain co-location scenario where we are tasked to just make pairs out of an applica-

tion pool is highly unlikely in a real-world setting. Nonetheless, our aforementioned approach can be utilized to make real-time decisions. Specifically, when an application is ready to be placed in the system’s nodes and multiple potential spots are available, we can choose among the candidate options, the cores that contain the application that will maximize the aforementioned score. An added benefit of our approach is that it takes both applications into consideration when making decisions, thus trying to avoid or minimize cases where even if one application heavily benefits from a co-location, the other application is heavily harmed by it. More sophisticated variants of our method may be possible in an on-line scenario, where algorithms can make use of waiting queue re-orders to further optimize the co-location decisions.

### 4.3 Empirical Heatmap Prediction

We then devised an empirical qualitative model that aims to predict the heatmap created by a particular application pool. The model consists of a set of rules that were then used to make an empirical decision tree that receives as input two applications A and B and predicts application A’s speedup if paired with application B. The three labels the model uses to characterize the speedup of application A when paired with application B are **good**, **stationary** and **bad**. In order to extract the rules used to construct the decision tree, we analyzed a heatmap from the ARIS supercomputer, which consisted of all NAS benchmarks, as well as a heatmap from the ARIS supercomputer that consisted of some of the SPEC<sub>hpc</sub> 2021 benchmarks. The two heatmaps are depicted in Figures 4.5 and 4.6. In order to calculate the speedup value for each co-execution pairing, we placed each pair on the same node and replicated the pairing across the required number of nodes. The pair was co-executed for 10 minutes, with any completed job being restarted during this period. The median execution times from these repeated runs were used as the co-execution times for the benchmark. The reported speedups were calculated as the ratio of the original execution time (under compact allocation) to the co-execution time, just like we mentioned earlier in Section 3.1. It is important to note here that most of the pairings in these heatmaps exhibit speedups or mild slowdowns, giving a first indication that co-scheduling can be beneficial as a strategy for a plethora of scientific applications. Moreover, the fact that NAS and SPEC benchmarks, which are representative of many types of scientific applications, form the foundation for our empirical model suggests its applicability for general scientific workloads.

We once again used the performance profiles created with the help of `perf` and `mpiP` to generate spider plots for all benchmarks, allowing us to categorize them effectively. Among the three main metrics—FLOPS, MPI Time, and Memory—the one with the highest value determined whether the benchmark was compute-bound, communication-bound, or memory-bound, respectively. The upper bound for the Memory axis was calculated using the STREAM benchmark, as explained in Section 2.2, while for the FLOPS axis we used a value larger than all the used benchmarks’ respective values. The MPI Time axis ranges from 0 to 1 as each MPI Time value is the percentage of the execution time spent in MPI calls. Finally, the spider plots also depict Last-Level Cache hits and misses, as they will be useful in some cases as it will be seen later on. Their upper bounds

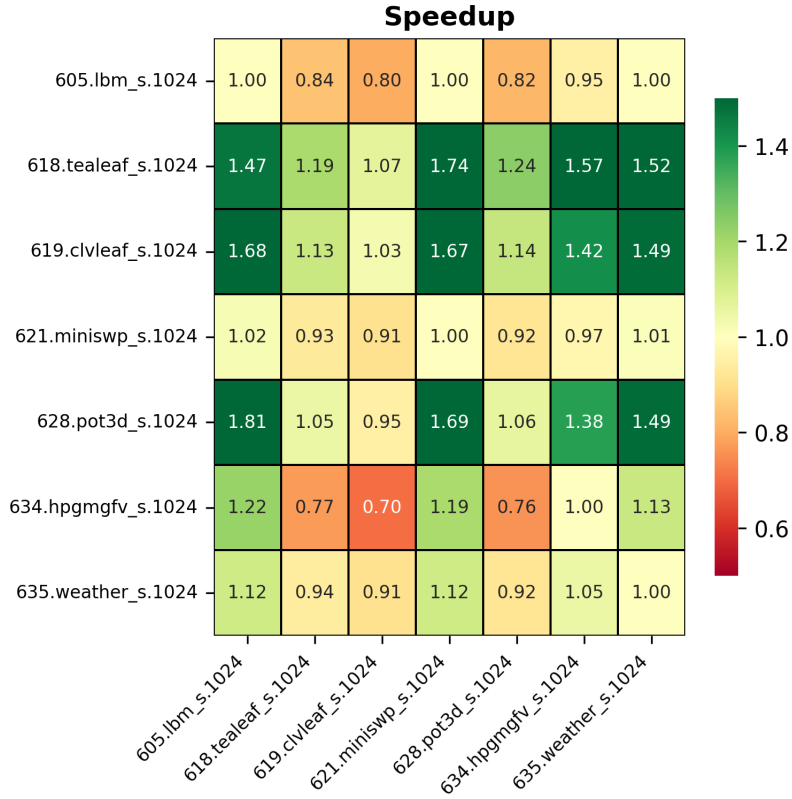


**Figure 4.5.** Heatmap of the used NAS benchmarks in a co-execution environment (Average Speedup : 1.12)

coincide with the greatest observed value of these metrics among the used benchmarks. The spider plots for a single case of processes from each of the 8 NAS benchmarks used are shown in Figure 4.7.

### 4.3.1 Compute-bound applications

We first of all turn our attention to the EP benchmark. As shown in Figure 4.7c, EP is a heavily compute-bound program with only minimal memory accesses and MPI communication. Observing the heatmap at Figure 4.5, we notice that the EP benchmark remains almost fully stationary when co-executed with all other benchmarks (its speedups range from 0.96 to 1.01). As a result, it is safe to assume that heavily compute-bound applications have the tendency to stay stationary when co-located with all kinds of applications, something that was also explained earlier. To further investigate this, we examine the case of the BT benchmark which is also compute-bound

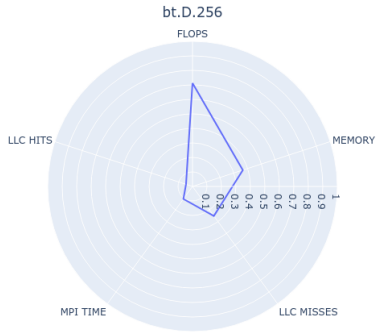
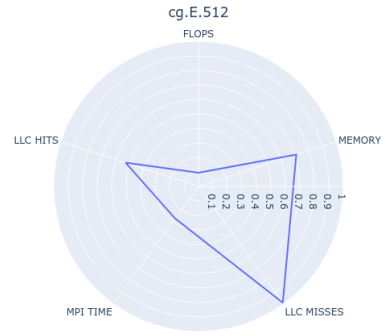
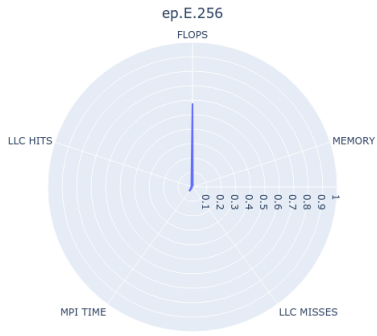
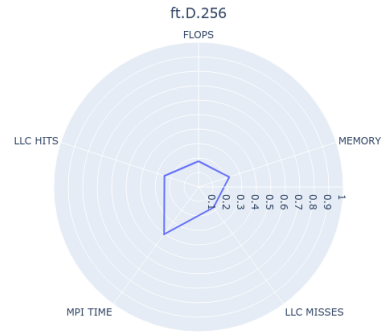
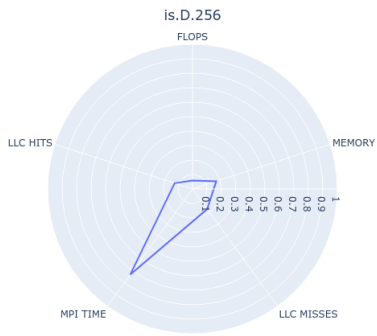
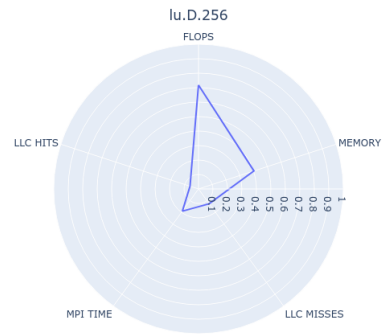
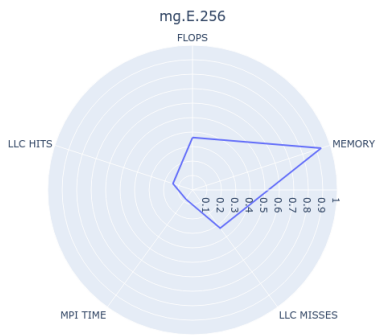
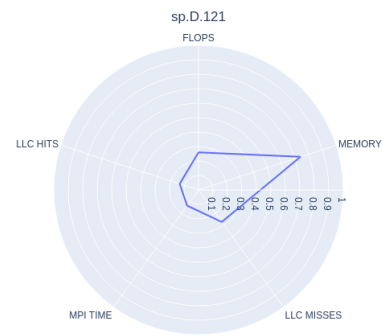


**Figure 4.6.** Heatmap of the used SPEC benchmarks in a co-execution environment (Average Speedup : 1.14)

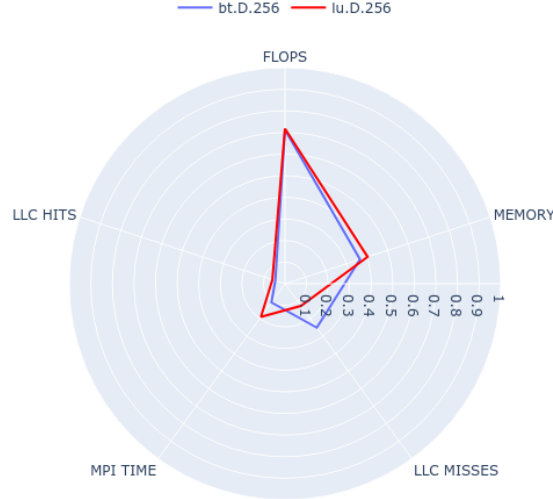
as seen in Figure 4.7a. This benchmark is also most of the times stationary with speedup values never exceeding 1.09 but may also experience severe slowdowns when co-located with SP or MG. The difference between BT and EP lies in the fact that BT also has a considerable amount of memory accesses taking place. SP and MG are two heavily memory-bound benchmarks and, as it is intuitively expected, co-locating two benchmarks that make extensive use of the memory resources creates increased contention for the said resources, thus decreasing the performance. In that case, the least memory-intensive benchmark seems to pay the biggest price. Turning our attention to the LU benchmark, we notice that the spider plots of BT and LU are strikingly similar, as seen in Figure 4.8.

However, LU seems to avoid significant slowdown even when co-located with SP or MG. Additionally, even if most of the times it is virtually stationary, it certainly achieves larger speedups than BT or EP, even reaching 1.16. LU's most important difference from BT seems to be its communication, which is higher than that of BT. From a communication perspective, more resources means more buffers necessary for receiving and sending data and access to network ports with potentially less traffic if co-located with a non-communication intensive benchmark. Furthermore, we observe that these three benchmarks constitute good neighbors for other benchmarks, as they allow their neighboring benchmarks to exhibit speedups (i.e., the columns corresponding to these benchmarks



(a) *bt.D.256*(b) *cg.E.512*(c) *ep.E.256*(d) *ft.D.256*(e) *is.D.256*(f) *lu.D.256*(g) *mg.E.256*(h) *sp.D.121*

*Diploma Thesis* **Figure 4.7.** Spider Plots for each one of the 8 utilized NAS benchmarks



**Figure 4.8.** *Comparison of BT and LU*

in the heatmap are predominantly green). This is expected since compute-bound applications make limited use of shared resources in the node and mostly utilize the cores to which they have exclusive access. Rounding up this discussion around the three compute-bound applications, we have the first set of empirical rules:

#### Empirical Rules Part One: Compute-bound Applications

- Heavily compute-bound applications are stationary when co-located with all kinds of benchmarks.
- Compute-bound applications with a considerable amount of memory accesses are stationary in all cases, except when co-located with memory-bound applications. In that case they exhibit a slowdown.
- Compute-bound applications with a considerable amount of communication can exhibit a speedup when co-located with non-communication-bound or non-memory-bound benchmarks.
- Compute-bound applications are good neighbors for other applications as they make minimal use of shared memory or communication resources in a node.

### 4.3.2 Memory-bound applications

Afterwards, we will discuss the behavior of memory-bound applications when it comes to co-execution. The most predominantly memory-bound applications in the heatmap of Figure 4.5 are MG and SP. We notice that these applications exhibit remarkable speedups in almost all co-location cases, reaching even 1.71. The only case where their speedups deteriorate is when they are co-executed with other heavily memory-bound applications, due to the increased resource contention and the reduced memory bandwidth available to each application. The slowdown that

these applications will experience in this case will depend on how much memory-intensive their neighboring benchmark is. At the same time, both MG and SP seem to be bad neighbors for other benchmarks, given their monopolization of the shared memory resources (i.e., the columns corresponding to these benchmarks in the heatmap are predominantly red).

From the spider plot in Figure 4.7b, we can see that the CG benchmark is also memory-bound. However, a quick look at the heatmap in Figure 4.5 shows that the CG benchmark behaves much differently than SP and MG. Specifically, it never achieves the remarkably high speedups of the other two benchmarks, it experiences tremendous slowdowns when co-located with memory-bound applications like SP and MG and it is also surprisingly a good neighbor for other benchmarks! To explain this divergent behavior we study the CG benchmark more intensively. As mentioned in [48], the CG benchmark’s memory accesses are irregular, due to the fact that it is performing calculations on a large sparse matrix. Irregular memory accesses tend to result in inefficient bandwidth utilization, as they are unpredictable and prevent the memory system from efficiently prefetching or batch-loading data, thus increasing the memory latency. In sequential memory access, the system can anticipate the next memory locations to be accessed, thereby making more efficient use of the bandwidth and enabling streaming or pre-loading of data into the cache. Furthermore, irregular accesses often result in fetching cache lines that are not fully utilized. For example, if an algorithm accesses memory in a random order, each memory access could cause a separate cache line to be fetched, leading to a waste of cache space. At the same time, the lack of spatial locality of the accessed data means that there is a high probability that the requested data will not be in the cache and will require fetching it from main memory. Consequently, memory-bound applications with irregular memory accesses will experience many cache misses. This is evident for the CG benchmark in its spider plot in Figure 4.7b. The cache misses experienced by CG are much higher than any other NAS benchmark. This characteristic of this type of memory-bound applications can be utilized to distinguish them from regular memory bandwidth bound applications. Furthermore, CG’s bandwidth underutilization justifies why it has different co-location behavior from MG or SP. Specifically, if an application underutilizes the memory bandwidth, increasing it by allocating more resources will not significantly help the application’s performance. By underutilizing the bandwidth, this type of applications are also not monopolizing the memory resources, thus being good neighbors to other applications. Finally, being co-located with someone who indeed monopolizes the memory resources further exacerbates memory latency, adding to the inefficiencies of their already suboptimal memory access patterns. Through these observations, we devise a second set of empirical rules:

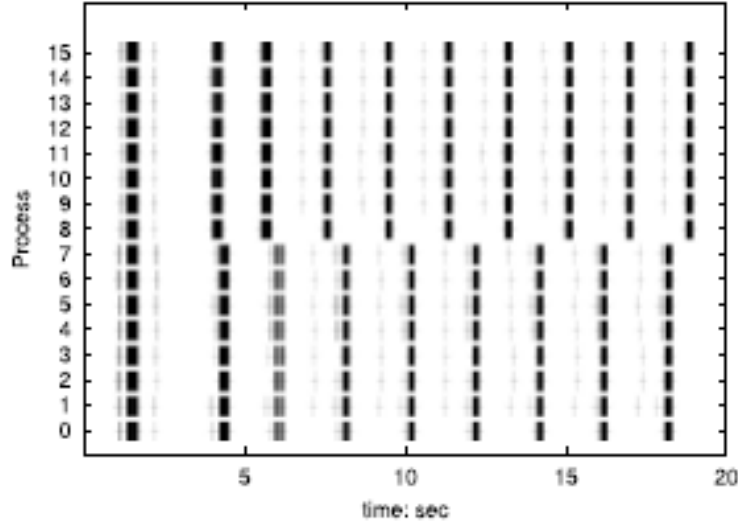
**Empirical Rules Part Two: Memory-bound Applications**

- Memory-bound applications can achieve remarkable speedups when co-located with compute-bound or communication-bound applications.
- When co-located with other memory-bound applications, their performance is worsened, albeit not tremendously. The more heavily memory-bound application out of the two in the pair tends to do better.
- Memory-bound applications are bad neighbors for other applications due to their high demands on the system's memory resources.
- Memory-bound applications that carry out irregular memory accesses cannot achieve high speedups due to their bandwidth underutilization. For the same reason, they constitute good neighbors for other applications. Their already significant memory latency is exacerbated when co-executed with memory-bound applications, leading to a tremendous slowdown.

**4.3.3 Communication-bound applications**

Our final two NAS benchmarks, FT and IS are both communication-bound as illustrated in Figures 4.7d and 4.7e. From the heatmap in Figure 4.5, we infer that these two benchmarks tend to have high speedups, albeit not as high as memory-bound benchmarks. This is expected, due to the higher number of memory buffers in spread mode. On the contrary to memory-bound benchmarks though, they seem to maintain this behavior with virtually all potential neighbors. This stems from the fact that they do not utilize the memory bandwidth as much as memory-bound applications and thus have higher tolerance of such bad neighbors. At the same time, even when co-located with other communication-bound applications (e.g. IS with FT), we do not observe a slowdown as we would expect due to the common use of the network ports. This is a consequence of the fact that it is highly unlikely that two distinct benchmarks will have the exact same communication patterns and thus interfere with the execution of the other. But even when we co-locate two instances of the same benchmark, we notice either stationary behavior (is.E.256 with is.E.256 with a speedup of 1.02) or even a considerable speedup (ft.D.64 with ft.D.64 with a speedup of 1.13). An explanation for this is given in [8], where it is shown that the co-execution of two instances of the same NAS benchmark will cause conflicts during the initial stages of the communication but then because of the stall that one of the two instances will suffer, their communication patterns go out of sync. Thus the conflicts from that point onwards are decreased. This phenomenon is illustrated in Figure 4.9.

As a result, communication-bound benchmarks seem to be suitable for co-location and more often than not experience speedups during these circumstances. For the same reasons, they are usually good neighbors to other benchmarks, as shown in the columns of the heatmap. However, it is important to note that communication behavior may easily become unpredictable. Factors such as link contention in the overall system may exacerbate performance. As mentioned before, processes of the same application have similar demands on the system and thus lead to contention in shared

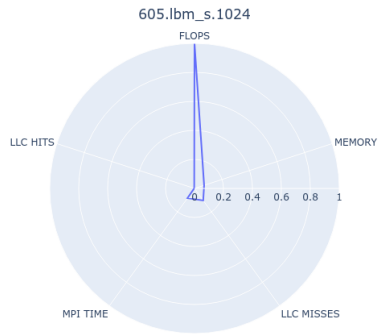


**Figure 4.9.** *Communication patterns of two instances of the same program (FT benchmark) in co-execution mode [8]*

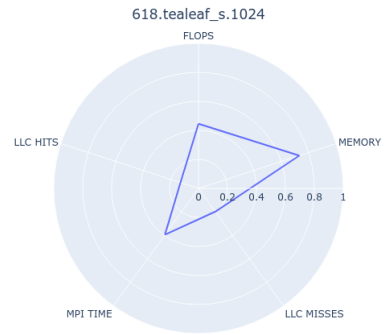
resources like network ports. This is limited in spread or co-execution mode, where less processes of the same application are present in a node. However, spread mode incurs one potential downside for communication-bound applications. When spreading a job to more nodes, we inevitably increase the required communication. In the above experiments, the benefits of the increased resources seem to outweigh the added cost of the increased communication. Nonetheless, the impact of this increased communication if we further increase the number of nodes in spread mode or during periods of high link congestion in the system, might grow and as a result slow down the application. Other factors, like communication patterns (All-to-all, point-to-point) or message sizes can also play a role.

NAS benchmarks did not manage to give us any more insights into the communication behavior under co-execution. For this reason we also turned our attention to the SPEC benchmarks. The spider plots for the utilized SPEC benchmarks whose heatmap is shown in Figure 4.6 are presented in Figure 4.10. We observe that this benchmark suite contains two heavily communication-bound benchmarks: 621.miniswp\_s (Figure 4.10d) and 635.weather\_s (Figure 4.10g). However, contrary to the behavior of FT and IS, the heatmap in Figure 4.6 shows us that these two benchmarks do not display high speedups but rather stay stationary most of the times. To explore this divergent behavior, we examine the mpiP reports of these two SPEC benchmarks in more detail. Specifically, we uncover how much time is spent in each distinct MPI directive. In the first row of table 4.1, the percentage of the total MPI time spent in each MPI call category is shown for the two SPEC benchmarks. For comparison purposes, in the second row of the table we present the same analysis for the FT benchmark.

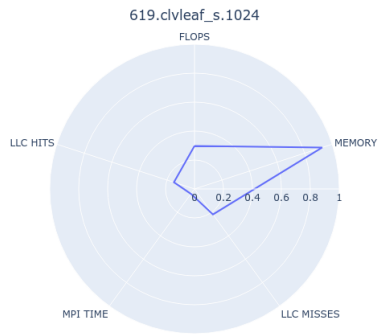
The MPI time of the 635.weather\_s benchmark is dominated by MPI\_Waitall calls which as mentioned in Section 2.1.3 represent idle waiting rather than effective communication. Consequently,



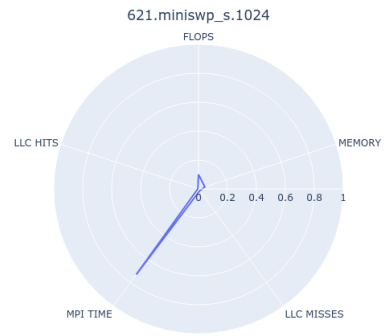
(a) 605.lbm\_s.1024



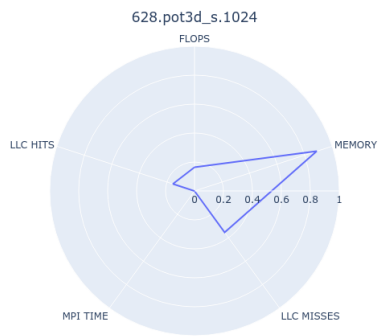
(b) 618.tealeaf\_s.1024



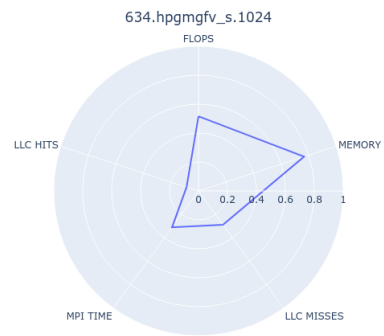
(c) 619.clvleaf\_s.1024



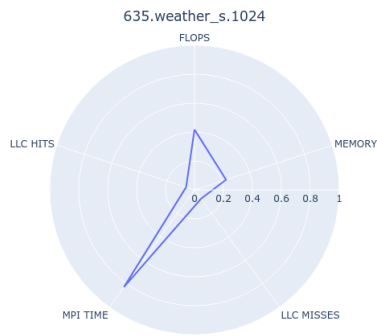
(d) 621.miniswp\_s.1024



(e) 628.pot3d\_s.1024



(f) 634.hpgmgfv\_s.1024



(g) 635.weather\_s.1024

**Figure 4.10.** Spider Plots for each one of the 7 utilized SPEC benchmarks

<b>635.weather_s.1024</b>	<b>621.miniswp_s.1024</b>
Callsites: 19456	Callsites: 15360
Waitall: 99.90%	Recv: 85.24%
Isend: 0.04%	Send: 14.49%
Irecv: 0.06%	Barrier: 0.24%
Barrier: 0.00%	Comm_split: 0.02%
	Comm_free: 0.00%
	Allreduce: 0.00%
<hr/>	
<b>ft.D.256</b>	
Callsites: 3840	
Alltoall: 97.16%	
Reduce: 1.70%	
Barrier: 0.88%	
Comm_split: 0.19%	
Bcast: 0.07%	

**Table 4.1.** *Percentage breakdown of total MPI time spent on each MPI directive*

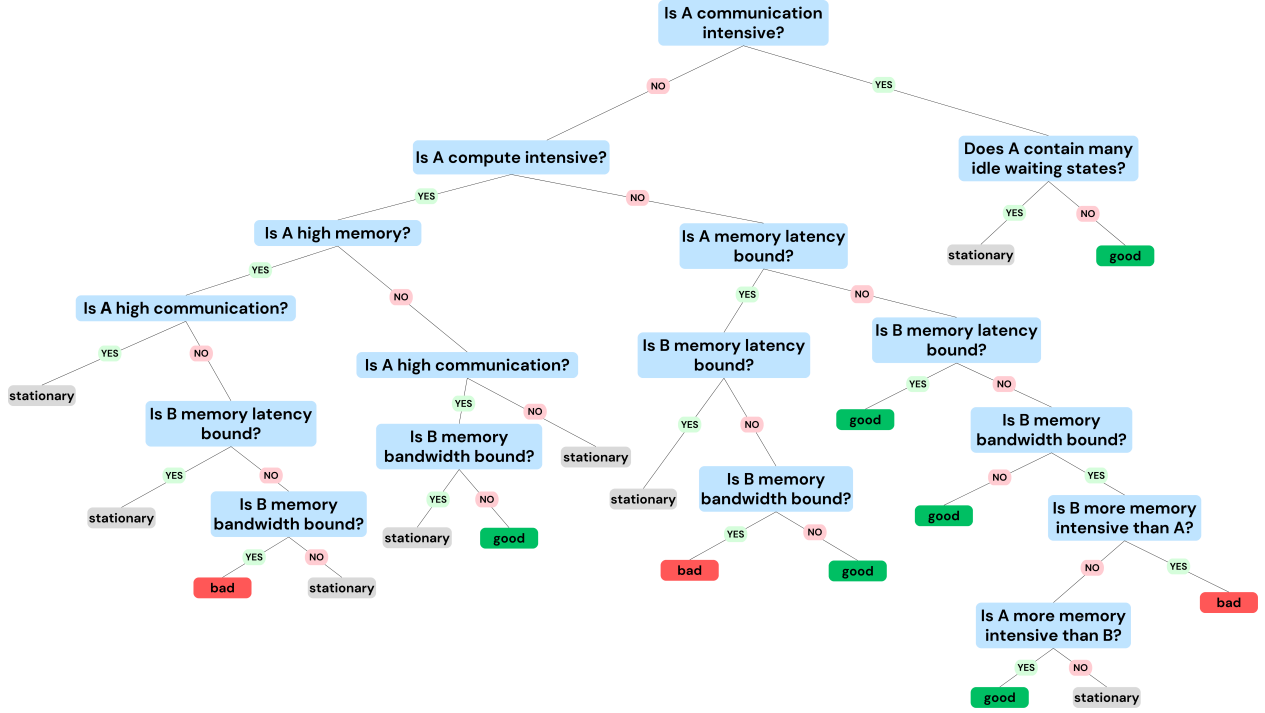
this portion of the execution time cannot be optimized further. Similarly, the MPI time for the 621.miniswp\_s benchmark is predominantly spent on MPI\_Recv calls. These calls also contribute to idle waiting time rather than effective communication, as MPI must wait for the sender process to prepare and transmit the required data. A significant amount of time spent in this call indicates an inefficient communication pattern, resulting in increased waiting times for receiving processes as they await data transmission. On the other hand, the MPI time of the FT benchmark is dominated by MPI\_Alltoall calls which constitute useful communication. As a result, we must take into consideration the amount of idle waiting time that is included in the MPI time of each benchmark when devising our empirical rules. Hence, the third set of empirical rules is presented below:

#### **Empirical Rules Part Three: Communication-bound Applications**

- Communication-bound applications that mostly contain useful communication are more often than not experiencing a speedup during co-execution.
- On the contrary, communication-bound applications that are dominated by idle waiting times exhibit a more stationary behavior during co-execution.
- Communication-bound benchmarks are good neighbors to other benchmarks due to the fact that they do not monopolize shared resources and the communication patterns of distinct applications are usually different.
- The behavior of communication-bound benchmarks may be more unpredictable if a job is spread in a large number of nodes or during periods of high link congestion in the system.

### 4.3.4 Prediction Results

Using the aforementioned rules we constructed an empirical decision tree, which is presented in Figure 4.11. As mentioned, this model receives as input two applications A and B and predicts the speedup characterization of application A if co-located with B. The three possible categorical labels are good, stationary and bad.



**Figure 4.11.** Decision Tree made by utilizing the aforementioned empirical rules

We utilized this model to predict the heatmap of the NAS benchmarks, by applying it to every single possible pairing from the NAS application pool. The predicted categorical heatmap can be seen in Figure 4.12. In Table 4.4, we analyze and present the model's accuracy.

BenchmarkA	bt.D.256	cg.D.128	cg.D.64	cg.E.512	ep.E.256	ft.D.256	ft.D.64	is.D.256	lu.D.256	mg.E.256	sp.C.64	sp.D.121	sp.D.256
bt.D.256	stationary	stationary	stationary	stationary	stationary	stationary	stationary	stationary	stationary	bad	bad	bad	bad
cg.D.128	good	stationary	stationary	stationary	good	good	good	good	good	bad	bad	bad	bad
cg.D.64	good	stationary	stationary	stationary	good	good	good	good	good	bad	bad	bad	bad
cg.E.512	good	stationary	stationary	stationary	good	good	good	good	good	bad	bad	bad	bad
ep.E.256	stationary	stationary	stationary	stationary	stationary	stationary	stationary	stationary	stationary	stationary	stationary	stationary	stationary
ft.D.256	good	good	good	good	good	good	good	good	good	good	good	good	good
ft.D.64	good	good	good	good	good	good	good	good	good	good	good	good	good
is.D.256	good	good	good	good	good	good	good	good	good	good	good	good	good
lu.D.256	good	good	good	good	good	stationary	stationary	stationary	good	stationary	stationary	stationary	stationary
mg.E.256	good	good	good	good	good	good	good	good	good	stationary	good	good	good
sp.C.64	good	good	good	good	good	good	good	good	good	bad	stationary	bad	bad
sp.D.121	good	good	good	good	good	good	good	good	good	bad	good	stationary	good
sp.D.256	good	good	good	good	good	good	good	good	good	bad	good	bad	stationary

**Figure 4.12.** Predicted categorical heatmap for the NAS benchmarks



We define the speedup bounds for the three categorizations as shown in Table 4.2.

Characterization	Speedup Bounds
Good	Speedup $\geq 1.1$
Stationary	$0.95 \leq \text{Speedup} < 1.1$
Bad	Speedup $< 0.95$

**Table 4.2.** *Speedup bounds for each categorization*

Given that what is considered good or bad around the case of speedup = 1 is subjective, we introduce a sufficiently correct category of predictions during evaluation. The bounds for this category are more lenient and are detailed in Table 4.3.

Characterization	Speedup Bounds
Good	Speedup $\geq 1.05$
Stationary	$0.90 \leq \text{Speedup} < 1.15$
Bad	Speedup $\leq 0.95$

**Table 4.3.** *Sufficiently correct speedup bounds for each categorization*

In Table 4.4, we analyze and present the model’s accuracy in predicting the NAS heatmap.

Characterization	Results
Fully Correct	141 out of 169 - 83.43%
Sufficiently Correct	12 out of 169 - 7.1%
Wrong	16 out of 169 - 9.47%

**Table 4.4.** *Analysis of the results of the empirical model for the NAS benchmarks*

We observe that are model accurately predicted 83.43% of the pairs in the NAS heatmap, while 7.1% of its predictions fall within the ‘Sufficiently Correct’ category. As a result, **the predictions were at least sufficient for 90.53% of the pairs**. In order to evaluate whether the model is significantly biased or suffers from overfitting due to the fact that the empirical rules were heavily based on the NAS benchmarks, we also predicted the SPEC heatmap using it. The predicted categorical heatmap can be seen in Figure 4.13.

BenchmarkA	605.lbm_s.1024	618.tealeaf_s.1024	619.clvleaf_s.1024	621.miniswp_s.1024	628.pot3d_s.1024	634.hpgmgfv_s.1024	635.weather_s.1024
605.lbm_s.1024	stationary	stationary	stationary	stationary	stationary	stationary	stationary
618.tealeaf_s.1024	good	stationary	bad	good	bad	bad	good
619.clvleaf_s.1024	good	good	stationary	good	good	good	good
621.miniswp_s.1024	stationary	stationary	stationary	stationary	stationary	stationary	stationary
628.pot3d_s.1024	good	good	bad	good	stationary	good	good
634.hpgmgfv_s.1024	good	good	bad	good	bad	stationary	good
635.weather_s.1024	stationary	stationary	stationary	stationary	stationary	stationary	stationary

**Figure 4.13.** *Predicted categorical heatmap for the SPEC benchmarks*

The model’s accuracy in predicting the SPEC heatmap is presented in Table 4.5. While the percentage of fully correct predictions is considerably smaller (63.27%) than the one for the NAS

benchmarks, we observe that a large number of predictions (22.45%) fall under the ‘Sufficiently Correct’ category. As a result, **the predictions were at least sufficient for 85.72% of the pairs**, which is once again satisfactory and the wrong predictions account only for 14.29% of the total pairs.

Characterization	Results
Fully Correct	31 out of 49 - 63.27%
Sufficiently Correct	11 out of 49 - 22.45%
Wrong	7 out of 49 - 14.29%

**Table 4.5.** *Analysis of the results of the empirical model for the SPEC benchmarks*

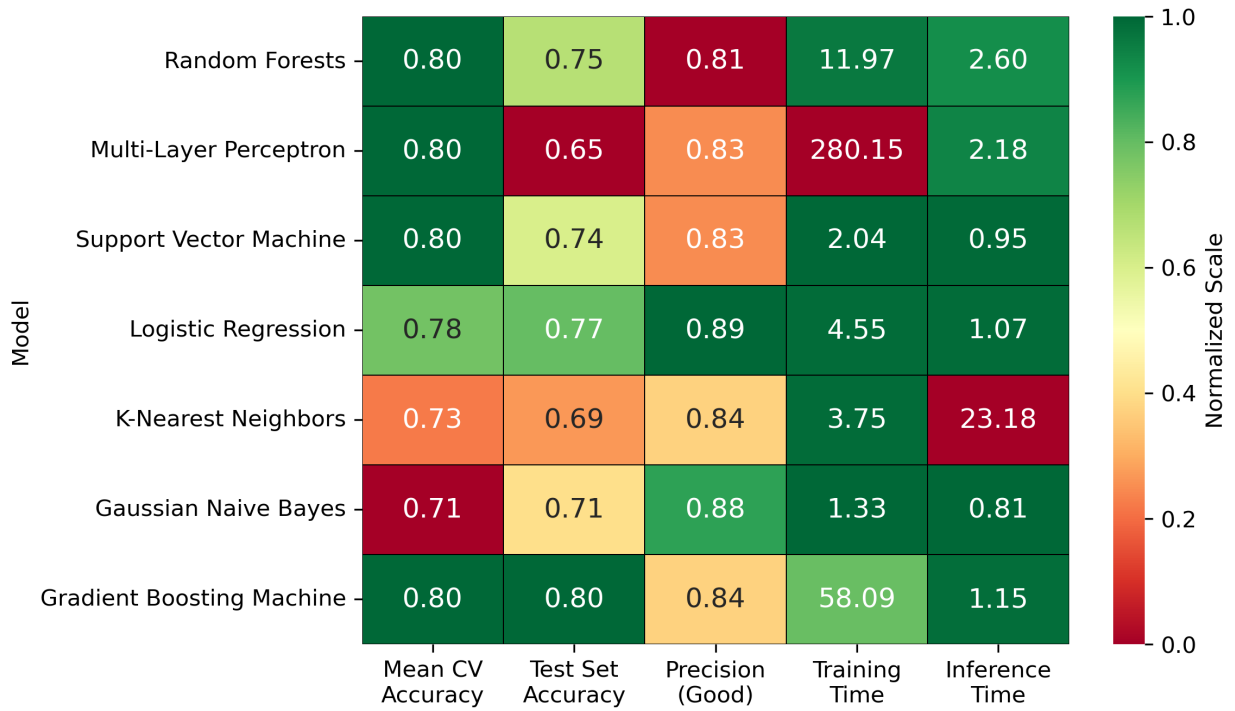
## 4.4 Heatmap Prediction using Machine Learning

Given the prohibitive cost of running all possible co-location pairs to evaluate their performance, Machine Learning models have been utilized in previous co-location research with the intent to predict the heatmap of a particular application pool. In [29], the authors claim that utilizing simple heuristics like for example only one particular performance metric (e.g. only cache misses) does not leverage the entire potential of co-scheduling. For this reason they use various performance counters so as to train Machine Learning models to predict the performance degradation of applications when co-located. They investigate the impact of using a plethora of performance counters and derived metrics in the training phase of their models and compare the accuracy of different Machine Learning models. Relying on these conclusions, we train our own Machine Learning classification models, so as to compare them with the empirical approach. We utilize the 5 axes previously presented in the spider plots as features in our training set, since we believe that they provide a holistic overview of the application’s behavior and were also used when devising the empirical model.

The first step is to decide which machine learning models fit our intended purpose. To do this, we need to understand the use case. This Machine Learning algorithm will need to make frequent predictions, as the application pool on a supercomputer frequently changes with users submitting new jobs. Consequently, new heatmaps must be continuously predicted to provide our advanced co-scheduling algorithm with the necessary information. As a result, our model must not have a large inference time. The training time is less critical since the model can be updated sporadically and trained during off-peak hours, such as at night, when the system is under less load. Additionally, as this is not a critical task, using an outdated model for a few hours during the training process is not detrimental. In this particular case, we aim to use a low number of features given the fact that profiling a large number of hardware events that exceeds the available hardware counters leads to the need for excessive multiplexing which lowers the accuracy of the estimations. Additionally we aim to use the derived metrics calculated earlier that provide us with a holistic overview of the application’s behavior to also compare the models with our empirical model. The fact that co-scheduling is not a widely used method and has not yet found its way to production systems means that the datasets we possess for training and validation are small and come mainly from experiments

conducted during research. Even if this may change in the future, models that do not require large datasets for their training phase may be more suitable for the time being. Interpretability would also be desirable in order to enhance the community's knowledge around performance analysis, while outliers may be common because the experiments are conducted on real systems, where load fluctuations, particularly in the interconnection network, can occur frequently and affect an application's execution time.

With all these in mind, we trained seven different types of Machine Learning models and compared their inference results. For each model type, we trained a plethora of models using all possible combinations of many of their hyperparameters. For each model, we conducted 5-fold cross-validation and calculated the average of the fold scores. We then retained for each model type, the model with the highest average cross-validation accuracy score and evaluated it using an unseen test set. Figure 4.14 presents some metrics regarding the best model from each model type. The training time and inference time metrics are in milliseconds.



**Figure 4.14.** Evaluation of the best-performing models from each model type

These results can be explained by examining each model type's inherent characteristics [31]. In particular:

- Neural Networks like **Multi-Layer Perceptron (MLP)** require a large training set in order to yield accurate results. Even if this model type achieved a mean cross-validation accuracy of 0.8, it fell completely short when predicting the unseen data, achieving only an accuracy of 0.65. It also required a very large training time, another innate feature of neural

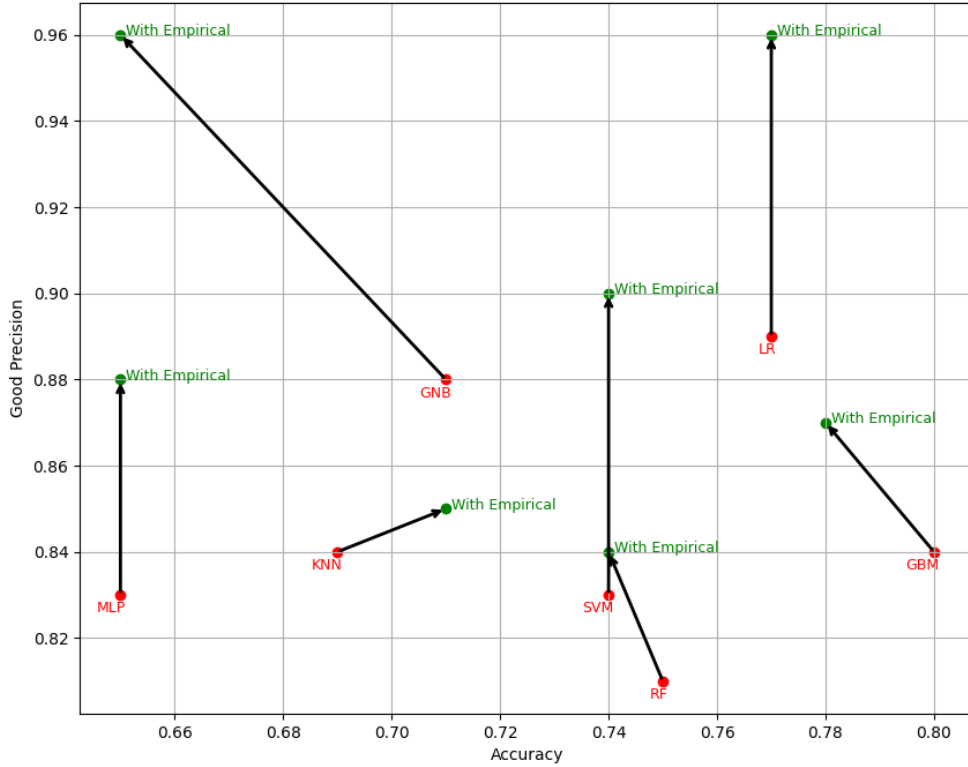
networks compared to traditional machine learning techniques. They are also not humanly interpretable.

- The best model types overall are **Gradient Boosting Machine (GBM)**, **Random Forests (RF)** and **Logistic Regression (LR)**. The former two methods constitute ensemble learning techniques and guarantee higher accuracy and less overfitting than simple Machine Learning algorithms [49]. However they require more training time. Logistic Regression is a simple model that attempts to find the correlation between a dependent and one or more independent variables. The relatively small training set that we have at our disposal benefits this simple model.
- Other simple models like **K-Nearest Neighbors (KNN)** and the probabilistic technique **Gaussian Naive Bayes (GNB)** do not seem to perform well. This may be due to the complexity in the relations between the features and the co-location characterization. K-Nearest Neighbors also has a very high inference time, which is expected since each prediction requires calculating the distance from all points in the training set. A large inference time might be prohibitive in real-time co-scheduling decision-making.
- Finally **Support Vector Machine (SVM)** seems to have a good accuracy in the unseen data. It can accomplish solid predictions even with smaller datasets, but its main strength is classification in cases where the training set is high-dimensional (e.g. text classification, bioinformatics etc.), which is not the case here where each sample contains only 8 features.

A main limitation for many of these models is the relatively small training set. As co-scheduling is further integrated in production systems, this can easily change, as we are provided with more real-world data. Afterwards, the efficiency of neural networks can be re-examined, especially given the fact that training time may not be as crucial as fast inference and accuracy. Another important factor is the stability of the model when the training set is enhanced, which will frequently occur during model re-training and the sensibility to outliers, which may occur if measurements are taken during high system load. These limitations also pose challenges for simple machine learning models, as far as real-world implementation is concerned.

As highlighted in [29], underestimating application degradation in a co-location scenario can have severe consequences in real-time environments, as it may lead the scheduler to mistakenly co-locate incompatible applications. Similarly, in our use case, False Positives labeled as “good” can mislead the scheduler, adversely impacting the workload’s makespan. For this reason, we included the precision of our models as far as the “good” label is concerned in the heatmap of Figure 4.14 and it could be useful to try and improve it. One way to achieve this is by utilizing two models at the same time and only labeling a sample as “good” if both models have predicted it as “good”. This may of course affect the overall accuracy of the model. We first implemented this logic by using our

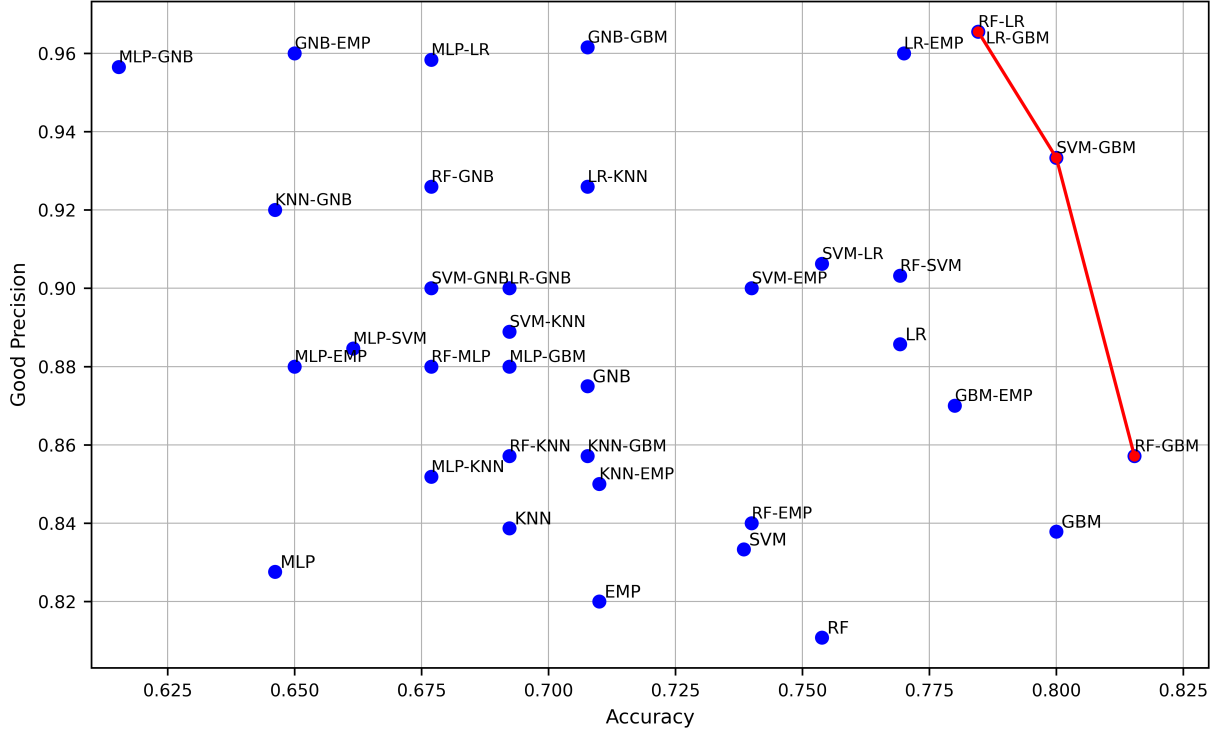
empirical model in conjunction with all the ML models in Figure 4.14. Figure 4.15 illustrates the variations in accuracy and precision for the “good” label across each ML model upon implementing the aforementioned policy.



**Figure 4.15.** *Impact of using all ML models in conjunction with our empirical model on accuracy and precision*

We observe that in all cases the precision of the “good” label is increased even reaching 0.96, proving that this approach is promising for our intended purpose. However, we also observe that the overall accuracy may sometimes decrease which is undesirable. We end up with a two-criteria optimization problem: we want to have large overall accuracy while also achieving large precision for the “good” label. To further investigate this, we perform this logic once again for all possible pairs of the Machine Learning models of Figure 4.14. We present the results in Figure 4.16. The red line depicts the Pareto frontier of the scatterplot, which indicates that the combinations of models closest to satisfying concurrently both our optimization objectives are RF-LR, LR-GBM, SVM-GBM and RF-GBM. The four models that comprise these pairs were also among our most well-performing models when evaluated standalone. Depending on the desired level of precision for the “good” label, we can select one of these combinations for use. It is important to note that by raising the precision for the “good” label, we may potentially decrease the recall of this label. For the cases of RF-LR, LR-GBM, SVM-GBM, the recall of the “good” label is decreased by 9% in comparison to the largest recall between the two models of each pair. In the case of RF-GBM, where the precision for the “good” label is lower, the recall is decreased by 3%. Not predicting specific good co-locations (sacrificing recall) constitutes a missed opportunity for the scheduler to make better use of the application pool. On the other hand, mispredicting many bad co-locations as

good (sacrificing precision) can result in a slowdown in the overall makespan, due to incompatible pairings by the scheduler. This is a trade-off that is worth investigating and may also depend on the co-scheduling algorithm itself. However, a conservative approach that makes sure that the predicted good co-locations are indeed good, coupled with the innate benefits of co-scheduling can ensure that we avoid makespan slowdowns.



**Figure 4.16.** *Pareto Plot of all our models and all possible combinations of models*

---

## Co-Scheduling Simulations

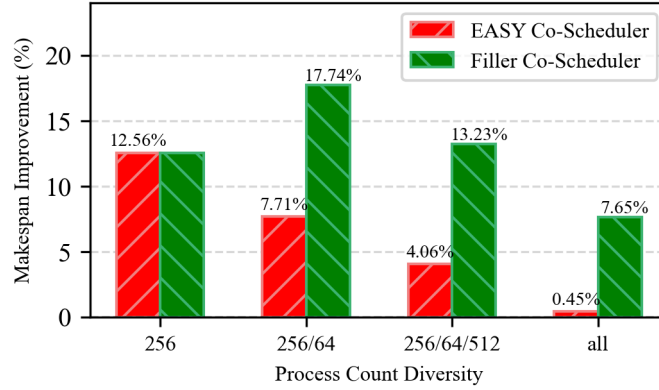
---

In this chapter, we will demonstrate the benefits of co-scheduling as far as makespan speedup and throughput are concerned. We firstly propose a simple Co-Scheduler called EASY Co-Scheduler, which places applications in the system in a First Come First Served manner, while also utilizing EASY backfill, as described in Section 3.2.2. In addition to the EASY Co-Scheduler, we will introduce a method for developing more advanced co-scheduling algorithms, by trying to shed light on important optimization metrics. To achieve these, we will conduct experiments utilizing a novel simulator tool that was developed in CSLab NTUA and is called **ELiSE (Efficient Lightweight Scheduling Estimator)**. ELiSE has the ability to simulate the scheduling or co-scheduling of workloads in a user-defined configuration. It requires the compact execution times of the applications in the simulated workloads, as well as the heatmap of co-execution times so as to simulate co-scheduling algorithms. Both basic and sophisticated (co-)scheduling algorithms have already been implemented in ELiSE and its simple object-oriented design allows the seamless development of new, more complex algorithms by using inheritance from simpler implemented ones. In the end, we will also pinpoint use cases, where sophisticated insights into our workloads, like the ones gained from the methodologies in Chapter 4 are useful.

### 5.1 Simple Experiments

Co-scheduling can result in a considerable improvement in the makespan of a particular workload. In order to prove it, we conducted experiments using ELiSE. We utilized four workload types of 500 jobs each. Each workload type consisted of jobs with an increasing variety of process counts. For instance, diversity\_1 included only jobs running on 256 processes, diversity\_2 included jobs running on either 256 or 64 processes, and so forth. For each workload type, we conducted four shuffled experiments and measured the average makespan speedup of the EASY Co-Scheduler and

a sophisticated Co-Scheduler, called Filler Co-Scheduler with regards to the state-of-the-practice EASY scheduler. The Filler Co-Scheduler, tries to decrease the fragmentation of the co-schedule by selecting the best-fit job to enter the nodes at each step, based on its number of processes, while also trying to re-order the waiting queue as little as possible. As a result, it aims at improving the utilization of the system, which is one of the key metrics for system administrators. Figure 5.1 presents the results of these experiments.



**Figure 5.1.** *Makespan improvement for two co-scheduling algorithms with regards to the EASY Scheduler and for diverse process count mixes*

It is evident that co-scheduling results in a makespan speedup for all cases, regardless of the process count diversity. However not all algorithms are equally effective in different co-scheduling scenarios. In the case of diversity\_1, where all submitted jobs have the same process count, there is nothing to be gained by trying to fill the created gaps in the co-schedule and as a result the Filler Co-Scheduler cannot further improve the makespan. Furthermore, the makespan improvement achieved by the EASY Co-Scheduler gradually decreases as process count diversity increases, whereas the Filler Co-Scheduler is useful in larger process count diversities as more gaps are created. This study proves that optimizations on the basic EASY Co-Scheduler are necessary in diverse workloads and that a Co-Scheduler’s performance is reliant on the type of workload submitted. Moreover, no sophisticated Co-Scheduler comes without drawbacks. Since the primary goal of the Filler Co-Scheduler is to ensure high utilization, it often achieves this by completely disrupting the order of the waiting queue, thereby compromising fairness.

Nevertheless, even the simple EASY Co-Scheduler can yield significant makespan improvements in several cases. The results in Table 5.1 demonstrate the high makespan improvements of workloads containing jobs with equal process counts and with varying average speedups, calculated based on the pairwise speedups of all possible job pairs in each workload, using the Marconi heatmap. Each workload consists of 500 jobs, and for each average speedup case we conducted four experiments using ELiSE to simulate a cluster with 100 nodes and 48 cores.

These findings further emphasize a correlation between the average speedup of all job pairs in



Mean Job Speedup	Makespan Improvement (%)
1.07	24.42%
1.10	25.29%
1.12	25.87%
1.155	30.63%

**Table 5.1.** *Makespan Improvement with regards to the mean job speedup of the used sub-heatmap*

an application pool and the makespan improvement achieved through co-scheduling, presenting an approach to developing a more advanced Co-Scheduler. As a result, co-scheduling involves numerous trade-offs and requires careful consideration of many parameters when designing co-scheduling algorithms and selecting which workloads to co-schedule.

## 5.2 Towards Sophisticated Co-Schedulers

One preliminary approach to studying co-location of applications would be to exhaustively calculate all potential pairings from the application pool and calculate the possible average speedups and slowdowns. This analysis can provide us insights as to which application mixes would work well in a co-scheduling environment. However, this approach cannot provide insights into makespan or user satisfaction in real-world scenarios, as it falls significantly short of being a comprehensive co-scheduling study by amongst others neglecting the possibility that an application may pair with multiple other applications during its runtime and in general not taking into account the temporal aspect of scheduling. A more complete temporal analysis would require the usage of a co-scheduling estimator tool like ELiSE. We conducted 100 experiments for a particular application pool using ELiSE, the heatmap from the ARIS supercomputer and a simulated cluster of 200 nodes with 20 cores per node. The workload for each experiment consisted of the exact same applications, but their order was each time shuffled. The simple EASY Co-Scheduler presented earlier was utilized. This way, one can attempt to find different possible co-schedules and determine which metrics are of importance to optimize system and user satisfaction. For system satisfaction, we utilized the makespan speedup of co-scheduling, compared to the EASY scheduler. As seen in Figure 5.2a, the correlation of makespan speedup with the average job speedup as depicted by the Pearson correlation coefficient is significant, as it is intuitively expected, suggesting that an algorithm oriented at improving the average job speedup will be beneficial for the total makespan speedup in this case. The makespan speedup improvement seems to be even more highly correlated with the weighted average job speedup (Figure 5.2b), which is more representative in a co-scheduling environment and is calculated through the following formula:

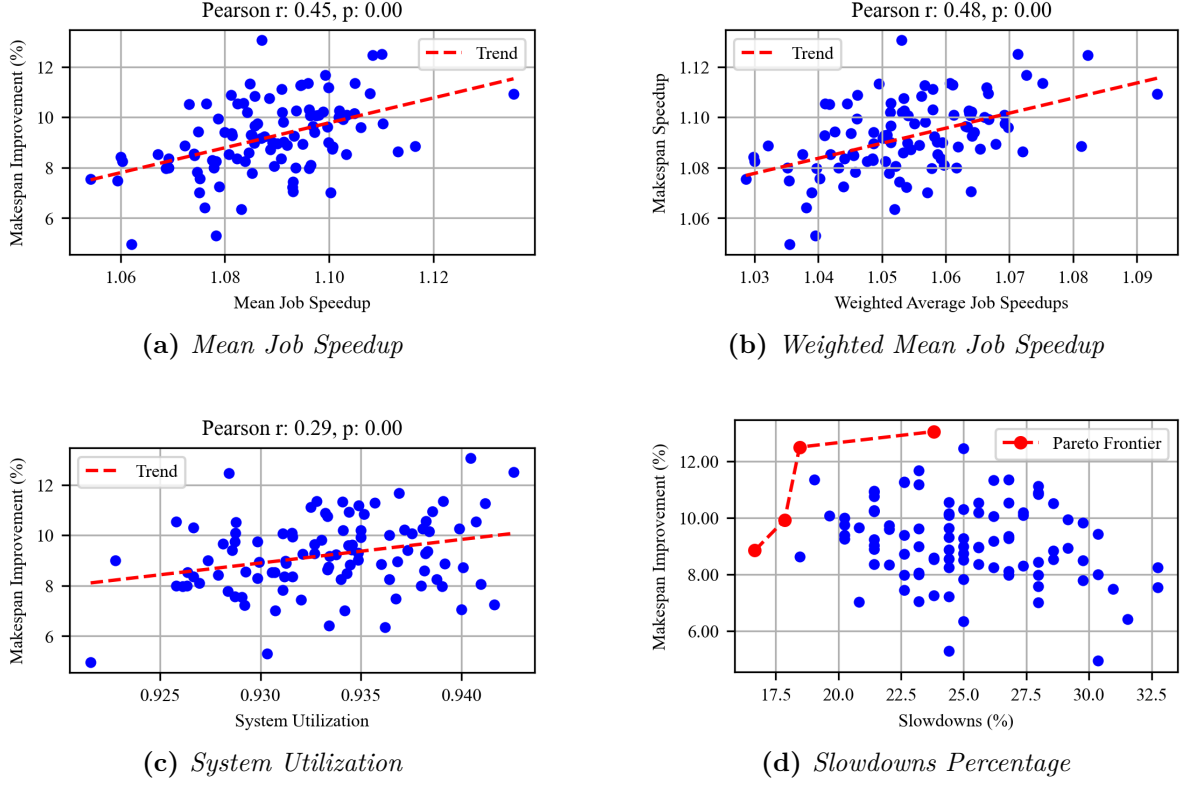
$$\text{wavg\_job\_speedup} = \frac{\sum_{i=1}^N \text{speedup}_i \cdot \text{weight}_i}{\sum_{i=1}^N \text{weight}_i}$$

where the weights are calculated as:

$$\text{weight}_i = \frac{\text{runtime}_i \cdot \text{procs\_per\_app}_i}{\text{max\_weight}}$$

and the maximum weight is defined as the following product:

$$\text{max\_weight} = \text{makespan}_{\text{max}} \cdot \text{nodes} \cdot \text{cores}$$



**Figure 5.2.** Correlation between Makespan Improvement and various metrics related to system throughput and user satisfaction

The weighted average job speedup takes into consideration both the spatial (procs\_per\_app) and the temporal (runtime) aspect of the co-schedule so as to gain a clearer insight into how much speedup our entire application pool experienced. The max\_weight represents the entire area of the co-schedule, since it is a product of the maximum values of its two axes. The average (weighted) job speedup can also be viewed as a crucial metric for user satisfaction, meaning that an algorithm aiming to improve it can satisfy both our optimization goals concurrently.

At the same time though, system utilization seems to also play an important role in improving the makespan speedup, as seen in Figure 5.2c, something that is also verified by the Filler Co-Scheduler’s behavior in the majority of workload cases from the previous section. Finally, Figure 5.2d presents a Pareto Plot illustrating the makespan improvement alongside the percentage of

applications that experienced a slowdown, which is another indicator of user satisfaction. These two parameters create a multi-criteria optimization problem: we need an algorithm that can reach the Pareto frontier so as to balance the objectives of high makespan improvement, indicating high system satisfaction and a low percentage of applications experiencing a slowdown, thereby ensuring user satisfaction. In any case, the abovementioned and many more research directions and trade-off experimentations around co-scheduling can only be possible through the use of a dedicated tool to come closer to real life circumstances rather than relying solely on statistical analysis.

### 5.3 Need for sophistication

In the previous sections, we firstly saw that diverse workloads necessitate a sophisticated co-scheduling approach so as to offer benefits in comparison with simple, traditional scheduling. In addition, we observed that the mean job speedup in a specific co-schedule has a significant impact on improving its makespan. To this end, the techniques discussed in Chapter 4 can be employed so as to gain information about our workloads and provide them to an advanced algorithm, aiming to make informed decisions that enhance the mean job speedup in a co-schedule and, consequently, improve its makespan. The gathered insights about the applications in the application pool can be utilized by the workload manager in a variety of ways:

- To benefit the system administrator, the information can be used in order to reduce the makespan of a particular application pool. A lower makespan entails a higher throughput and thus more work done in less time and with less consumed energy.
- To benefit the users, the information can be used so as to minimize the jobs that experience slowdowns, while making sure that the makespan is at least moderately reduced. In such a scenario, a developed algorithm must find a middle ground between trying to improve the makespan, without treating users unfairly which could result in significant losses of both time and money.

In these two cases, the gathered insights can be utilized either real-time during the co-scheduling process, so as to decide which nodes will be allocated to the job currently at the top of the waiting queue, or in order to reorder a particular waiting queue to possibly achieve higher mean job speedups. The latter method of course compromises fairness, as the arrival order is disrupted. Apart from decision-making during the co-scheduling process, the insights gathered through our proposed methods can also be leveraged so as to decide how to approach a particular workload in advance. Specifically:

- The system can enable or disable co-scheduling altogether for a particular workload, according to its characteristics.

- The system can support two separate job queues: one for simple scheduling and one for co-scheduling. Then, a sophisticated algorithm can decide which jobs will go to each queue in order to improve particular metrics, like throughput or user satisfaction.

In conclusion, co-scheduling appears to hold significant potential for improving many common HPC target metrics. The methods discussed for gathering information about applications can serve as valuable inputs to specialized co-scheduling algorithms designed to enhance these metrics. Given the need to balance the requirements of both system administrators and HPC users, these algorithms must address multi-criteria optimization problems rather than focusing solely on improving a single metric. As demonstrated in [23], this must be seriously taken into consideration, particularly when trying to implement (co-)scheduling algorithms using Machine or Reinforcement Learning techniques. With this work, we aimed to provide a groundwork for the creation of sophisticated Co-Schedulers, by pinpointing ways to gather their necessary inputs, as well as showing the significance of various metrics and the trade-offs that emerge.

---

## Future Research Directions

---

Co-Scheduling, while promising for increasing system throughput and energy efficiency, as well as reducing job runtimes, has not yet been adopted in production HPC clusters. In order for that to be accomplished, modern Workload Managers like SLURM need to be programmed to support it, which can be a cumbersome process. Until this integration begins taking shape, there are many questions that researchers need to find answers to.

Regarding performance analysis and its usage to retrieve application profiles so as to inform co-scheduling decisions, the following questions have to be addressed in future research:

- MPI communication requires further study, incorporating new insights to improve the accuracy of performance predictions for communication-bound applications. In particular, execution during periods of link congestion is certain to increase the runtime of such applications. As shown in [38], a high demand for communication bandwidth is heavily correlated to performance degradation for MPI applications when limited communication bandwidth is available due to link congestion. As a result, link congestion profilers must be integrated into our profiling methodology to leverage link congestion data for predicting the behavior of communication-bound applications. Additionally, apart from the idle time during MPI communication which was exploited by our empirical approach, communication patterns (point-to-point, all-to-all) can also be taken into account when predicting the performance of an application.
- We implemented and evaluated various Machine Learning classification models, as well as a tag-based approach. More research can be performed regarding, Machine Learning Regression

models, like the ones presented in [29], so that co-scheduling algorithms can have more precise speedup estimations at their disposal. It would be important to once again compare the chosen machine learning algorithms according to their innate characteristics to choose the appropriate one for our use case. Another direction would be to focus on co-location-centric approaches. This would be useful, especially in order to choose to enable or disable co-scheduling according to the workload, or to divide a workload into a co-scheduling and a simple scheduling queue.

- As the integration of co-scheduling in production systems advances, more data regarding the behavior of applications in a co-execution environment need to be gathered so as to train and re-evaluate the behavior of approaches that require a lot of data like Neural Networks.

Regarding co-scheduling in general, more questions are being raised, especially considering its implementation in a real-world scenario:

- It is important to come to a consensus about the payment scheme that will be employed in a cluster that supports co-scheduling. Specifically, the possibility for performance degradation due to co-location may lead users to pay a higher cost, compared to the same application's compact execution. Since performance degradation of specific applications can never be fully avoided in a co-scheduling environment, the community needs to agree on a new set of rules regarding this topic. The authors in [1] have made some interesting proposals to combat this issue.
- In this study, we concentrated only on node-sharing between two applications. In order to further improve system throughput, one approach would be to co-locate a higher number of applications on the same node, with each application utilizing the same or a different percentage of the available cores relative to the others. Of course, this would result in an even more complicated interference behavior between the applications in a node and would require different models to predict an application's performance when co-located with more than one neighbors.
- One technique to minimize contention in the shared Last-Level cache (LLC) is cache partitioning [50]. Cache partitioning divides the shared cache into regions and assigns each one to a specific application. Thus, one application does not evict the cache blocks of another, but this comes with the drawback that each application has access to a reduced portion of the LLC. This technique can be employed to reduce interference and simplify our predictions regarding performance during co-execution.
- As highlighted in [23], it is crucial for researchers to choose suitable metrics when evaluating (co-)scheduling algorithms. With co-scheduling being an emerging area of research, the need

for metrics used for evaluation will grow.

- In the exascale era, energy efficiency needs to be ensured. Techniques such as powering down nodes during periods of low system utilization and concentrating applications onto fewer nodes require further investigation.

Co-Scheduling as a technique is not limited to supercomputers. It can also be applied in cloud environments, both during the execution of applications and when it comes to consolidation of Virtual Machines in over-subscribed servers. In these and other relevant scenarios, similar trade-offs and interference patterns to those discussed in this thesis arise.

---

Εκτεταμένη Περίληψη

---

## 7.1 Επισκόπηση

Στη σημερινή εποχή, ένας τεράστιος αριθμός εφαρμογών απαιτεί πολύ μεγαλύτερη επεξεργαστική ισχύ και μνήμη από αυτήν που μπορεί να προσφέρει ένας απλός επιτραπέζιος υπολογιστής. Αυτοί οι περιορισμοί ξεπερνιούνται μέσω της χρήσης Υπολογιστικών Συστημάτων Υψηλής Επίδοσης (High Performance Computing - HPC). Τα συστήματα HPC αποτελούνται από πολλαπλούς διακομιστές ή κόμβους, συνδεδεμένους μέσω ενός δικτύου υψηλού εύρους ζώνης, ώστε να λειτουργούν ως ένας ενιαίος, ισχυρός υπολογιστής, ικανός να εκτελεί εκατομμύρια μαθηματικές πράξεις και να επεξεργάζεται τεράστιες ποσότητες δεδομένων [10]. Ως αποτέλεσμα, όλο και περισσότερες χώρες επενδύουν στους δικούς τους υπερυπολογιστές και τα συστήματα HPC έχουν αναδειχθεί σε ένα εξαιρετικά ενεργό πεδίο έρευνας.

Τα συστήματα HPC παραδοσιακά βοηθούν στην εκτέλεση διαφόρων επιστημονικών εφαρμογών και προσομοιώσεων. Ορισμένα παραδείγματα περιλαμβάνουν εφαρμογές από τους τομείς της βιοχημείας, της χημείας, της μετεωρολογίας και της υπολογιστικής δυναμικής ρευστών (CFD). Πρόσφατα, πέρα από τις βασικές επιστήμες, έχουν εμφανιστεί αρκετές νέες κατηγορίες εφαρμογών που απαιτούν την ανάθεση μεγάλου όγκου πόρων για την αποτελεσματική εκτέλεσή τους. Αυτές περιλαμβάνουν [11]:

- **Εφαρμογές Big Data:** Η εποχή του Big Data απαιτεί την επεξεργασία τεράστιων ποσοτήτων δεδομένων, τα οποία προέρχονται από διάφορες πηγές.
- **Τεχνητή Νοημοσύνη/Μηχανική Μάθηση:** Οι αλγόριθμοι AI/ML περιλαμβάνουν μια φάση εκπαίδευσης, η οποία απαιτεί πολλούς υπολογιστικούς πόρους για την γρήγορη διεκπεραίωση.



ίωσή της.

- **Επιστήμη Δεδομένων:** Σκοπός της Επιστήμης Δεδομένων είναι η εξαγωγή γνώσης και πληροφοριών από μεγάλα σύνολα δεδομένων. Περιλαμβάνει τη συλλογή και αποθήκευση δεδομένων, την επεξεργασία και ανάλυσή τους, καθώς και την οπτικοποίησή τους. Όλες αυτές οι εργασίες μπορεί να είναι ιδιαίτερα απαιτητικές χωρίς την κατάλληλη υποδομή.

### 7.1.1 Πολυεπεξεργαστικά Συστήματα

Ένας πολύ κρίσιμος παράγοντας της παράλληλης επεξεργασίας είναι οι τύποι αρχιτεκτονικών υπολογιστών που χρησιμοποιούνται. Το 1966, ο Michael J. Flynn κατηγοριοποίησε τις διαθέσιμες αρχιτεκτονικές σε τέσσερις κατηγορίες, σε αυτό που είναι γνωστό ως ‘Ταξινόμια του Flynn’ [12]:

- **Single Instruction, Single Data (SISD):** Ένας σειριακός υπολογιστής, ικανός να εκτελεί μία εντολή κάθε φορά σε ένα συγκεκριμένο σύνολο δεδομένων αποθηκευμένων σε μία μνήμη. Αυτή η αρχιτεκτονική χρησιμοποιήθηκε στους πρώτους προσωπικούς υπολογιστές (PCs), όπου δεν μπορούσε να αξιοποιηθεί η παράλληλη επεξεργασία.
- **Single Instruction, Multiple Data (SIMD):** Μία εντολή εφαρμόζεται ταυτόχρονα σε πολλαπλές ροές δεδομένων, αξιοποιώντας την παραλληλία σε επίπεδο δεδομένων. Ένα κοινό παράδειγμα χρήσης αυτής της αρχιτεκτονικής είναι οι GPUs (Μονάδες Επεξεργασίας Γραφικών).
- **Multiple Instruction, Single Data (MISD):** Πολλαπλές εντολές εφαρμόζονται στο ίδιο σύνολο δεδομένων. Αυτή η αρχιτεκτονική είναι σπάνια και έχει χρησιμοποιηθεί κυρίως για σκοπούς ανοχής σε σφάλματα.
- **Multiple Instruction, Multiple Data (MIMD):** Πολλαπλές μονάδες εκτελούν διαφορετικές εντολές σε διαφορετικές ροές δεδομένων ταυτόχρονα. Αυτή η αρχιτεκτονική είναι η πιο διαδεδομένη στα συστήματα HPC.

Τα συστήματα παράλληλης επεξεργασίας μπορούν, επίσης, να κατηγοριοποιηθούν περαιτέρω με βάση τον τρόπο που χρησιμοποιούν το σύστημα μνήμης σε **συστήματα κοινόχρηστης μνήμης** και **συστήματα κατανεμημένης μνήμης**:

- Τα συστήματα κοινόχρηστης μνήμης επιτρέπουν σε όλους τους επεξεργαστές να έχουν πρόσβαση στην ίδια μνήμη. Διακρίνονται σε UMA (όπου όλοι οι επεξεργαστές έχουν ίσο χρόνο πρόσβασης στη μνήμη) και NUMA (όπου κάθε επεξεργαστής διαθέτει τη δική του τοπική μνήμη αλλά μπορεί να προσπελάσει και τις μνήμες άλλων επεξεργαστών με μεγαλύτερη καθυστέρηση). Το OpenMP είναι το πιο γνωστό API για τέτοια συστήματα. Παρότι η κοινόχρηστη μνήμη διευκολύνει την επικοινωνία των επεξεργαστών, μπορεί να αποτελέσει σημείο συμφόρησης, ενώ η επεκτασιμότητα

ενός τέτοιου συστήματος παραμένει πρόκληση.

- Τα συστήματα κατανεμημένης μνήμης αποτελούνται από πολλαπλούς κόμβους, καθένας με τη δική του μνήμη. Η επικοινωνία γίνεται μέσω δικτύου διασύνδεσης και χαρακτηριστικά παραδείγματα τέτοιων συστημάτων αποτελούν τα MPP (Massively Parallel Processors) και οι συστοιχίες υπολογιστών (clusters), οι οποίες περιλαμβάνουν ανεξάρτητους υπολογιστικούς κόμβους. Η επικοινωνία μεταξύ των διεργασιών ενός παράλληλου προγράμματος που εκτελούνται σε διαφορετικούς κόμβους επιτυγχάνεται συνήθως μέσω MPI (Message Passing Interface). Τα συστήματα κατανεμημένης μνήμης προσφέρουν υψηλή επεκτασιμότητα, αν και απαιτούν δικτύωση υψηλού εύρους ζώνης. Οι σύγχρονοι υπερυπολογιστές συνήθως υιοθετούν ένα υβριδικό μοντέλο, συνδυάζοντας κοινόχρηστη μνήμη σε κάθε κόμβο και κατανεμημένη μνήμη μεταξύ κόμβων, για τη μεγιστοποίηση της παραλληλίας.

### 7.1.2 Προκλήσεις

Η συνεχώς αυξανόμενη ανάγκη για υψηλότερες αποδόσεις έχει γεννήσει πολλές προκλήσεις για τα σύγχρονα HPC συστήματα [11]. Συγκεκριμένα, τα μοντέρνα HPC συστήματα είναι πλέον αρκετά **ετερογενή**, αποτελούμενα από αρχιτεκτονικές που συνδυάζουν CPUs, GPUs, διαφορετικούς τύπους μνημών (DRAM, SRAM), καθώς και FPGAs και ASICs. Παράλληλα, η ραγδαία ανάπτυξη της Τεχνητής Νοημοσύνης οδήγησε στη δημιουργία εξειδικευμένου hardware, όπως AI chips και Neural Processing Units (NPUs), ενώ η σταδιακή εξέλιξη των χβαντικών επεξεργαστών θα απαιτήσει σύντομα την ενσωμάτωσή τους στα HPC συστήματα. Δεδομένης της ετερογένειας αυτών των συστημάτων, προκύπτει επίσης η ανάγκη για νέες, απλοποιημένες προγραμματιστικές προσεγγίσεις, όπως το oneAPI της Intel.

Μια άλλη μεγάλη πρόκληση είναι η **ενεργειακή αποδοτικότητα**. Τα HPC συστήματα καταναλώνουν τεράστιες ποσότητες ενέργειας, αλλά μόνο ένα μικρό μέρος χρησιμοποιείται για την εκτέλεση των εντολών. Το μεγαλύτερο μέρος χάνεται σε διαρροές και προσβάσεις μνήμης, ειδικά σε συστήματα με πολύπλοκες ιεραρχίες μνήμης [13]. Αυτό καθιστά αναγκαία τη βελτίωση των μηχανισμών προγραμματισμού ώστε να διατηρείται η τοπικότητα των δεδομένων αλλά και την εκτέλεση των υπολογισμών αν αυτό είναι δυνατό απευθείας στην μνήμη με την τεχνική Processing In Memory (PIM). Τέλος, η **ανθεκτικότητα** σε σφάλματα είναι κρίσιμη, καθώς οι σύγχρονες HPC αρχιτεκτονικές περιλαμβάνουν πολλούς κόμβους και είναι πολύπλοκες, αυξάνοντας τη συχνότητα σφαλμάτων.

Στην παρούσα εργασία, θα επικεντρωθούμε στη βελτίωση της επίδοσης των εφαρμογών και του συστήματος μέσω **συνδρομολόγησης (co-scheduling)** εφαρμογών σε υπερυπολογιστικούς κόμβους. Θα μελετήσουμε μεθόδους πρόβλεψης της συμπεριφοράς των εφαρμογών και θα αξιολογήσουμε την συνδρομολόγηση ως στρατηγική βελτίωσης της επίδοσης, εξετάζοντας πιθανά trade-offs σε αυτό το πιο σύνθετο πλαίσιο.

## 7.2 Υποδομή και Benchmarks

Τα πειράματα αυτής της εργασίας πραγματοποιήθηκαν στον υπερυπολογιστή ARIS της GRNET S.A. [4]. Ο ARIS εγκαταστάθηκε το 2015 και περιλαμβάνει 532 υπολογιστικούς κόμβους, οργανωμένους σε διαφορετικά partitions. Διαθέτει αρχιτεκτονική x86-64, λειτουργικό σύστημα Redhat/Centos 6.7 και οι κόμβοι του συνδέονται μέσω δικτύου Infiniband FDR (56Gb/s) σε τοπολογία fat tree. Οι χρήστες συνδέονται μέσω SSH και υποβάλλουν τις εργασίες τους στον SLURM Workload Manager. Για τα πειράματά μας, χρησιμοποιήθηκαν οι thin nodes, με κάθε κόμβο να διαθέτει δύο sockets των 10 πυρήνων. Οι μετρήσεις έγιναν πολλαπλές φορές σε διαστήματα 10-15 λεπτών, και κρατήθηκε η διάμεσος των αποτελεσμάτων για ελαχιστοποίηση των ακραίων τιμών.

Ο υπερυπολογιστής ARIS χρησιμοποιεί τον SLURM Workload Manager [5] για τη διαχείριση και προγραμματισμό των υποβληθέντων εργασιών. Ο SLURM κατανέμει πόρους σε χρήστες για συγκεκριμένο χρονικό διάστημα, είτε με αποκλειστική είτε με μη αποκλειστική πρόσβαση. Στα πειράματα αυτής της εργασίας χρησιμοποιήθηκε αποκλειστική πρόσβαση για την ακριβή μέτρηση της επίδοσης, χωρίς παρεμβολές από άλλες εργασίες.

Το SLURM διαχειρίζεται μια ουρά αναμονής, παρακολουθεί την εκτέλεση των εργασιών και τις τερματίζει όταν ολοκληρωθούν ή όταν λήξει ο προβλεπόμενος από τον χρήστη χρόνος εκτέλεσης (wallclock time). Ο κύριος daemon, ονόματι slurmetld διαχειρίζεται το σύστημα, αποθηκεύει πληροφορίες, παρακολουθεί την κατάσταση των κόμβων και προγραμματίζει τις εργασίες. Σε κάθε κόμβο, ο slurmd daemon εκκινεί, παρακολουθεί και ολοκληρώνει τις εργασίες. Ο slurmdbd διαχειρίζεται βάσεις δεδομένων με πληροφορίες των πόρων και αρχεία καταγραφής.

Οι χρήστες αλληλεπιδρούν με το SLURM μέσω εντολών command line, όπως εκείνες που φαίνονται στον παρακάτω πίνακα:

Εντολή	Επεξήγηση
srun	υποβολή εργασίας στο σύστημα
sbatch	υποβολή ενός σεναρίου (script) που μπορεί να περιλαμβάνει πολλές εργασίες
squeue	αναφορά της κατάστασης όλων των εργασιών του συστήματος
scancel	ακύρωση της εκτέλεσης μιας εργασίας

**Πίνακας 7.1.** Τέσσερις σημαντικές εντολές του SLURM

Ο προγραμματισμός σε μια αρχιτεκτονική κατανεμημένης μνήμης είναι πιο περίπλοκος από ότι σε συστήματα κοινής μνήμης, καθώς απαιτείται μια διεπαφή ανταλλαγής μηνυμάτων. Το MPI (Message Passing Interface) είναι η πιο ευρέως χρησιμοποιούμενη βιβλιοθήκη για τον σκοπό αυτόν. Το MPI προσφέρει δύο τύπους επικοινωνίας: **σημείο-προς-σημείο (point-to-point)** και **συλλογική (collective)**.

Στην σημείο-προς-σημείο επικοινωνία, δύο διεργασίες επικοινωνούν απευθείας μέσω εντολών όπως

MPI\_Send και MPI\_Recv. Στην συλλογική επικοινωνία, η επικοινωνία μπορεί να είναι Ένας προς Πολλούς (π.χ. MPI\_Bcast, MPI\_Scatter), Πολλοί προς Έναν (π.χ. MPI\_Gather, MPI\_Reduce) ή Πολλοί προς Πολλούς (π.χ. MPI\_Alltoall, MPI\_AllReduce, MPI\_Allgather). Οι παραπάνω εντολές είναι blocking, δηλαδή επιστρέφουν μόνο όταν οι buffers επικοινωνίας μπορούν να ξαναχρησιμοποιηθούν με ασφάλεια. Το MPI προσφέρει επίσης non-blocking εκδοχές των εντολών (π.χ. MPI\_Isend, MPI\_Irecv), οι οποίες επιστρέφουν αμέσως αλλά απαιτούν από τον προγραμματιστή να ελέγξει την ολοκλήρωση της επικοινωνίας πριν επαναχρησιμοποιήσει τα δεδομένα. Αυτές οι non-blocking λειτουργίες επιτρέπουν επικάλυψη επικοινωνίας και υπολογισμού, καθώς και αποφυγή deadlocks.

Τέλος, υπάρχουν και εντολές συγχρονισμού, που εισάγουν χρόνο αναμονής ώστε να συντονίζονται τις διεργασίες (π.χ. MPI\_Barrier, MPI\_Wait, MPI\_Waitall). Άεργος χρόνος αναμονής εισάγεται και από διάφορες blocking εντολές, εφόσον το MPI αναγκάζεται να περιμένει τη στιγμή που και ο αποστολέας και ο παραλήπτης είναι έτοιμοι για να διεξαχθεί η επικοινωνία. Τέτοιων ειδών αναμονές μειώνουν τον διαθέσιμο παραλληλισμό και επηρεάζουν αρνητικά την κλιμακωσιμότητα και την επιτάχυνση (speedup) της εφαρμογής.

Κατά την διεξαγωγή των πειραμάτων σε αυτήν την εργασία χρησιμοποιήθηκαν κυρίως τα **NAS Parallel Benchmarks (NPB)**. Τα NAS Parallel Benchmarks [14] προέρχονται από εφαρμογές υπολογιστικής ρευστοδυναμικής (CFD) και χρησιμοποιούνται ευρέως για μοντελοποίηση επίδοσης και προσομοιώσεις HPC. Κάθε benchmark υποστηρίζει διαφορετικά μεγέθη προβλημάτων, που ονομάζονται κλάσεις. Η ονομασία κάθε MPI benchmark ακολουθεί τη δομή:

(όνομα).<κλάση>.<αριθμός\_διεργασιών>

Για παράδειγμα, το mg.E.128 αντιστοιχεί στο benchmark MG, με κλάση E, που εκτελείται με 128 διεργασίες. Η σουίτα NPB περιλαμβάνει οκτώ benchmarks, εκ των οποίων πέντε είναι υπολογιστικοί πυρήνες (EP, MG, CG, FT, IS) και τρία είναι ψευδο-εφαρμογές (LU, SP, BT).

Επιπλέον, σε αυτή την εργασία χρησιμοποιήθηκαν και benchmarks από τη σουίτα **SPEChpc 2021**, η οποία καλύπτει ένα ευρύ φάσμα επιστημονικών πεδίων και είναι αντιπροσωπευτική των σύγχρονων HPC φορτίων εργασίας. Όπως και τα NAS benchmarks, έτσι και τα SPEC benchmarks υποστηρίζουν διαφορετικά μεγέθη φορτίου εργασίας: tiny, small, medium και large. Η ονομασία κάθε benchmark ακολουθεί τη δομή:

<αναγνωριστικό>.<όνομα>.<μέγεθος>.<αριθμός\_διεργασιών>

Το αναγνωριστικό είναι ένας μοναδικός αριθμός για κάθε benchmark, με πρόθεμα που δηλώνει το μέγεθος του φορτίου (5 για tiny, 6 για small, 7 για medium και 8 για large). Για παράδειγμα, το 619.clvleaf\_s.1024 αναφέρεται στο benchmark clvleaf, με μικρό μέγεθος φορτίου (small), που εκτελείται με 1024 διεργασίες.

Τέλος, για την εύρεση του μέγιστου θεωρητικού bandwidth μνήμης του κάθε κόμβου του υπερυπολογιστή ARIS αξιοποιήθηκε το STREAM benchmark [15], το οποίο μετράει τον χρόνο εκτέλεσης τεσσάρων υπολογιστικών πυρήνων που εκτελούν διαφορετικές πράξεις πάνω σε μεγάλους πίνακες που δεν χωράνε στην cache του μηχανήματος. Για τον ARIS, το μέγιστο θεωρητικό bandwidth μνήμης προέκυψε ίσο με 69177.3 MB/s για τον πυρήνα Scale.

## 7.3 Profiling

Το profiling χρησιμοποιείται με σκοπό την μέτρηση της συχνότητας εμφάνισης συγκεκριμένων γεγονότων (π.χ. εκτέλεση μιας συνάρτησης, ενός βρόχου ή μιας γραμμής κώδικα) ή την καταγραφή μετρικών του συστήματος (π.χ. συνολικός αριθμός εκτελεσμένων εντολών, αστοχίες cache) κατά την εκτέλεση ενός προγράμματος. Αυτή η τεχνική είναι ιδιαίτερα χρήσιμη, καθώς παρέχει πολύτιμες πληροφορίες για τη συμπεριφορά ενός προγράμματος και βοηθά στη βελτιστοποίηση του κώδικα και την ανάλυση της απόδοσής του.

Το profiling αρκετές φορές πραγματοποιείται offline, δηλαδή τα αποτελέσματα των μετρήσεων του είναι διαθέσιμα μετά από μια πλήρη εκτέλεση του προγράμματος. Σε περιπτώσεις εύρεσης πιθανών σημείων που απαιτούν βελτιστοποίηση, αλλά και σε περιπτώσεις όπου απαιτούνται πληροφορίες για ένα πρόγραμμα που πρόκειται να τρέξει πολλές φορές (π.χ. πρόβλεψη μετεωρολογικών δεδομένων με χρήση HPC), μια τέτοια προσέγγιση είναι επαρκής. Ωστόσο, υπάρχουν και αρκετά σενάρια χρήσης του profiling που τα αποτελέσματα ή μια εκτίμηση αυτών απαιτείται ενόσω η εφαρμογή ακόμα τρέχει. Χαρακτηριστικό τέτοιο παράδειγμα αποτελούν τα συστήματα δυναμικής μεταγλώττισης (JIT compilers). Ως εκ τούτου, έχουν αναπτυχθεί και συστήματα online profiling, που με δεδομένες τις μετρήσεις τους ως ένα συγκεκριμένο σημείο της εκτέλεσης του προγράμματος προβλέπουν την μετέπειτα συμπεριφορά του. Σε κάθε περίπτωση, οι profilers πρέπει να μεριμνούν να μην εισάγουν πολλά περισσότερα χρονικά ή χωρικά overheads στην εκτέλεση των προγραμμάτων, κάτι που συχνά επιτυγχάνεται με χρήση επιλεκτικού (selective) profiling από τον προγραμματιστή.

Οι profilers χρησιμοποιούν διάφορες μεθόδους για τη συλλογή πληροφοριών σχετικά με την εκτέλεση μιας εφαρμογής. Οι πιο διαδεδομένες είναι οι εξής:

- **Instrumentation:** Απαιτεί τροποποίηση του πηγαίου ή εκτελέσιμου κώδικα ώστε να προστεθούν εντολές που συλλέγουν δεδομένα κατά την εκτέλεση. Αυτό μπορεί να γίνει αυτόματα από τον compiler, μέσω linking με pre-instrumented βιβλιοθήκες ή χειροκίνητα από τον προγραμματιστή [16]. Αν και προσφέρει λεπτομερή δεδομένα, μπορεί να έχει σημαντικό overhead, ειδικά αν εφαρμοστεί εκτενώς και όχι επιλεκτικά.
- **Sampling:** Συλλέγει δείγματα δεδομένων σε ταχτά χρονικά διαστήματα μέσω interrupt routines του λειτουργικού συστήματος [17]. Αν και δεν παρέχει πλήρη εικόνα όπως το instrumentation, έχει πολύ μικρότερο overhead και είναι χρήσιμο για online profiling.

- **Hardware Performance Counters:** Πρόκειται για ειδικούς καταχωρητές εντός του επεξεργαστή που μετρούν hardware/software events. Προσφέρουν χαμηλό overhead αφού τους διαχειρίζεται απευθείας ο επεξεργαστής και όχι το λειτουργικό σύστημα και δεν απαιτούν επιπλέον υλικό αφού συμπεριλαμβάνονται εντός των περισσότερων μοντέρνων επεξεργαστών. Ωστόσο, δεδομένου του μικρού πλήθους τους, επιτρέπουν την ταυτόχρονη μέτρηση μικρού αριθμού γεγονότων. Χρησιμοποιούνται ευρέως για ανάλυση απόδοσης εφαρμογών, αλλά και σε τομείς όπως η ασφάλεια συστημάτων [18] [19].

Παρότι το profiling παρέχει χρήσιμες πληροφορίες, δεν αποτυπώνει χρονικές σχέσεις μεταξύ γεγονότων, κάτι σημαντικό για παράλληλες εφαρμογές. Σε αυτές τις περιπτώσεις, πιο κατάλληλο είναι το tracing, το οποίο προσφέρει λεπτομερή χρονολογική καταγραφή των γεγονότων, με κόστος τη δημιουργία μεγαλύτερου overhead και την ανάγκη για μεγάλη χωρητικότητα αποθήκευσης για τα δεδομένα μετρήσεων που προκύπτουν. Επιπλέον, υπάρχει και η δυνατότητα παθητικού profiling μέσω της παρακολούθησης ολόκληρου του συστήματος όπου εκτελείται μια εφαρμογή. Με μια τέτοια προσέγγιση, το profiling δεν παρεμβαίνει καθόλου στην εκτέλεση της εφαρμογής αλλά εξάγει και λιγότερο αξιόπιστα συμπεράσματα.

Σε αυτή την εργασία, το profiling χρησιμοποιείται για τη γενική κατηγοριοποίηση των πειραματικών εφαρμογών, χωρίς την ανάγκη εξαιρετικά λεπτομερούς ανάλυσης όπως αυτής που απαιτείται για βελτιστοποίηση κώδικα. Συνεπώς, εστιάζουμε σε:

- **Hardware Performance Counters:** Μετρώνται μέσω του εργαλείου perf του Linux, βοηθώντας στον διαχωρισμό των εφαρμογών σε compute-intensive ή memory-intensive.
- **Στατιστικά Επικοινωνίας:** Σε σενάρια HPC απαιτείται επίσης profiling της πραγματοποιούμενης επικοινωνίας μεταξύ των διεργασιών. Τα απαραίτητα δεδομένα συλλέγονται μέσω του εργαλείου mpiP, επιτρέποντας την αναγνώριση εφαρμογών με έντονη επικοινωνία (communication-intensive) και τη διάκριση μεταξύ διαφορετικών μοτίβων επικοινωνίας.

### 7.3.1 perf

Το perf [20] είναι μια εντολή του Linux που χρησιμοποιείται για lightweight profiling και βασίζεται στη διεπαφή perf\_events. Όταν γίνεται profiling μετρικών απόδοσης με το perf, δεν απαιτείται re-compilation ή re-linking του εκτελέσιμου, καθώς η εντολή perf μπορεί απλά να τοποθετηθεί μπροστά από το εκτελέσιμο και όλες οι μετρήσεις πραγματοποιούνται κατά την εκτέλεση. Η πιο σημαντική εντολή που σχετίζεται με το perf είναι το **perf stat**. Αυτή η εντολή χρησιμοποιείται για να εκτυπώνει στην standard output, μετά την εκτέλεση ενός προγράμματος, τους συνολικούς αριθμούς εμφανίσεων των καθορισμένων PMU (Performance Monitoring Unit) γεγονότων. Τα γεγονότα υλικού της PMU, όπως αστοχίες κρυφής μνήμης (cache misses), FLOPS ή εντολές φόρτωσης (load instructions), αντιστοιχίζονται από το λογισμικό σε performance counters (που είναι φυσικοί καταχωρητές, όπως περιγράφηκε προηγουμένως) ώστε να μπορούν να παρακολουθούνται. Η εντολή perf stat μπορεί



να διατυπωθεί με δύο διαφορετικούς τρόπους: χρησιμοποιώντας ένα σύνολο προκαθορισμένων γεγονότων ή χρησιμοποιώντας δεκαεξαδικούς κωδικούς. Επιπλέον, ξεκινώντας από τη μικροαρχιτεκτονική Intel Nehalem, οι επεξεργαστές άρχισαν να υποστηρίζουν offcore events [21]. Αυτά τα γεγονότα παρακολουθούν τις αλληλεπιδράσεις μνήμης που συμβαίνουν εκτός του πυρήνα.

```
perf stat -e cycles,instructions,cache-misses [executable details]
perf stat -e r1a8 [executable details]
```

Όπως αναφέρθηκε προηγουμένως, κάθε CPU έχει περιορισμένο αριθμό performance counters. Ως αποτέλεσμα, αν ο χρήστης καθορίσει στην εντολή perf stat περισσότερα γεγονότα υλικού από τους διαθέσιμους μετρητές, ο πυρήνας θα πρέπει να χρησιμοποιήσει πολυπλεξία (multiplexing). Αυτό επιτρέπει στον πυρήνα να εναλλάσσει γεγονότα μέσα και έξω από τους performance counters, που σημαίνει ότι δεν μετρώνται όλα τα γεγονότα συνεχώς καθ' όλη τη διάρκεια της εκτέλεσης. Στο τέλος της εκτέλεσης, το perf κλιμακώνει τα αποτελέσματα για κάθε γεγονός ώστε να παρέχει μια εκτίμηση των συνολικών μετρήσεων, σαν να είχαν χρησιμοποιηθεί οι performance counters καθ' όλη τη διάρκεια εκτέλεσης της εφαρμογής, για κάθε ένα από τα μετρούμενα γεγονότα. Επιπλέον, υπάρχουν και άλλες εντολές του perf που χρησιμοποιούνται ευρέως. Οι πιο σημαντικές περιγράφονται συνοπτικά στον Πίνακα 7.2.

Εντολή	Επεξήγηση
perf record	Εκτελεί δειγματοληψία βασισμένη σε διακοπές (interrupt-based sampling) για την καταγραφή γεγονότων προς μεταγενέστερη ανάλυση και τα αποθηκεύει σε δυαδικό αρχείο
perf report	Διαβάζει το αρχείο που παράγεται από την εντολή perf record και δημιουργεί ένα συνοπτικό προφίλ εκτέλεσης
perf annotate	Μετράει εμφανίσεις γεγονότων ανά γραμμή πηγαίου κώδικα ή κώδικα assembly
perf top	Παρουσιάζει ζωντανά τις μετρήσεις των γεγονότων, παρόμοια με το εργαλείο top του Linux

**Πίνακας 7.2.** Τέσσερις σημαντικές εντολές του perf

### 7.3.2 mpiP

Το mpiP είναι ένα ελαφρύ εργαλείο για profiling επικοινωνίας σε MPI εφαρμογές [22]. Χρησιμοποιείται για τη συλλογή στατιστικών πληροφοριών σχετικά με τις κλήσεις MPI, αξιοποιώντας τη διεπαφή PMPI ώστε να καταγράφει τότε ξεκινά και τότε ολοκληρώνεται μια εντολή [8]. Υποστηρίζει τόσο link-time όσο και run-time instrumentation, χωρίς να απαιτείται επαναμεταγλώττιση της εφαρμογής. Όπως και άλλα εργαλεία profiling που αναφέρθηκαν προηγουμένως, επιτρέπει επίσης το επιλεκτικό profiling μόνο σε συγκεκριμένα, προκαθορισμένα από τον χρήστη, τμήματα της εφαρμογής.

## 7.4 Μετρικές και είδη εφαρμογών

Πιθανά schedules ή co-schedules σε HPC και οι αλγόριθμοι που χρησιμοποιούνται για την παραγωγή τους θα πρέπει να αξιολογούνται βάσει συγκεκριμένων μετρικών. Όπως τονίζεται στο [23], η εύρεση ενός κατάλληλου συνόλου μετρικών αξιολόγησης για αυτόν τον σκοπό δεν είναι απλή υπόθεση. Επιπλέον, η ανάγκη κατηγοριοποίησης εφαρμογών ώστε να περιγραφεί καλύτερα η συμπεριφορά τους αναδεικνύει τη σημασία της αναγνώρισης διακριτών τύπων εφαρμογών.

Στον χώρο του HPC, κάθε μετρική δεν έχει την ίδια σημασία για όλους τους stakeholders. Οι βασικοί stakeholders είναι οι διαχειριστές συστήματος και οι χρήστες που υποβάλλουν εργασίες. Προφανώς, οι διαχειριστές ενδιαφέρονται περισσότερο για τη συνολική επίδοση και αποδοτικότητα του συστήματος, σε αντίθεση με τους χρήστες, οι οποίοι θέλουν οι δικές τους εργασίες να ολοκληρώνονται όσο το δυνατόν νωρίτερα. Αυτή η αντίθεση οδηγεί συχνά σε προβλήματα πολυκριτηριακής βελτιστοποίησης κατά τον σχεδιασμό ενός αλγορίθμου χρονοπρογραμματισμού, καθώς πρέπει να λαμβάνουμε υπόψη τόσο την **ικανοποίηση του συστήματος όσο και των χρηστών**. Οι βασικές μετρικές που θα χρησιμοποιήσουμε σε αυτή την εργασία είναι οι εξής:

- **Makespan** : Αναφέρεται στον συνολικό χρόνο που απαιτείται για την ολοκλήρωση ενός συνόλου εργασιών, ξεκινώντας από την υποβολή της πρώτης εργασίας έως την ολοκλήρωση της τελευταίας. Αποτελεί ένδειξη του throughput του supercomputer και αφορά αποκλειστικά τους διαχειριστές συστήματος.
- **Utilization** : Αναφέρεται στο ποσοστό του συστήματος που βρίσκεται σε χρήση κατά τη διάρκεια ενός πειράματος. Ωστόσο, όπως αναφέρεται στο [23], πρέπει να είμαστε προσεκτικοί ώστε να μετράμε μόνο το steady-state utilization όταν χρησιμοποιούμε αυτήν τη μετρική για την αξιολόγηση ενός αλγορίθμου. Η χαμηλή χρήση του συστήματος είναι αναπόφευκτη κατά τις φάσεις αρχικοποίησης και αδειάσματος του συστήματος, καθώς αυτό σταδιακά γεμίζει και αδειάζει. Επιπλέον, τα HPC συστήματα δεν χρησιμοποιούνται στο έπακρο καθ' όλη τη διάρκεια της ημέρας. Όταν η χρήση του συστήματος είναι χαμηλή (π.χ. κατά τη διάρκεια της νύχτας), αυτή η μετρική δεν είναι αξιόπιστη για την αξιολόγηση πιθανών αλγορίθμων χρονοπρογραμματισμού. Όπως και το makespan, η μετρική utilization αφορά την απόδοση του συστήματος και ενδιαφέρει μόνο τους διαχειριστές.
- **Turnaround time** : Αναφέρεται στο άθροισμα του χρόνου αναμονής και του χρόνου εκτέλεσης μιας συγκεκριμένης εργασίας. Ως εκ τούτου, είναι μια μετρική που αφορά τον χρήστη που έχει υποβάλει την εργασία στο σύστημα.
- **Job Speedup/Slowdown** : Σε ένα σενάριο co-scheduling, όταν δύο εργασίες συνυπάρχουν στον ίδιο κόμβο, οι χρόνοι εκτέλεσής τους διαφέρουν από αυτούς που θα είχαν αν εκτελούνταν μεμονωμένα (compact mode). Σε αυτό το πλαίσιο, η επιτάχυνση (speedup) μιας εφαρμογής



δίνεται από τον εξής τύπο:

$$\text{JobSpeedup} = \frac{\text{Compact Execution Time}}{\text{Co-located Execution Time}}$$

Αν η τιμή του JobSpeedup είναι μικρότερη του 1, τότε λέμε ότι η εφαρμογή παρουσιάζει slowdown όταν συνυπάρχει με έναν συγκεκριμένο γείτονα. Σε ένα σενάριο scheduling, το slowdown μπορεί να περιλαμβάνει και τον χρόνο αναμονής μιας εφαρμογής στην ουρά. Στο πλαίσιο του co-scheduling, προς το παρόν εστιάζουμε αποκλειστικά στη μεταβολή του χρόνου εκτέλεσης μιας εφαρμογής για τον υπολογισμό του slowdown. Η μετρική Job Speedup/Slowdown έχει σημασία για την ικανοποίηση του χρήστη, αλλά δίνει και μια εικόνα κατά πόσο ο αλγόριθμος χρονοπρογραμματισμού μας αξιοποιεί σωστά τις εφαρμογές της ουράς αναμονής.

Στην περίπτωση μετρικών όπως το turnaround time και το speedup/slowdown, υπάρχουν και σταθμισμένες εκδοχές τους που, εκτός από τη χρονική διάσταση, λαμβάνουν υπόψη και τον χώρο που καταλαμβάνει μια εφαρμογή [23].

Επιπλέον, υπάρχουν τρεις βασικές κατηγορίες MPI εφαρμογών σε ένα περιβάλλον HPC: **compute-bound** εφαρμογές, που εκτελούν κυρίως υπολογισμούς και αξιοποιούν τις CPU των κόμβων, **memory-bound** εφαρμογές, που εκτελούν κυρίως προσβάσεις στη μνήμη και των οποίων ο βασικός περιοριστικός παράγοντας είναι οι πόροι μνήμης, και **communication-bound** εφαρμογές, που αφιερώνουν τον περισσότερο χρόνο τους σε κλήσεις MPI. Σε ένα περιβάλλον κοινόχρηστης μνήμης, οι compute-bound εφαρμογές κλιμακώνουν καλά (scalable) όταν προστίθεται περισσότερη παραλληλία μέσω επιπλέον πυρήνων CPU. Αντίθετα, οι memory-bound εφαρμογές έχουν περιορισμένη κλιμάκωση λόγω των πεπερασμένων πόρων μνήμης. Αυτό αποδεικνύεται επίσης στο [24] για την περίπτωση του EP (ενός έντονα compute-bound benchmark) και του MG (ενός έντονα memory-bound benchmark) από τη σουίτα NAS.

Σε ένα περιβάλλον co-scheduling, μια εφαρμογή συνήθως κατανέμεται σε διπλάσιο αριθμό κόμβων σε σχέση με την απομονωμένη εκτέλεση, αλλά καταλαμβάνει τους μισούς πυρήνες σε κάθε κόμβο (**spread mode**). Έτσι, ο συνολικός αριθμός πυρήνων που χρησιμοποιούνται για την εκτέλεση της εφαρμογής παραμένει ίδιος με την απομονωμένη εκτέλεση (**compact mode**), πράγμα που σημαίνει ότι οι compute-bound εφαρμογές πιθανότατα δεν θα βιώσουν ούτε speedup ούτε slowdown σε αυτές τις συνθήκες. Ωστόσο, οι memory-bound εφαρμογές μπορούν να βελτιώσουν δραστικά την απόδοσή τους, καθώς η κατανομή σε περισσότερους κόμβους εξασφαλίζει περισσότερους πόρους μνήμης και υψηλότερο memory bandwidth. Οι communication-bound εφαρμογές μπορεί να εμφανίσουν πιο απρόβλεπτη συμπεριφορά όταν εφαρμόζεται το παραπάνω σχήμα co-scheduling. Αυτό συμβαίνει επειδή, αφενός, οι επιπλέον κόμβοι παρέχουν επιπλέον memory buffers και θύρες δικτύου, αλλά αφετέρου, εισάγουν επιπλέον ανάγκη για επικοινωνία μεταξύ των καινούργιων κόμβων. Όπως φαίνεται στη συνέχεια, όμως,

οι communication-bound εφαρμογές συνήθως παρουσιάζουν speedups στο spread mode σε σύγκριση με την εκτέλεσή τους σε compact mode.

## 7.5 Παραδοσιακή Χρονοδρομολόγηση

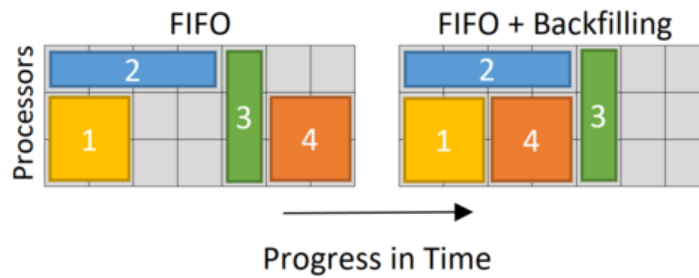
Παραδοσιακά, οι πόροι κατανέμονται στις εργασίες μέσω ενός batch scheduler, ο οποίος είναι υπεύθυνος για τον χρονοπρογραμματισμό των εργασιών στους κόμβους με δίκαιο τρόπο [25]. Επιπλέον, διαχειρίζεται την πρόσβαση σε διαφορετικές ουρές ενός υπερυπολογιστή. Για να μπορεί να λαμβάνει σύνθετες αποφάσεις χρονοπρογραμματισμού, ο batch scheduler χρειάζεται να διαθέτει μια εκτίμηση της διάρκειας εκτέλεσης κάθε υποβαλλόμενης εργασίας. Για αυτόν τον λόγο, οι χρήστες καλούνται να παρέχουν μια εκτίμηση του χρόνου εκτέλεσης της εργασίας τους, η οποία αναφέρεται συνήθως ως wallclock. Δεδομένης αυτής της εκτίμησης ή μιας προηγμένης πρόβλεψης του χρόνου εκτέλεσης κάθε εργασίας, ο batch scheduler χρησιμοποιεί συγκεκριμένες ευρετικές συναρτήσεις, όπως για παράδειγμα First Come First Served (FCFS) ή Shortest Job/Area First (SJF/SAF), ώστε να δρομολογήσει τις εργασίες στο σύστημα. Πρόσφατες προσεγγίσεις ενσωματώνουν επίσης τεχνικές μηχανικής μάθησης για τον χρονοπρογραμματισμό των εισερχόμενων εργασιών.

Το πιο σημαντικό πρόβλημα που αντιμετωπίζεται κατά τη χρήση παραδοσιακών ευρετικών για χρονοπρογραμματισμό είναι η πιθανή υποχρησιμοποίηση του συστήματος, καθώς η εργασία με τη μεγαλύτερη προτεραιότητα σε μια δεδομένη χρονική περίοδο ενδέχεται να μην μπορεί να εκτελεστεί άμεσα λόγω έλλειψης διαθέσιμων πόρων και να χρειαστεί να περιμένει έως ότου ελευθερωθούν πόροι. Η μέθοδος που χρησιμοποιείται στην πράξη για την αντιμετώπιση αυτού του περιορισμού είναι η χρήση του **backfilling**, το οποίο στοχεύει να καλύψει ορισμένα από τα κενά που δημιουργούνται στο χρονοπρόγραμμα με εργασίες χαμηλότερης προτεραιότητας, χωρίς να καθυστερήσει την εκτέλεση των εργασιών υψηλότερης προτεραιότητας. Το Σχήμα 7.1 οπτικοποιεί την βασική αρχή λειτουργίας του backfilling. Οι πιο βασικές κατηγορίες backfilling είναι οι εξής [26]:

- **Conservative Backfill** : Μια εργασία που χωράει στα δημιουργηθέντα κενά μετακινείται προς τα εμπρός στην ουρά μόνο εάν δεν καθυστερεί **καμία** από τις εργασίες υψηλότερης προτεραιότητας στην ουρά.
- **Aggressive (EASY) Backfill** : Μια εργασία που χωράει στα δημιουργηθέντα κενά μετακινείται προς τα εμπρός στην ουρά μόνο εάν δεν καθυστερεί την εργασία που βρίσκεται αυτή τη στιγμή **στην κεφαλή της ουράς**. Σε αντίθεση με το conservative backfill, αυτή η τεχνική ενδέχεται να επηρεάσει τη δικαιοσύνη στον χρονοπρογραμματισμό.

## 7.6 Συνδρομολόγηση και τεχνικές

Η συνδρομολόγηση εφαρμογών (co-scheduling) είναι μια τεχνική που έχει ως στόχο την αύξηση της ρυθμαπόδοσης του συστήματος. Η εκτέλεση μιας εργασίας σε spread mode οδηγεί τις περισσότερες



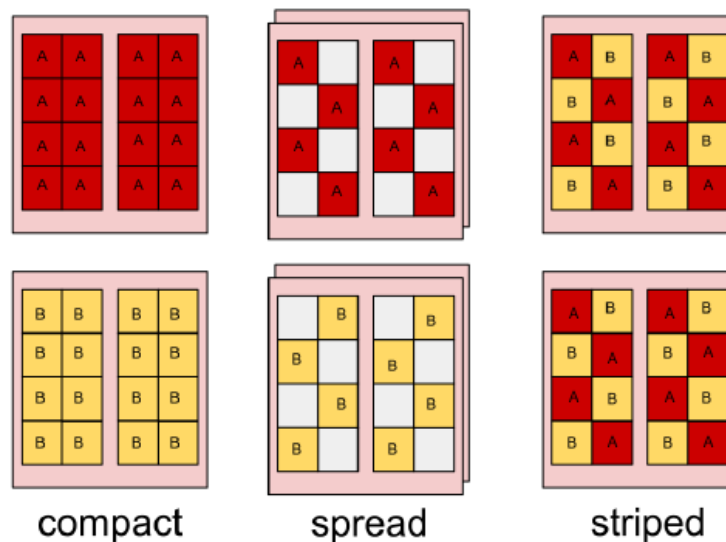
Σχήμα 7.1. Οπτικοποίηση του backfilling [42]

φορές σε αυξημένη απόδοση. Στην εργασία [1], οι συγγραφείς αποδίδουν αυτό το φαινόμενο στο γεγονός ότι οι διεργασίες της ίδιας MPI εργασίας εκτελούν σχεδόν πανομοιότυπες εργασίες και, ως εκ τούτου, προσπαθούν να έχουν ταυτόχρονη πρόσβαση στους ίδιους πόρους. Μειώνοντας τον αριθμό των διεργασιών της ίδιας εργασίας σε έναν συγκεκριμένο κόμβο, μειώνεται ο ανταγωνισμός για τους εν λόγω πόρους (π.χ. caches, εύρος ζώνης μνήμης, network ports), ενώ το προτεινόμενο σχήμα μπορεί επίσης να επωφεληθεί από τους επιπλέον πόρους για την αύξηση της απόδοσης. Ωστόσο, το απλό spread mode συνεπάγεται ότι οι μισοί πυρήνες σε κάθε κόμβο παραμένουν ανενεργοί, οδηγώντας έτσι σε σημαντική υποχρησιμοποίηση του συστήματος. Επιπλέον, η εκτέλεση μιας εργασίας σε spread mode συνεπάγεται υψηλότερο κόστος για τους χρήστες, καθώς σε περιβάλλοντα HPC, οι χρήστες χρεώνονται βάσει των ωρών χρήσης των πυρήνων που καταναλώνουν (core hours). Ως αποτέλεσμα, αυτή η λειτουργία χρησιμοποιείται κυρίως σε καταστάσεις urgent computing, όπου υπάρχει άμεση ανάγκη για αυξημένη απόδοση.

Για την καθημερινή χρήση σε production συστήματα, τα προτεινόμενα σχήματα συνδρομολόγησης (co-scheduling) στην έρευνα περιλαμβάνουν την τοποθέτηση δύο ή περισσότερων εργασιών στον ίδιο κόμβο (co-location). Το πιο διαδεδομένο και αποδοτικότερο σχήμα σύμφωνα με το [1] ονομάζεται **job striping**. Ας υποθέσουμε ότι κάθε κόμβος σε έναν υπερυπολογιστή περιέχει δύο sockets και κάθε socket περιέχει 8 πυρήνες (σύνολο 16 πυρήνες ανά κόμβο). Χρησιμοποιώντας το σχήμα job striping, οι μισοί πυρήνες κάθε socket (στην περίπτωση μας τέσσερις) θα ανατεθούν στην πρώτη εργασία, ενώ οι υπόλοιποι μισοί θα ανατεθούν στη δεύτερη εργασία. Ο Πίνακας 7.3 περιγράφει τους προαναφερθέντες τρόπους ανάθεσης πόρων για μια MPI εργασία που χρησιμοποιεί 16 διεργασίες και εκτελείται στο προαναφερθέν σύστημα. Στη συνέχεια, το Σχήμα 7.2 παρέχει μια οπτικοποίηση των τριών τρόπων ανάθεσης.

Τρόπος ανάθεσης	Απαιτούμενοι κόμβοι	Χρησιμοποίηση Πυρήνων ανά κόμβο
Compact	1	16 πυρήνες (χρησιμοποιούνται όλοι οι πυρήνες)
Spread	2	8 πυρήνες (8 πυρήνες ανενεργοί)
Job Striping	2	8 πυρήνες (οι υπόλοιποι χρησιμοποιούνται από άλλη εργασία)

Πίνακας 7.3. Σύγκριση των τρόπων ανάθεσης πόρων



Σχήμα 7.2. Οπτικοποίηση των τριών τρόπων ανάθεσης πόρων [1]

Το job striping δεν είναι η μοναδική εφαρμόσιμη προσέγγιση συν-τοποθέτησης. Μία άλλη προσέγγιση είναι η εκχώρηση ενός ολόκληρου socket σε κάθε εργασία εντός ενός κόμβου (**socket-exclusive ανάθεση**). Το κύριο πλεονέκτημα της job striping είναι ότι, εκχωρώντας ετερογενείς διεργασίες στο ίδιο socket, μειώνουμε τον ανταγωνισμό στην Last-Level Cache (LLC) του συγκεκριμένου socket σε σύγκριση με την προσέγγιση socket-exclusive, όπου πολλαπλές παρόμοιες διεργασίες ασκούν παρόμοιες απαιτήσεις στην LLC. Ωστόσο, οι διεργασίες από το πρώτο socket θα πρέπει να επικοινωνούν με διεργασίες της ίδιας εργασίας από το δεύτερο socket, δημιουργώντας έτσι ένα επιπλέον κόστος επικοινωνίας, το οποίο αποφεύγεται στην προσέγγιση socket-exclusive. Συνεπώς, δημιουργείται ένα trade-off μεταξύ του ανταγωνισμού στην LLC και στο εύρος ζώνης μνήμης, καθώς και στη χωρική τοπικότητα του υλικού, και η κατάλληλη προσέγγιση εξαρτάται από το εκάστοτε σύστημα. Για τα πειράματα που διεξήχθησαν σε αυτή την εργασία, χρησιμοποιήθηκε η προσέγγιση job striping. Οι communication-bound εφαρμογές επίσης επωφελούνται από τη συν-τοποθέτηση, εφόσον διαθέτουν διαφορετικά επικοινωνιακά μοτίβα [8]. Δεδομένου ότι υπάρχει μεγάλη πιθανότητα δύο διαφορετικές εργασίες να έχουν διαφορετικά επικοινωνιακά μοτίβα, κάτι τέτοιο είναι εφικτό.

Το co-scheduling έχει αποδειχθεί ότι οδηγεί σε συνολική βελτίωση του makespan ενός συνόλου εφαρμογών καθώς και σε speedups για την πλειονότητα των τύπων εφαρμογών που συμμετέχουν. Ωστόσο, μια συγκεκριμένη μεθοδολογία ή αλγόριθμος για την εύρεση του βέλτιστου co-schedule δεδομένου ενός συγκεκριμένου συνόλου εφαρμογών παραμένει ένα ανοιχτό πρόβλημα. Στο [27], οι συγγραφείς απέδειξαν ότι η εύρεση του βέλτιστου co-schedule σε ένα σύστημα με περισσότερους από δύο πυρήνες ανά chip είναι ένα NP-complete πρόβλημα. Άλλες μελέτες έχουν επικεντρωθεί στην ασυμπτωτική προσέγγιση του βέλτιστου co-schedule. Σε κάθε περίπτωση, ορισμένες εργασίες αναπόφευκτα θα επιβραδυνθούν, γεγονός που δημιουργεί ένα ζήτημα δικαιοσύνης. Αυτό οδηγεί στο προαναφερθέν πρόβλημα βελτιστοποίησης πολλαπλών κριτηρίων, όπου πρέπει ταυτόχρονα να διατηρούμε ικανοποιημένους τόσο το σύστημα όσο και τον χρήστη όσον αφορά τη συνολική απόδοση και την απόδοση κάθε εργασίας ξεχωριστά. Για να αναπτύξουμε έναν εξελιγμένο αλγόριθμο λήψης αποφάσεων για το

co-scheduling των εργασιών, ο οποίος λαμβάνει υπόψη τόσο την απόδοση του συστήματος όσο και την ικανοποίηση του χρήστη, αντί να βασιζόμαστε σε τυχαία συνδρομολόγηση, είναι απαραίτητο να κατανοήσουμε τα συγκεκριμένα χαρακτηριστικά των εφαρμογών στο σύνολο εφαρμογών μας. Χωρίζουμε τις μεθόδους που μπορούν να αξιοποιηθούν προς αυτόν τον σκοπό σε δύο ευρείες κατηγορίες: **tag-based** μοντέλα και **pairwise** μοντέλα.

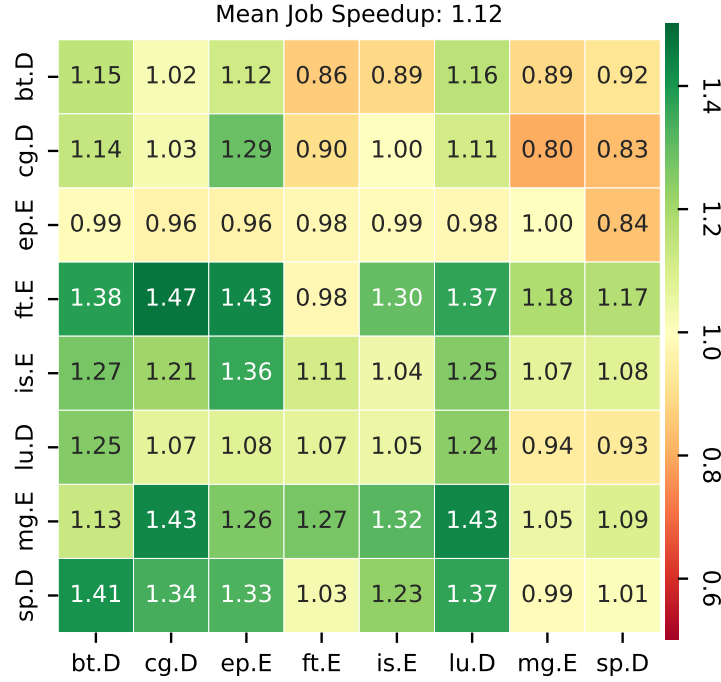
### 7.6.1 Tag-based μοντέλα

Τα tag-based μοντέλα στοχεύουν στην ανάκτηση ενός αυτόνομου χαρακτηρισμού για όλες τις εφαρμογές στο σύνολο εφαρμογών ο οποίος αργότερα θα αξιοποιηθεί στη λήψη τεκμηριωμένων αποφάσεων για το co-scheduling, λαμβάνοντας υπόψη το προφίλ κάθε εφαρμογής. Αυτός ο χαρακτηρισμός μπορεί να είναι είτε **resource-centric** είτε **co-location-centric**. Στην προσέγγιση resource-centric, χαρακτηρίζουμε τις εφαρμογές σύμφωνα με τα μοτίβα κατανάλωσης πόρων τους. Ο χαρακτηρισμός μπορεί να κυμαίνεται από μια απλή ετικέτα (π.χ. memory-bound, compute-bound ή communication-bound) έως μια λεπτομερή ανάλυση της χρήσης των πόρων από την εφαρμογή, ώστε να αποκτήσουμε ένα αναλυτικό προφίλ. Έχοντας αποκτήσει ένα προφίλ για κάθε εφαρμογή στο σύνολο εφαρμογών μας, μπορούμε στη συνέχεια να λάβουμε είτε στατικές είτε δυναμικές αποφάσεις σχετικά με το πού πρέπει να τοποθετηθεί κάθε εφαρμογή στο σύστημά μας. Στην προσέγγιση co-location-centric, κάθε εφαρμογή πρέπει να χαρακτηριστεί ως είτε co-location friendly είτε co-location unfriendly. Αυτή η προσέγγιση μας επιτρέπει είτε να χρησιμοποιήσουμε αυτές τις πληροφορίες για να καθορίσουμε ποια ζευγάρια εφαρμογών θα συν-τοποθετηθούν είτε να επιλέξουμε ένα υποσύνολο εφαρμογών για μια ουρά co-scheduling, με τις υπόλοιπες εφαρμογές να εκχωρούνται στην κανονική ουρά χρονοπρογραμματισμού. Οι προσεγγίσεις resource-centric απαιτούν μία εκτέλεση για κάθε εφαρμογή σε compact mode, ενώ ταυτόχρονα εκτελείται profiling, ώστε να προσδιοριστούν τα μοτίβα κατανάλωσης πόρων της. Αυτό αποτελεί ένα απλό βήμα προεπεξεργασίας στις περισσότερες περιπτώσεις HPC, καθώς η πλειονότητα των εφαρμογών HPC εκτελούνται πολλές φορές από τους χρήστες (π.χ. μοντέλα πρόβλεψης καιρού) και συνεπώς το profiling μόνο της πρώτης εκτέλεσης προσθέτει ελάχιστο επιπλέον φόρτο. Αντιθέτως, οι προσεγγίσεις co-location-centric απαιτούν έναν πιο εξελιγμένο τρόπο χαρακτηρισμού των εφαρμογών.

### 7.6.2 Pairwise μοντέλα

Τα **Pairwise μοντέλα** απαιτούν γνώση ή εκτίμηση της συμπεριφοράς co-scheduling για όλους τους πιθανούς συνδυασμούς εφαρμογών εντός του συνόλου εφαρμογών. Τα heatmaps, όπως αυτό που παρουσιάζεται στο Σχήμα 7.3, προσφέρουν μια οπτικοποίηση αυτών των δεδομένων, όπου κάθε γραμμή αντιπροσωπεύει τα speedups της αντίστοιχης εφαρμογής όταν συν-τοποθετείται με τις εφαρμογές που αντιπροσωπεύονται σε κάθε στήλη.

Η βασική ιδέα είναι ότι το heatmap των εφαρμογών του συνόλου εφαρμογών μας θα χρησιμοποιηθεί από έναν αλγόριθμο βελτιστοποίησης για τη λήψη αποφάσεων co-scheduling. Αυτό μπορεί να γίνει είτε με την εξαντλητική συνεκτέλεση όλων των πιθανών συνδυασμών και τη μέτρηση της απόδοσής τους σε σύγκριση με το compact mode (μια μη πρακτική μέθοδος σε ένα περιβάλλον production), είτε με



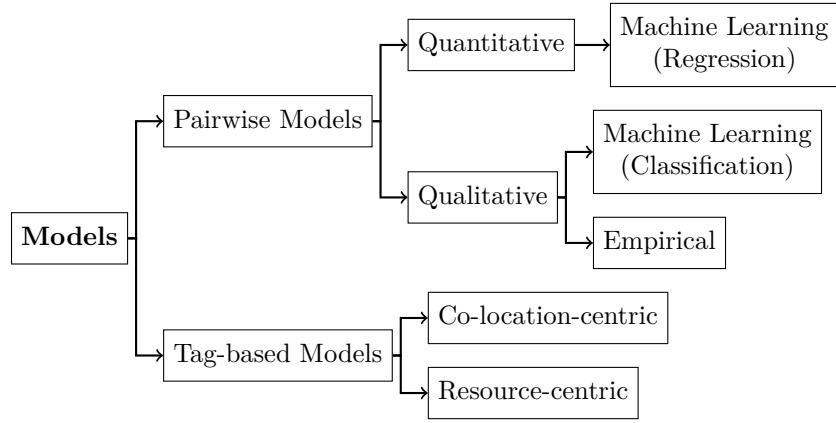
**Σχήμα 7.3.** Heatmap των speedups από την συνεκτέλεση οκτώ NPB benchmarks με 256 processes το καθένα στον υπερυπολογιστή Marconi

την ποιοτική ή ποσοτική πρόβλεψη του heatmap. Μια **ποιοτική** προσέγγιση θα μπορούσε να χρησιμοποιήσει ετικέτες (π.χ. good, bad, stationary) για να χαρακτηριστεί η συμπεριφορά κάθε εφαρμογής όταν συνεκτελείται με τις υπόλοιπες. Αυτές οι ετικέτες μπορούν να παραχθούν είτε **εμπειρικά** είτε χρησιμοποιώντας **αλγορίθμους Machine Learning classification**. Αντίθετα, μια **ποσοτική** προσέγγιση στοχεύει στην πρόβλεψη της ακριβούς αριθμητικής τιμής των speedups στο heatmap. Αυτό μπορεί να επιτευχθεί χρησιμοποιώντας **αλγορίθμους Machine Learning regression**. Στα επόμενα κεφάλαια, θα προτείνουμε και θα διερευνήσουμε μεθόδους για την απόκτηση πληροφοριών σχετικά με την απόδοση των εφαρμογών σε λειτουργία συνεκτέλεσης, ώστε να παραχθούν τα απαραίτητα δεδομένα που θα χρησιμοποιηθούν από προηγμένους αλγορίθμους co-scheduling για τη λήψη τεκμηριωμένων προβλέψεων. Η παραπάνω ταξινόμηση απεικονίζεται συνοπτικά στο Σχήμα 7.4.

## 7.7 Μεθοδολογία profiling

Ο κοινός παρονομαστής όλων των μοντέλων που παρουσιάζονται σε αυτήν την εργασία είναι η χρήση performance counters και communication events. Όπως αναφέρθηκε προηγουμένως, πραγματοποιήσαμε profiling στα benchmarks που χρησιμοποιήθηκαν στα πειράματά μας χρησιμοποιώντας το **perf** και το **mpiP**, προκειμένου να αποκαλύψουμε τη συμπεριφορά τους όσον αφορά τις υπολογιστικές πράξεις, τη μνήμη και την επικοινωνία. Ο Πίνακας 7.4 παρουσιάζει τις μετρικές στόχους μας, καθώς και τους performance counters και τα communication events που χρησιμοποιήσαμε για να τις υπολογίσουμε. Περιορίσαμε το profiling σε μικρό αριθμό hardware events, καθώς όταν ο αριθμός των μετρούμενων γεγονότων υπερβαίνει τους διαθέσιμους φυσικούς performance counters σε έναν





**Σχήμα 7.4.** Ταξινόμηση των μοντέλων *performance analysis* για χρήση σε αλγορίθμους συνδρομολόγησης

επεξεργαστή, το **perf** χρησιμοποιεί χρονική πολυπλεξία, γεγονός που μπορεί να μειώσει την ακρίβεια των μετρήσεων.

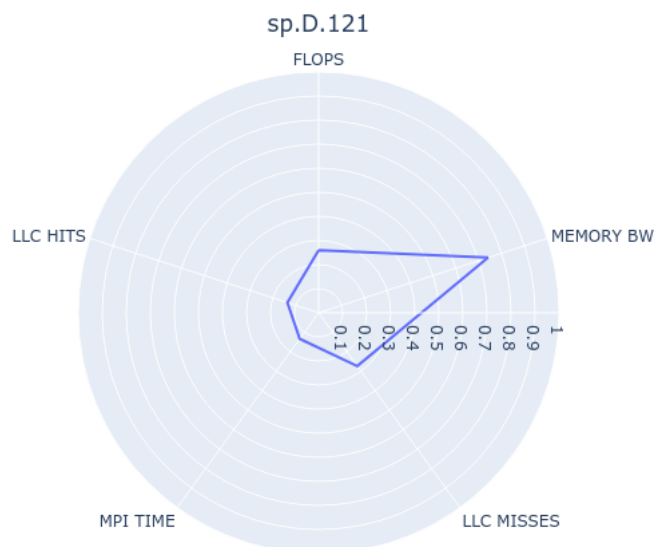
Μετρική	Εργαλείο	Γεγονότα
FLOPS	perf	SSE_FP_PACKED_DOUBLE, SSE_SCALAR_DOUBLE
Memory Bandwidth	perf	OFFCORE_RESPONSE_0:LLC_MISS_LOCAL OFFCORE_RESPONSE_0:LLC_MISS_REMOTE
LLC hits/misses	perf	LLC_MISSES, LLC_REFERENCES
Total MPI Time	mpiP	MPI% (Aggregate MPI Time)
MPI Directive Time	mpiP	Callsite Time Statistics

**Πίνακας 7.4.** Μετρικές και σχετικά μετρούμενα γεγονότα

Έπειτα, χρησιμοποιήσαμε τα γεγονότα που μετρήσαμε προκειμένου να υπολογίσουμε τις σύνθετες μετρικές στόχους. Για παράδειγμα, για το memory bandwidth χρησιμοποιήθηκε ο ακόλουθος τύπος ώστε να υπολογιστεί σε επίπεδο κόμβου:

$$\text{Memory Bandwidth} = \frac{\sum_{i=1}^{np} ((\text{LLC\_MISS\_LOCAL}_i + \text{LLC\_MISS\_REMOTE}_i) \times \text{Cache Line Size})}{\text{Number of Nodes} \cdot \text{Execution Time}}$$

όπου το  $np$  συμβολίζει τον αριθμό των processes του κάθε benchmark. Τα FLOPS, Memory Bandwidth, MPI Time, LLC Hits and LLC Misses που χαρακτηρίζουν μια εφαρμογή ομαδοποιούνται και οπτικοποιούνται με χρήση **spider plots**. Το Σχήμα 7.5 δείχνει ένα παράδειγμα ενός spider plot για το SP benchmark που τρέχει χρησιμοποιώντας 121 processes.



Σχήμα 7.5. Spider plot για το SP benchmark

## 7.8 Tag-based μοντέλα

Τα Tag-based μοντέλα στοχεύουν στο να παρέχουν στους Co-Schedulers έναν συγκεκριμένο χαρακτηρισμό κάθε εφαρμογής στο σύνολο εφαρμογών. Αυτός ο χαρακτηρισμός μπορεί να ποικίλει από μια βασική ετικέτα έως μια σύνθετη, που χρησιμοποιεί ένα ολοκληρωμένο σύνολο μετρήσεων που συλλέγονται μέσω performance counters. Η προσέγγισή μας είναι απλή, χωρίς να απαιτεί πολύπλοκη μοντελοποίηση. Συγκεκριμένα περιλαμβάνει τη συλλογή δεδομένων σχετικά με τη μνήμη, το πλήθος υπολογιστικών πράξεων και την επικοινωνία, και τη χρήση αυτών ως χαρακτηρισμό της εφαρμογής με σκοπό την καθοδήγηση της λήψης αποφάσεων σε σενάρια συνεκτέλεσης. Ενδεικτικά, χρησιμοποιούμε μετρικές που σχετίζονται με την υπολογιστική ένταση (FLOPS), τη χρήση εύρους ζώνης μνήμης (MEMORY BW), τα μοτίβα πρόσβασης στην κρυφή μνήμη (LLC hits και misses), καθώς και την ένταση επικοινωνίας (MPI time), όλα εκ των οποίων στη συνέχεια κανονικοποιούνται. Οι μετρικές αυτές επιλέχθηκαν επειδή επηρεάζουν τη συμπεριφορά μιας εφαρμογής σε ένα σενάριο συνεκτέλεσης, αποκαλύπτοντας τη συμπεριφορά της όταν υπάρχει ανταγωνισμός για κοινόχρηστους πόρους.

Παρουσιάζουμε μια προσέγγιση για την αξιοποίηση του παραγόμενου χαρακτηρισμού των εφαρμογών με σκοπό τη βελτίωση του μέσου speedup των εργασιών σε ένα σενάριο συνεκτέλεσης. Επεκτείνουμε τη συλλογιστική που παρουσιάζεται στο [28], προσπαθώντας να δημιουργήσουμε ζεύγη εφαρμογών των οποίων τα spider plots και, κατά συνέπεια, τα μοτίβα κατανάλωσης πόρων, διαφέρουν όσο το δυνατόν περισσότερο. Εξερευνούμε ένα απλό και στατικό σενάριο, όπου διαθέτουμε μια ομάδα 300 εφαρμογών και δημιουργούμε 150 ζεύγη. Στην αρχική baseline περίπτωση, οι εφαρμογές συν-τοποθετούνται διαδοχικά (η πρώτη με τη δεύτερη, η τρίτη με την τέταρτη κ.ο.κ.). Στη συνέχεια, υλοποιούμε μια πιο προηγμένη προσέγγιση, όπου αξιοποιούμε την προαναφερθείσα συλλογιστική. Συγκεκριμένα, για κάθε πιθανό ζεύγος εφαρμογών στο σύνολο εφαρμογών, υπολογίζουμε ένα score που υποδεικνύει τον βαθμό επικάλυψης μεταξύ των spider plots των δύο εφαρμογών. Όσο μεγαλύτερο είναι αυτό το score για ένα συγκεκριμένο ζεύγος, τόσο πιο διαφορετικές είναι οι εφαρμογές μεταξύ τους, γεγονός που



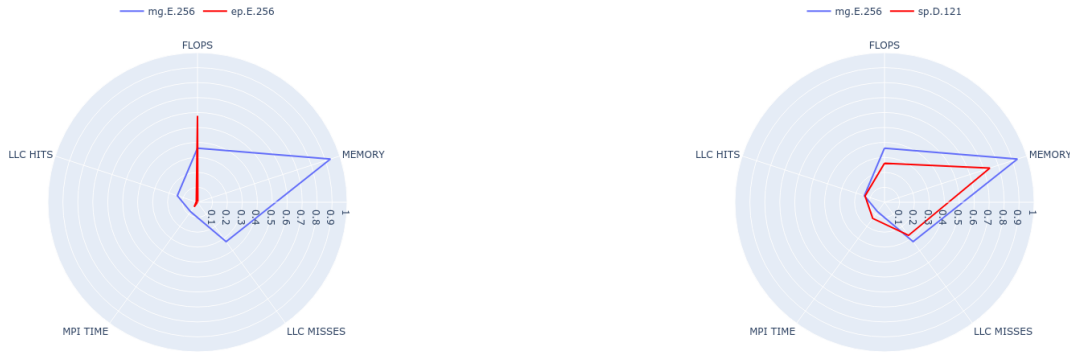
υποδηλώνει ένα καλύτερο αποτέλεσμα συνεκτέλεσης. Το Σχήμα 7.6α' παρουσιάζει δύο εφαρμογές με ελάχιστη επικάλυψη στα spider plots τους, ενώ το Σχήμα 7.6β' παρουσιάζει δύο εφαρμογές με αρκετή επικάλυψη στα spider plots τους. Αυτή η βαθμολογία υπολογίζεται ως εξής:

$$\text{FLOPSdiff} = |\text{FLOPS}_1 - \text{FLOPS}_2|$$

$$\text{MEMORYdiff} = |\text{Memory}_1 - \text{Memory}_2|$$

$$\text{COMMdiff} = |\text{MPI\_time}_1 - \text{MPI\_time}_2|$$

$$\text{score} = \sqrt{(\text{FLOPSdiff})^2 + (\text{MEMORYdiff})^2 + (\text{COMMdiff})^2}$$



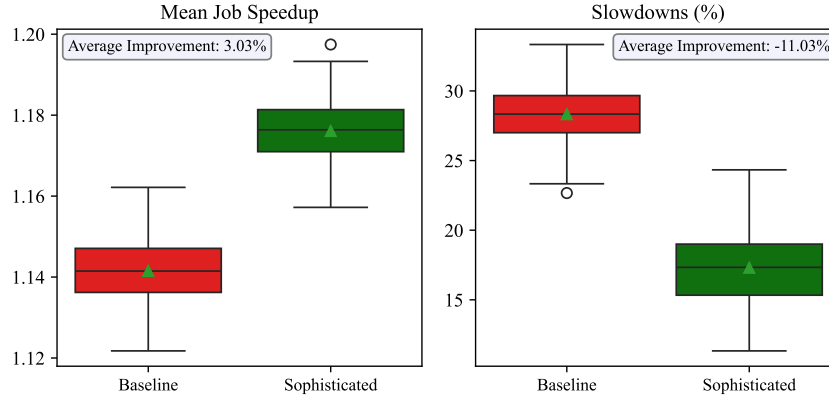
(α') Παράδειγμα μιας καλής συν-τοποθέτησης

(β') Παράδειγμα μιας κακής συν-τοποθέτησης

**Σχήμα 7.6.** Spider Co-Plots για ένα σενάριο καλής και ένα σενάριο κακής συν-τοποθέτησης

Αυτή η απλοϊκή προσέγγιση συν-τοποθετεί τις εφαρμογές με έναν greedy τρόπο. Συγκεκριμένα, καθώς κινούμαστε σειριακά μέσα στο σύνολο εφαρμογών, συν-τοποθετούμε κάθε εφαρμογή με εκείνη την εφαρμογή η οποία οδηγεί στο μέγιστο score. Αυτή η προσέγγιση ευνοεί σημαντικά τις πρώτες εφαρμογές της ουράς, καθώς αυτές θα συν-τοποθετηθούν με τους βέλτιστους γείτονες. Ωστόσο, οδηγεί σε σημαντική βελτίωση του μέσου speedup των εργασιών και σε αξιοσημείωτη μείωση του ποσοστού των εφαρμογών που παρουσιάζουν slowdown κατά τη συνεκτέλεση. Το Σχήμα 7.7 καταδεικνύει τα οφέλη της προσέγγισής μας σε σύγκριση με την baseline περίπτωση που περιγράφηκε προηγουμένως. Παρατηρούμε βελτίωση του μέσου speedup των εργασιών κατά 3.03% και μείωση του μέσου ποσοστού των εργασιών που παρουσιάζουν slowdown κατά 11.03%.

Τα resource-centric tag-based μοντέλα είναι αρκετά απλά και απαιτούν μόνο δεδομένα που μπορούν να συλλεχθούν εύκολα από monitoring units. Είναι σημαντικό να σημειωθεί ότι αυτή η τεχνική απαιτεί μόνο μία εκτέλεση για κάθε εφαρμογή σε compact mode, ενώ ταυτόχρονα γίνεται profiling αυτής ώστε να προσδιοριστούν τα μοτίβα κατανάλωσης πόρων. Αυτό συνιστά μια πολύ απλούστερη προσέγγιση με πολύ λιγότερη προεπεξεργασία και πολυπλοκότητα, σε σύγκριση με τα Pairwise μοντέλα που θα συζητηθούν στη συνέχεια.



**Σχήμα 7.7.** Τα πλεονεκτήματα της *resource-centric tag-based* προσέγγισής μας σε σύγκριση με την *baseline* περίπτωση

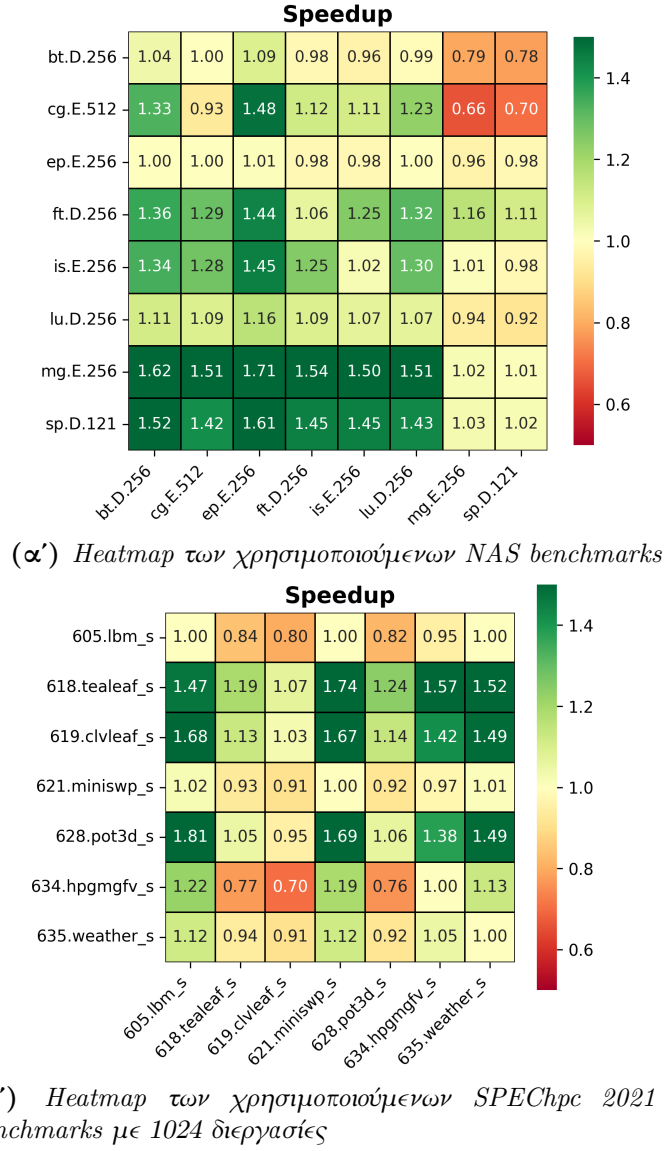
## 7.9 Pairwise μοντέλα : Εμπειρικός classifier

Το πρώτο μοντέλο που συζητούμε στην κατηγορία των pairwise μοντέλων είναι ένα εμπειρικό ποιοτικό μοντέλο που στοχεύει στον χαρακτηρισμό κάθε πιθανού συνδυασμού του συνόλου εφαρμογών ως *good*, *stationary* ή *bad*. Το μοντέλο αποτελείται από ένα σύνολο κανόνων που χρησιμοποιούνται για τη δημιουργία ενός εμπειρικού δέντρου αποφάσεων, το οποίο λαμβάνει ως είσοδο δύο εφαρμογές, A και B, και προβλέπει το *speedup* της εφαρμογής A όταν συνεκτελείται με την εφαρμογή B. Η βασική ιδέα πίσω από αυτό το μοντέλο είναι η χρήση των δεδομένων χρήσης των πόρων για την πρόβλεψη της αναμενόμενης συμπεριφοράς με έναν επεξηγήσιμο τρόπο. Για την εξαγωγή των κανόνων που χρησιμοποιούνται στην κατασκευή του δέντρου αποφάσεων, αναλύσαμε δύο heatmaps από τον υπερυπολογιστή ARIS, για τα NAS και SPEChpc 2021 benchmarks, όπως φαίνεται στα Σχήματα 7.8α' και 7.8β' αντίστοιχα.

Χρησιμοποιούμε ξανά τα profiles που δημιουργήθηκαν με τη βοήθεια των *perf* και *mpiP* για τη δημιουργία spider plots για όλα τα benchmarks, επιτρέποντάς μας να τα κατηγοριοποιήσουμε αποτελεσματικά. Τα spider plots παρουσιάζονται στα Σχήματα 4.7 και 4.10. Μεταξύ των τριών κύριων μετρικών—FLOPS, MPI Time και Memory Bandwidth—εκείνη με τη μεγαλύτερη τιμή καθορίζει αν το benchmark είναι *compute-intensive*, *communication-intensive* ή *memory-intensive* αντίστοιχα. Τέλος, τα spider plots απεικονίζουν επίσης τα Last-Level Cache hits και misses, τα οποία θα είναι χρήσιμα σε ορισμένες περιπτώσεις. Παρακάτω παραθέτουμε τους κανόνες που συνθέσαμε μέσω παρατήρησης των heatmaps και των spider plots, για καθεμία εκ των τριών πιθανών τύπων εφαρμογών.

### 7.9.1 Compute-bound εφαρμογές

Σε αυτή την κατηγορία ανήκουν τα benchmarks EP, BT και LU από τα NAS benchmarks. Με δεδομένη την συμπεριφορά τους όπως παρατηρείται στο heatmap, αλλά και στα spider plots εξάγουμε τους εξής κανόνες:



**Σχήμα 7.8.** Heatmaps των NAS και SPEChpc 2021 benchmarks στον υπερυπολογιστή ARIS

#### Εμπειρικοί Κανόνες: Compute-bound εφαρμογές

- Η απόδοση των ισχυρά compute-bound εφαρμογών (EP) μένει στάσιμη όταν συν-τοποθετούνται με οποιαδήποτε άλλη εφαρμογή
- Οι compute-bound εφαρμογές με σημαντικό αριθμό από προσβάσεις μνήμης (BT) μένουν στάσιμες σε όλες τις περιπτώσεις, εκτός από όταν συν-τοποθετούνται με memory-bound εφαρμογές. Σε αυτή την περίπτωση, παρουσιάζουν slowdown.
- Οι compute-bound εφαρμογές με αρκετή επικοινωνία (LU) μπορούν να παρουσιάσουν speedup όταν συν-τοποθετούνται με non-communication-bound ή non-memory-bound benchmarks.
- Οι compute-bound εφαρμογές αποτελούν καλούς γείτονες για όλους τους τύπους εφαρμογών καθώς κάνουν αμελητέα χρήση των κοινόχρηστων πόρων του κόμβου.

### 7.9.2 Memory-bound εφαρμογές

Οι πιο ισχυρά memory-bound εφαρμογές στο heatmap των NAS benchmarks (Σχήμα 7.8α') είναι το MG και το SP. Επιπλέον ισχυρά memory-bound δείχνει να είναι το CG. Ωστόσο, όπως φαίνεται από το heatmap, η συμπεριφορά του είναι αρκετά διαφορετική από εκείνη των δύο προαναφερθέντων benchmarks. Αυτό οφείλεται στο γεγονός ότι εν αντιθέσει με τα MG, SP, το CG εκτελεί ακανόνιστες (irregular) προσβάσεις μνήμης και άρα δεν αξιοποιεί καλά το memory bandwidth. Τέτοιες εφαρμογές συνήθως εμφανίζουν πολλά cache misses και συνεπώς μέσω αυτής της μετρικής καθίσταται δυνατή η αναγνώρισή τους. Με βάση τα δεδομένα που έχουμε, συνάγουμε τους ακόλουθους κανόνες:

#### Εμπειρικοί Κανόνες: Memory-bound εφαρμογές

- Οι memory-bound εφαρμογές μπορούν να επιτύχουν πολύ υψηλά speedups όταν συν-τοποθετούνται με compute-bound ή communication-bound εφαρμογές.
- Όταν συν-τοποθετούνται με άλλες memory-bound εφαρμογές, η απόδοσή τους μειώνεται. Η πιο ισχυρά memory-bound εφαρμογή από τις δύο στο ζευγάρι τείνει να έχει καλύτερη απόδοση.
- Οι memory-bound εφαρμογές είναι κακοί γείτονες για άλλες εφαρμογές λόγω των υψηλών τους απαιτήσεων στους πόρους μνήμης του κόμβου.
- Οι memory-bound εφαρμογές που εκτελούν ακανόνιστες (irregular) προσβάσεις μνήμης δεν μπορούν να επιτύχουν μεγάλα speedups εξαιτίας του γεγονότος ότι υποχρησιμοποιούν το memory bandwidth. Για τον ίδιο λόγο, αποτελούν καλούς γείτονες για άλλες εφαρμογές. Το ήδη υψηλό τους memory latency δυσχεραίνεται ακόμη περισσότερο όταν συν-εκτελούνται με memory-bound εφαρμογές, οδηγώντας σε σημαντικά slowdowns.

### 7.9.3 Communication-bound εφαρμογές

Οι δύο εναπομείνουσες εφαρμογές από τα NAS benchmarks (IS, FT) είναι communication-bound. Ενώ αυτά τα δύο benchmarks παρουσιάζουν παρόμοια συμπεριφορά, τα δύο communication-bound benchmarks της σουίτας SPEChpc 2021 (621.miniswp\_s, 635.weather\_s) παρουσιάζουν διαφορετική, στάσιμη συμπεριφορά. Ως εκ τούτου, καθίσταται απαραίτητη μια πιο ενδελεχής ανάλυση της επικοινωνίας των εφαρμογών ώστε να συμπεράνουμε τι είδους MPI εντολές εκτελούνται σε κάθε περίπτωση. Η ανάλυση αυτή παρατίθεται στον Πίνακα 7.5

Με βάση τα δεδομένα που έχουμε στην κατοχή μας συνάγουμε τους εξής εμπειρικούς κανόνες:

635.weather_s.1024	621.miniswp_s.1024
Callsites: 19456	Callsites: 15360
Waitall: 99.90%	Recv: 85.24%
Isend: 0.04%	Send: 14.49%
Irecv: 0.06%	Barrier: 0.24%
Barrier: 0.00%	Comm_split: 0.02%
	Comm_free: 0.00%
	Allreduce: 0.00%
ft.D.256	
Callsites: 3840	
Alltoall: 97.16%	
Reduce: 1.70%	
Barrier: 0.88%	
Comm_split: 0.19%	
Bcast: 0.07%	

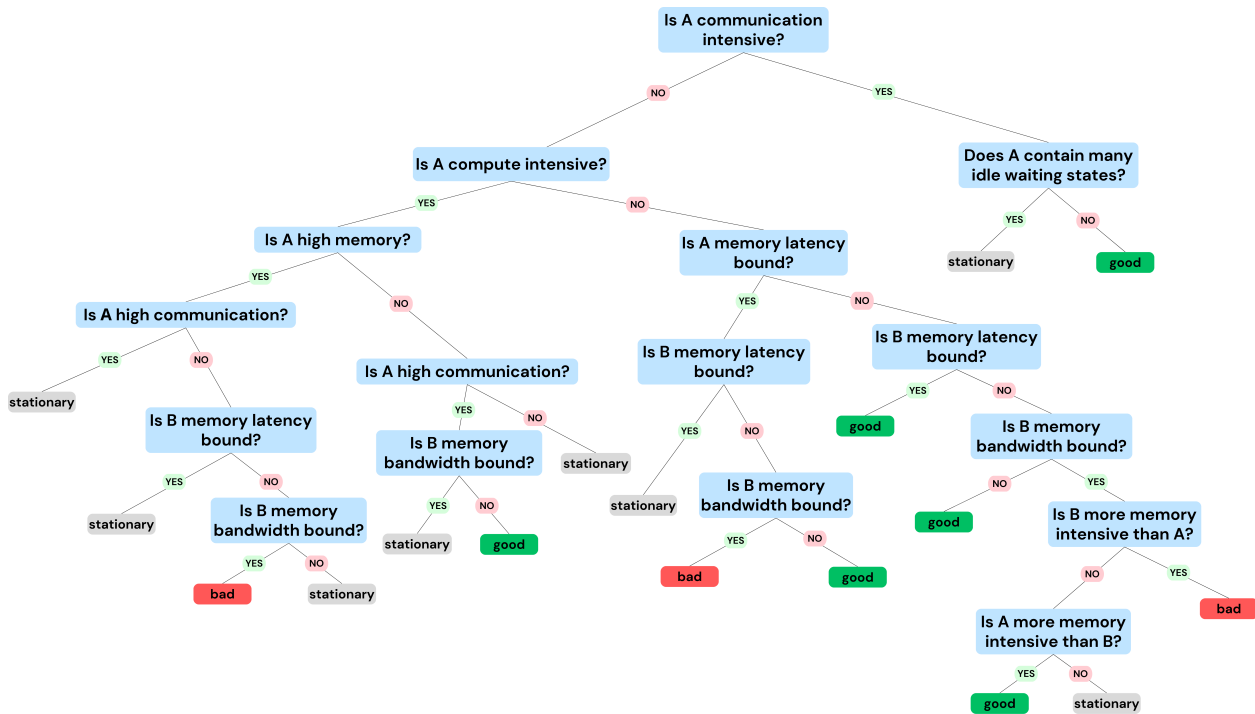
**Πίνακας 7.5.** Ποσοστιαία κατανομή του συνολικού χρόνου MPI που δαπανάται σε κάθε εντολή MPI

#### Εμπειρικοί Κανόνες: Communication-bound εφαρμογές

- Οι communication-bound εφαρμογές που περιέχουν κυρίως χρήσιμη επικοινωνία (FT, IS) συνήθως παρουσιάζουν speedup κατά τη διάρκεια της συνεκτέλεσής τους με κάθε τύπο εφαρμογών.
- Αντιθέτως, οι communication-bound εφαρμογές των οποίων ο χρόνος επικοινωνίας αποτελείται κυρίως από χρόνους αναμονής (635.weather\_s, 621.miniswp\_s) εμφανίζουν στάσιμη συμπεριφορά κατά την συνεκτέλεση.
- Οι communication-bound εφαρμογές είναι συνήθως καλοί γείτονες για άλλες εφαρμογές, αφού δεν μονοπωλούν τους κοινόχρηστους πόρους και τα μοτίβα επικοινωνίας διαφορετικών εφαρμογών είναι συνήθως διαφορετικά μεταξύ τους.
- Η συμπεριφορά των communication-bound εφαρμογών μπορεί όμως να γίνει απρόβλεπτη όταν μια δουλειά μοιράζεται σε αρκετά μεγάλο αριθμό κόμβων ή σε περιόδους υψηλής συμφόρησης του δικτύου διασύνδεσης του υπερυπολογιστικού συστήματος.

#### 7.9.4 Αποτίμηση προβλέψεων

Χρησιμοποιώντας τους προαναφερθέντες κανόνες, κατασκευάσαμε ένα εμπειρικό δέντρο αποφάσεων, το οποίο παρουσιάζεται στο Σχήμα 7.9. Όπως αναφέρθηκε, αυτό το μοντέλο λαμβάνει ως είσοδο δύο εφαρμογές, A και B, και προβλέπει τον χαρακτηρισμό του speedup της εφαρμογής A όταν συνεκτελείται με την B. Οι τρεις δυνατές κατηγορίες είναι: good, stationary και bad. Ορίζουμε αυστηρά και χαλαρά όρια speedup για αυτές τις τρεις κατηγοριοποιήσεις, όπως φαίνεται στον Πίνακα 7.6. Χαρακτηρίζουμε τις προβλέψεις που συμμορφώνονται με τα αυστηρά όρια speedup ως «Πλήρως Σωστές» και εκείνες που ακολουθούν τα χαλαρά όρια speedup ως «Επαρκώς Σωστές».



Σχήμα 7.9. Δέντρο αποφάσεων με βάση τους ανωτέρω εμπειρικούς κανόνες

Χαρακτηρισμός	Αυστηρά Όρια Speedup	Χαλαρά Όρια Speedup
Good	$\text{Speedup} \geq 1.1$	$\text{Speedup} \geq 1.05$
Stationary	$0.95 \leq \text{Speedup} < 1.1$	$0.90 \leq \text{Speedup} < 1.15$
Bad	$\text{Speedup} < 0.95$	$\text{Speedup} \leq 0.95$

Πίνακας 7.6. Σύγκριση των ορίων speedup για την αυστηρή και την χαλαρή κατηγοριοποίηση

Στον Πίνακα 7.7, αναλύουμε και παρουσιάζουμε την ακρίβεια του μοντέλου στην πρόβλεψη του NAS heatmap. Παρατηρούμε ότι το μοντέλο προέβλεψε με ακρίβεια το 83.43% των ζευγών στο NAS heatmap, ενώ το 7.1% των προβλέψεων του ανήκει στην κατηγορία «Επαρκώς Σωστές». Ως αποτέλεσμα, **οι προβλέψεις ήταν τουλάχιστον επαρκείς για το 90.53% των ζευγών**. Προκειμένου να αξιολογήσουμε αν το μοντέλο πάσχει από υπερπροσαρμογή (overfitting), λόγω του ότι οι εμπειρικοί κανόνες βασίστηκαν σε μεγάλο βαθμό στα NAS benchmarks, χρησιμοποιούμε το ίδιο μοντέλο και για την πρόβλεψη του SPEC heatmap. Η ακρίβεια του μοντέλου στην πρόβλεψη του SPEC heatmap παρουσιάζεται επίσης στον Πίνακα 7.7. Ενώ το ποσοστό των πλήρως σωστών προβλέψεων είναι σημαντικά μικρότερο (63.27%) σε σχέση με τα NAS benchmarks, παρατηρούμε ότι ένας μεγάλος αριθμός προβλέψεων (22.45%) εμπίπτει στην κατηγορία «Επαρκώς Σωστές». Ως αποτέλεσμα, **οι προβλέψεις ήταν τουλάχιστον επαρκείς για το 85.72% των ζευγών**, γεγονός που παραμένει ικανοποιητικό, ενώ οι λανθασμένες προβλέψεις αντιστοιχούν μόνο στο 14.29% του συνόλου των ζευγών.

Το βασικό πλεονέκτημα του εμπειρικού μοντέλου έγκειται στην ερμηνευσιμότητά του. Ενώ προσφέρει ικανοποιητική ακρίβεια χωρίς τη χρήση τεχνικών ML, η εφαρμοσιμότητά του σε άλλες μηχανές μπορεί

Χαρακτηρισμός	NAS Benchmarks	SPEC Benchmarks
Πλήρως Σωστές	141 από τις 169 - 83.43%	31 από τις 49 - 63.27%
Επαρκώς Σωστές	12 από τις 169 - 7.1%	11 από τις 49 - 22.45%
Λάθος	16 από τις 169 - 9.47%	7 από τις 49 - 14.29%

**Πίνακας 7.7.** Αποτίμηση προβλέψεων του εμπειρικού μοντέλου για τα NAS και SPEC benchmarks

να είναι περιορισμένη, καθώς αναπτύχθηκε χρησιμοποιώντας δεδομένα από ένα μόνο HPC cluster. Επιπλέον, λόγω των πολύπλοκων σχέσεων μεταξύ των δεδομένων profiling και της απόδοσης συνεκτέλεσης, λαμβάνει τη μορφή ενός αρκετά πολύπλοκου δέντρου αποφάσεων.

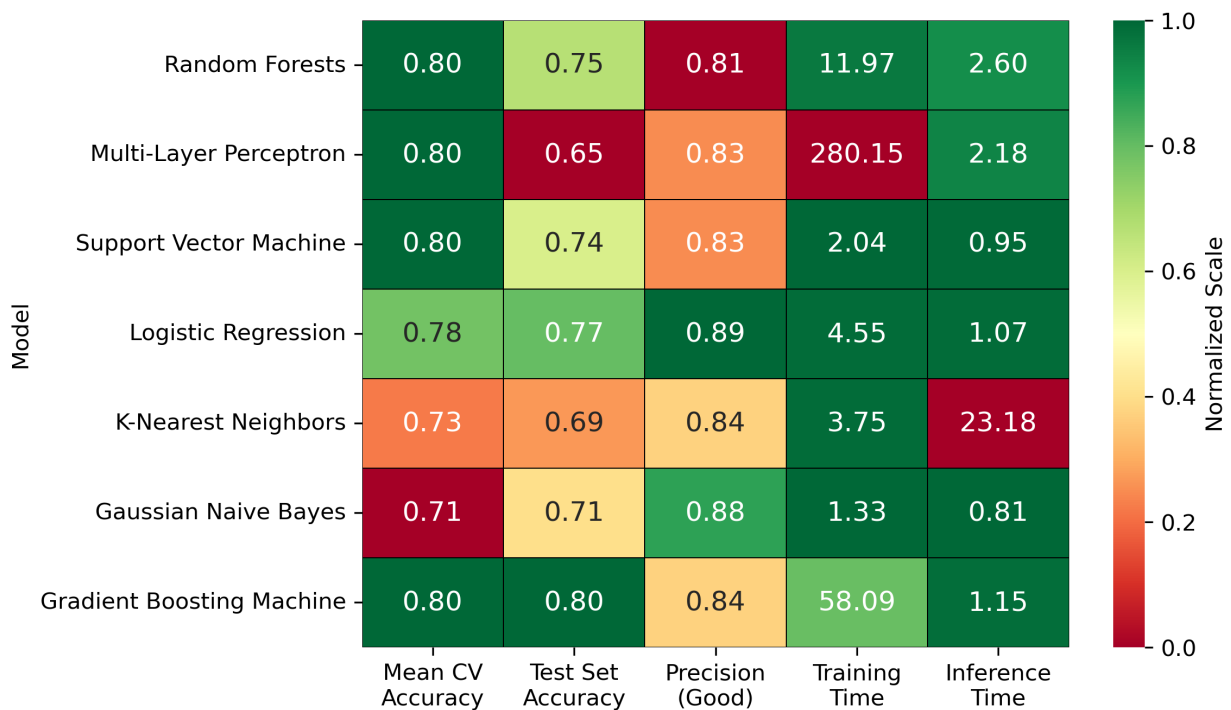
## 7.10 Pairwise μοντέλα : ML classifier

Τα μοντέλα Μηχανικής Μάθησης έχουν χρησιμοποιηθεί σε προηγούμενες έρευνες συνεκτέλεσης με σκοπό την πρόβλεψη του heatmap ενός συγκεκριμένου συνόλου εφαρμογών. Βασιζόμενοι σε προηγούμενες προσεγγίσεις [29, 30], εκπαιδεύουμε διάφορα μοντέλα ταξινόμησης ML, ώστε να τα συγκρίνουμε με την εμπειρική προσέγγιση. Χρησιμοποιούμε το ίδιο σύνολο δεδομένων με τη μελέτη της εμπειρικής μεθόδου, το οποίο αποτελείται συνολικά από 218 εγγραφές, με το 70% να διατίθεται για training και το υπόλοιπο 30% για testing. Χαρακτηρίζουμε ξανά κάθε κελί του heatmap ως good, stationary ή bad. Χρησιμοποιούμε τους πέντε άξονες που παρουσιάστηκαν προηγουμένως στα spider plots ως features στο σύνολο εκπαίδευσης, καθώς πιστεύουμε ότι παρέχουν μια καλή επισκόπηση της συμπεριφοράς της εφαρμογής και είχαν επίσης χρησιμοποιηθεί κατά την ανάπτυξη του εμπειρικού μοντέλου.

Το πρώτο βήμα είναι η επιλογή των κατάλληλων ML μοντέλων με βάση τη συγκεκριμένη περίπτωση χρήσης. Το μοντέλο θα πρέπει να πραγματοποιεί συχνές προβλέψεις του εκάστοτε heatmap, καθώς το σύνολο εφαρμογών μεταβάλλεται με νέες υποβολές εργασιών. Ταυτόχρονα, δεν υπάρχει ανάγκη για γρήγορη εκπαίδευση, καθώς εκείνη όταν απαιτείται μπορεί να εκτελείται κατά τις ώρες χαμηλού φόρτου. Στόχος μας είναι η χρήση ενός μικρού αριθμού features, ώστε να αποφευχθεί η υπερβολική χρονική πολυπλεξία κατά την καταγραφή performance counters από το perf. Δεδομένου ότι η συνεκτέλεση εξακολουθεί να βρίσκεται σε πειραματικό στάδιο, τα σύνολα δεδομένων μας για εκπαίδευση και επικύρωση είναι μικρά και προέρχονται κυρίως από ερευνητικά πειράματα. Λαμβάνοντας υπόψη τα παραπάνω, εκπαιδεύσαμε επτά διαφορετικούς τύπους μοντέλων Machine Learning και συγκρίναμε τα αποτελέσματα των προβλέψεών τους. Για κάθε τύπο μοντέλου, εκπαιδεύσαμε διάφορα μοντέλα με πολλούς πιθανούς συνδυασμούς hyperparameters. Τα δείγματα στο σύνολο εκπαίδευσης κανονικοποιήθηκαν ώστε να επιτευχθεί καλύτερη ακρίβεια, σύμφωνα με τα ανώτερα και κατώτερα όρια των μετρικών, όπως υπολογίστηκαν στο σύστημα ARIS. Ωστόσο, αυτή η κανονικοποίηση απαιτεί εκ νέου εκπαίδευση των μοντέλων εάν πρόκειται να χρησιμοποιηθούν σε άλλα συστήματα με διαφορετικά όρια μετρικών, όπως FLOPS, εύρος ζώνης μνήμης κ.λπ. Για κάθε μοντέλο, πραγματοποιήσαμε 5-fold cross-validation και υπολογίσαμε τον μέσο όρο των accuracy scores των επιμέρους folds. Στη συνέχεια, για κάθε τύπο μοντέλου, διατηρήσαμε το μοντέλο με τη μεγαλύτερη μέση cross-validation accuracy και το αξιολογήσαμε χρησιμοποιώντας ένα άγνωστο test set. Το Σχήμα 7.10 παρουσιάζει ορισμένες μετρικές σχετικά με το καλύτερο μοντέλο από κάθε τύπο μοντέλου. Οι χρόνοι εκπαίδευσης



και πρόβλεψης παρουσιάζονται σε χιλιοστά του δευτερολέπτου.



Σχήμα 7.10. Αξιολόγηση των καλύτερων μοντέλων από κάθε τύπο μοντέλου

Τα αποτελέσματα αυτά μπορούν να ερμηνευθούν εξετάζοντας τα εγγενή χαρακτηριστικά κάθε τύπου μοντέλου [31]. Συγκεκριμένα:

- **Neural Networks (MLP):** Απαιτούν μεγάλα σύνολα εκπαίδευσης για υψηλή ακρίβεια. Παρά την ακρίβεια 0.8 στο cross-validation, το MLP απέδωσε άσχημα σε μη γνωστά δεδομένα (0.65) και είχε μεγαλύτερο χρόνο εκπαίδευσης. Επιπλέον, δεν είναι ερμηνεύσιμο.
- **Gradient Boosting Machine (GBM), Random Forests (RF) και Logistic Regression (LR):** Πέτυχαν τα καλύτερα συνολικά αποτελέσματα. Το GBM και το RF προσφέρουν υψηλότερη ακρίβεια και λιγότερη υπερπροσαρμογή, ενώ το LR επωφελείται από το μικρό μας σύνολο δεδομένων λόγω της απλότητάς του.
- **K-Nearest Neighbors (KNN) και Gaussian Naive Bayes (GNB):** Δεν απέδωσαν όσο καλά όσο τα υπόλοιπα, πιθανώς λόγω των πολύπλοκων σχέσεων μεταξύ των χαρακτηριστικών. Το KNN έχει επίσης υψηλό χρόνο πρόβλεψης, ο οποίος αυξάνεται καθώς μεγαλώνει το σύνολο εκπαίδευσης.
- **Support Vector Machine (SVM):** Επίσης απέδωσε καλά. Η ισχύς του έγκειται σε μικρά, υψηλών διαστάσεων σύνολα δεδομένων, κάτι που όμως δεν είναι ιδιαίτερα σχετικό στην περίπτωση μας.

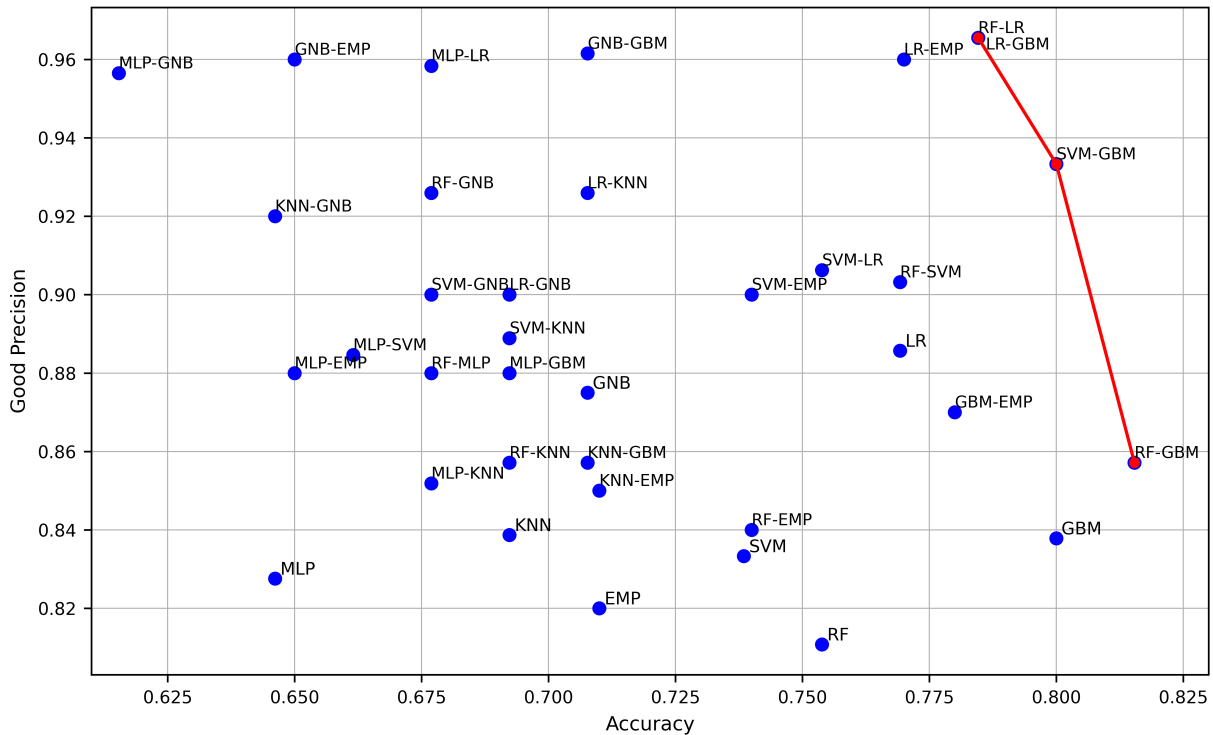


Ένας βασικός περιορισμός για πολλά από αυτά τα μοντέλα είναι το σχετικά μικρό σύνολο εκπαίδευσης. Καθώς η συνεκτέλεση ενσωματώνεται περαιτέρω σε production συστήματα, αυτό μπορεί εύκολα να αλλάξει, καθώς θα αποκτήσουμε περισσότερα πραγματικά δεδομένα. Στη συνέχεια, η αποδοτικότητα των Νευρωνικών Δικτύων μπορεί να επανεξεταστεί, ειδικά δεδομένου ότι ο χρόνος εκπαίδευσης μπορεί να μην είναι τόσο κρίσιμος όσο η ταχύτητα πρόβλεψης και η ακρίβεια.

Όπως επισημαίνεται στο [29], η υποτίμηση της υποβάθμισης της επίδοσης μιας εφαρμογής σε ένα σενάριο συνεκτέλεσης μπορεί να έχει σοβαρές συνέπειες σε περιβάλλοντα πραγματικού χρόνου, καθώς μπορεί να οδηγήσει τον scheduler σε λανθασμένη συνεκτέλεση μη συμβατών εφαρμογών. Παρομοίως, στην περίπτωση μας, τα False Positives που επισημαίνονται ως ‘good’ μπορεί να παραπλανήσουν τον scheduler, επηρεάζοντας αρνητικά το makespan του φορτίου εργασίας. Για τον λόγο αυτό, θα μπορούσε να είναι χρήσιμο να προσπαθήσουμε να βελτιώσουμε το precision των μοντέλων μας όσον αφορά την ετικέτα ‘good’ (δηλαδή το ποσοστό των δειγμάτων που προβλέπονται ως ‘good’ και είναι πράγματι ‘good’). Ένας τρόπος για να το επιτύχουμε αυτό είναι να χρησιμοποιήσουμε δύο μοντέλα ταυτόχρονα και να χαρακτηρίσουμε ένα δείγμα ως ‘good’ μόνο εάν και τα δύο μοντέλα το έχουν προβλέψει ως ‘good’. Αυτό μπορεί φυσικά να επηρεάσει το συνολικό accuracy του μοντέλου. Καταλήγουμε έτσι σε ένα πρόβλημα βελτιστοποίησης δύο κριτηρίων: θέλουμε να έχουμε υψηλό συνολικό accuracy, διατηρώντας παράλληλα υψηλό precision για την ετικέτα ‘good’.

Για να εξετάσουμε περαιτέρω αυτό το ζήτημα, αξιολογούμε όλα τα πιθανά ζεύγη των μοντέλων που παρουσιάστηκαν έως τώρα (το εμπειρικό μοντέλο και τα μοντέλα Machine Learning του Σχήματος 7.10). Τα αποτελέσματα παρουσιάζονται στο Σχήμα 7.11. Η κόκκινη γραμμή απεικονίζει το Pareto frontier του scatterplot, το οποίο δείχνει ότι οι συνδυασμοί μοντέλων που πλησιάζουν περισσότερο την ταυτόχρονη ικανοποίηση και των δύο στόχων βελτιστοποίησής μας είναι τα RF-LR, LR-GBM, SVM-GBM και RF-GBM. Τα τέσσερα μοντέλα που συνθέτουν τα ζεύγη ήταν επίσης από τα πιο αποδοτικά μας μοντέλα όταν αξιολογήθηκαν μεμονωμένα.

Ανάλογα με το επιθυμητό επίπεδο ακρίβειας για την ετικέτα ‘good’, μπορούμε να επιλέξουμε έναν από αυτούς τους συνδυασμούς για χρήση. Είναι σημαντικό να σημειωθεί ότι αυξάνοντας την ακρίβεια για την ετικέτα ‘good’, ενδέχεται να μειώσουμε το recall αυτής της ετικέτας. Στις περιπτώσεις των RF-LR, LR-GBM, SVM-GBM, το recall της ετικέτας ‘good’ μειώθηκε κατά 9% σε σύγκριση με το μεγαλύτερο recall μεταξύ των δύο μοντέλων κάθε ζεύγους. Στην περίπτωση του RF-GBM, όπου το precision της ετικέτας ‘good’ είναι χαμηλότερο, το recall μειώνεται κατά 3%. Η μη πρόβλεψη συγκεκριμένων καλών συνεκτελέσεων (θυσιάζοντας το recall) αποτελεί μια χαμένη ευκαιρία για τον scheduler να αξιοποιήσει καλύτερα το σύνολο εφαρμογών. Από την άλλη πλευρά, η εσφαλμένη πρόβλεψη πολλών κακών συνεκτελέσεων ως ‘good’ (θυσιάζοντας το precision) μπορεί να οδηγήσει σε επιβράδυνση του συνολικού makespan, λόγω ασύμβατων συνδυασμών από τον scheduler. Πρόκειται για ένα trade-off που αξίζει να διερευνηθεί και μπορεί επίσης να εξαρτάται από τον ίδιο τον αλγόριθμο συνεκτέλεσης. Συνοψίζοντας, οι ensemble techniques φαίνεται να είναι οι πιο αποδοτικές στην περίπτωση χρήσης μας εξαιτίας της σύνθετης σχέσης μεταξύ των performance counters και της συμπεριφοράς των εφαρμογών



Σχήμα 7.11. Pareto Plot όλων των μοντέλων μας και όλων των πιθανών συνδυασμών μοντέλων

σε καθεστώς συνεκτέλεσης.

## 7.11 Προσομοιώσεις Συνεκτέλεσης

Τέλος προσπαθήσαμε να κάνουμε μια αρχική αξιολόγηση της συμπεριφοράς της συνδρομολόγησης με χρήση ενός εργαλείου προσομοίωσης χρονοπρογραμμάτων που ονομάζεται ELiSE (Efficient Lightweight Scheduling Estimator). Αρχικά, χρησιμοποιήσαμε έναν αρκετά απλό συνδρομολογητή, τον EASY Co-Scheduler ο οποίος συν-τοποθετεί τις δουλειές με FCFS τρόπο χρησιμοποιώντας παράλληλα και EASY backfilling που είναι μια σχετικά aggressive μέθοδος backfilling. Ο Πίνακας 7.8 καταδεικνύει τις υψηλές βελτιώσεις στο makespan για φορτία εργασίας που περιέχουν εργασίες με ίσο αριθμό διεργασιών και για διαφορετικά mean job speedups, τα οποία υπολογίζονται βάσει των speedups όλων των πιθανών ζευγών εργασιών σε κάθε φορτίο εργασίας, χρησιμοποιώντας το Marconi heatmap. Κάθε φορτίο εργασίας αποτελείται από 500 εργασίες, και για κάθε περίπτωση μέσου speedup, πραγματοποιήσαμε 5 τυχαία πειράματα σε ένα cluster με 100 κόμβους και 48 πυρήνες.

Αυτά τα αποτελέσματα μας οδηγούν σε δύο παρατηρήσεις: Πρώτον, σημειώνουμε ότι η βελτίωση του makespan μέσω της συνεκτέλεσης σε αυτήν την ευνοϊκή περίπτωση είναι εντυπωσιακή, καθώς φτάνει σχεδόν το 31%. Δεύτερον, παρατηρούμε μια συσχέτιση μεταξύ του μέσου speedup όλων των ζευγών εργασιών σε ένα σύνολο εφαρμογών και της βελτίωσης του makespan που επιτυγχάνεται μέσω της συνεκτέλεσης.

Mean Job Speedup	Makespan Improvement (%)
1.07	24.42%
1.10	25.29%
1.12	25.87%
1.155	30.63%

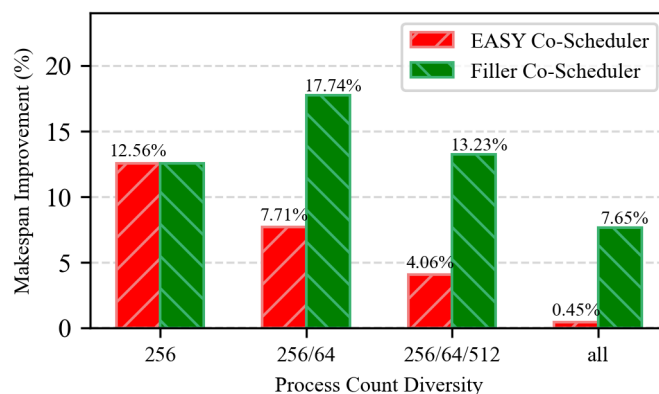
**Πίνακας 7.8.** Βελτίωση του makespan σε σχέση με το Mean Job Speedup του κάθε sub-heatmap

Ωστόσο, με την εκτέλεση περισσότερων προσομοιώσεων γίνεται εμφανές ότι η επίδοση του απλού EASY Co-Scheduler επηρεάζεται από την ποικιλία διαφορετικών πληθών διεργασιών σε ένα σύνολο εφαρμογών. Πειραματιστήκαμε με τέσσερις τύπους φορτίων εργασίας, καθένas από τους οποίους περιλάμβανε 500 εργασίες από το σύστημα ARIS. Κάθε τύπος φορτίου εργασίας αποτελούνταν από εργασίες με αυξανόμενη ποικιλομορφία στον αριθμό των διεργασιών, ενώ κάθε πείραμα εκτελέστηκε αναδιατεταγμένο πέντε φορές. Στο Σχήμα 7.12 παρουσιάζεται η βελτίωση του μη εξελιγμένου EASY Co-Scheduler σε σχέση με τον κλασικό EASY Scheduler. Αρχικά, παρατηρούμε ότι όταν εξετάζουμε μόνο εργασίες που ζητούν 256 διεργασίες, επιτυγχάνουμε μια βελτίωση στο makespan της τάξης του 12.6%, γεγονός που υποδηλώνει σημαντική αύξηση του ρυθμού διεκπεραίωσης λόγω συνεκτέλεσης. Ωστόσο, καθώς το φορτίο εργασίας γίνεται πιο ποικιλόμορφο ως προς τον αριθμό των διεργασιών, η αφελής συνεκτέλεση δυσκολεύεται να διατηρήσει το όφελός της, φτάνοντας μάλιστα στο σημείο να το χάνει όταν αντιμετωπίζει πλήρη ποικιλομορφία διεργασιών.

Με τη βοήθεια των εργαλείων οπτικοποίησης του ELiSE, μπορέσαμε να κατανοήσουμε εύκολα ότι αυτό οφειλόταν στη χαμηλή χρησιμοποίηση των πόρων, καθώς η αφελής συνεκτέλεση οδηγούσε σε κατακερματισμό και ανεχμετάλλευτους πόρους του συστήματος. Στη συνέχεια, υλοποιήσαμε τον Filler Co-Scheduler, ο οποίος συμπληρώνει με best-fit τρόπο τους ανεχμετάλλευτους πόρους με τις καταλλήλότερες εργασίες. Ο Filler καταφέρνει να διατηρήσει τις βελτιώσεις της συνεκτέλεσης, όμως με το τίμημα της παραβίασης της αρχής FCFS, καθιστώντας τον βέβαια άδικο. Αυτή η συμπεριφορά αναδεικνύει τόσο τις δυνατότητες της συνδρομολόγησης όσο και την ανάγκη για sophisticated Co-Schedulers ώστε να έχουμε βελτιώσεις του makespan και σε πιο σύνθετα σενάρια.

Δεδομένου ότι για σύνθετα σενάρια απαιτείται sophistication στον χρονοδρομολογητή μελετήσαμε την σημασία κάποιων χαρακτηριστικών μετρικών για την βελτίωση του makespan. Διεξαγάγαμε 100 πειράματα για ένα συγκεκριμένο σύνολο εφαρμογών χρησιμοποιώντας το ELiSE, το heatmap από τον υπερυπολογιστή ARIS και ένα cluster 200 κόμβων με 20 πυρήνες ανά κόμβο. Το φορτίο εργασίας για κάθε πείραμα αποτελούνταν από τις ίδιες ακριβώς εφαρμογές, αλλά η σειρά τους αναδιατάχθηκε τυχαία κάθε φορά.

Το Σχήμα 7.13 απεικονίζει τη βελτίωση του makespan και τη συσχέτισή του με το μέσο speedup των διεργασιών και το utilization του συστήματος. Ενώ και οι δύο μετρικές παρουσιάζουν μια σαφή



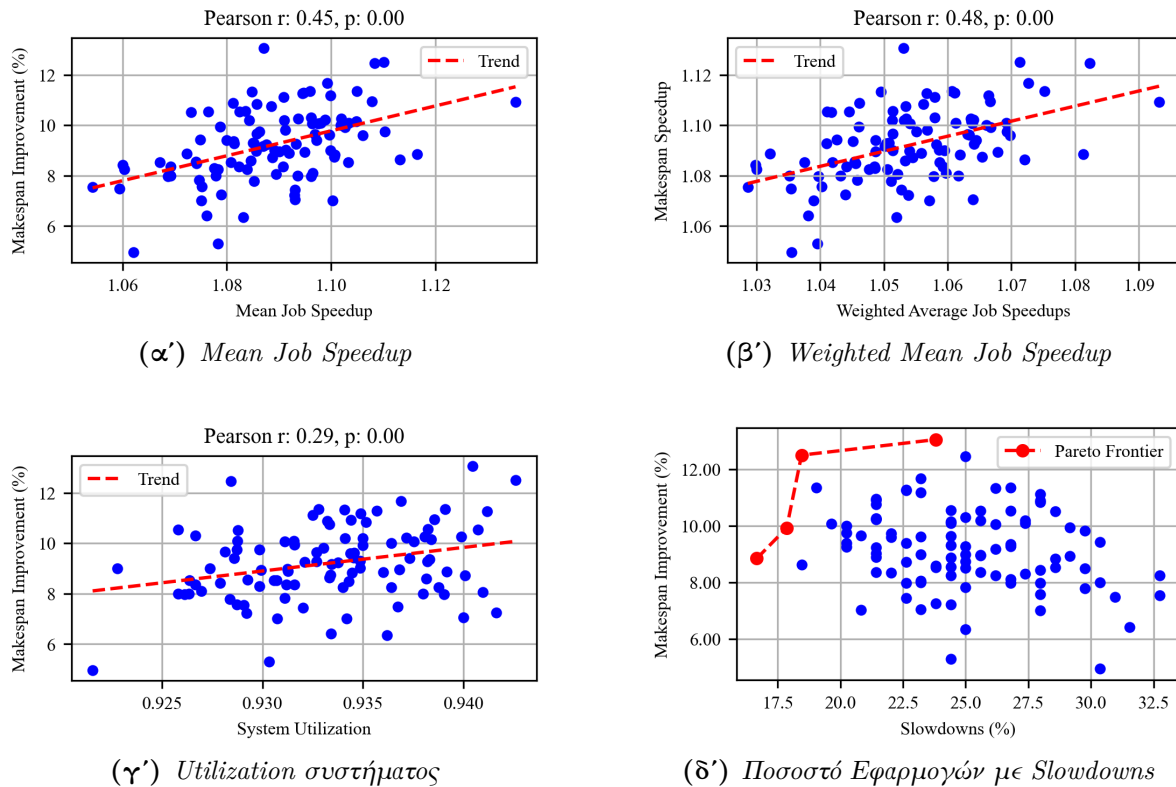
**Σχήμα 7.12.** Βελτίωση του makespan για δύο αλγορίθμους συνδρομολόγησης σε σχέση με τον απλό EASY Scheduler και για διαφορετικές ποικιλίες από πλήθη διεργασιών

τάση, τα speedups των εργασιών φαίνεται να έχουν μεγαλύτερη επίδραση, τουλάχιστον στο πλαίσιο του συγκεκριμένου πειράματος. Αυτές οι παρατηρήσεις μπορούν να καθοδηγήσουν τον μελλοντικό σχεδιασμό αλγορίθμων. Επιπλέον, η σημασία των speedups των εργασιών για την βελτίωση του συνολικού makespan αναδεικνύει τον ρόλο των προηγούμενων παρουσιασμένων μοντέλων που έχουν ως σκοπό την πρόβλεψη των heatmaps και την παροχή δεδομένων στον Co-Scheduler. Στην συνέχεια, το κάτω τμήμα του σχήματος επισημαίνει ότι οι διαφορετικές αναδιατάξεις των εργασιών οδήγησαν σε ένα ευρύ φάσμα αποτελεσμάτων για τον EASY Co-Scheduler ως προς δύο κρίσιμες μετρικές: τη βελτίωση του makespan (ικανοποίηση συστήματος) και το ποσοστό των εργασιών που παρουσίασαν slowdown σε σχέση με τη compact εκτέλεση (ικανοποίηση χρήστη). Αυτή η μεταβλητότητα υπογραμμίζει ακόμη περισσότερο την ανάγκη για έναν πιο εξελιγμένο αλγόριθμο συνεκτέλεσης, ώστε να επιλυθεί αυτό το πρόβλημα βελτιστοποίησης δύο κριτηρίων.

## 7.12 Μελλοντικές Ερευνητικές Κατευθύνσεις

Βάσει των ευρημάτων μας, διαμορφώνουμε τις ακόλουθες κατευθύνσεις για μελλοντική έρευνα και βελτίωση της συνεκτέλεσης εργασιών σε HPC συστήματα:

- **Ανάπτυξη εξελιγμένων αλγορίθμων συνεκτέλεσης:** Σχεδιασμός προηγμένων αλγορίθμων συνεκτέλεσης που αξιοποιούν τα δεδομένα από τα μοντέλα εφαρμογών μας καθώς και τα συμπεράσματα της παραπάνω αξιολόγησης της συνδρομολόγησης.
- **Υλοποίηση διπλής ουράς εργασιών:** Δημιουργία δύο ξεχωριστών ουρών στο υπερυπολογιστικό σύστημα:
  - μια παραδοσιακή ουρά για εργασίες που εκτελούνται με τον παραδοσιακό compact τρόπο
  - μια ειδική ουρά συνεκτέλεσης για εφαρμογές που μπορούν να ωφεληθούν από τη συνε-



**Σχήμα 7.13.** Συσχέτιση μεταξύ της βελτίωσης του *Makespan* και διάφορων μετρικών που σχετίζονται με την ικανοποίηση του συστήματος και του χρήστη

κτέλεση.

Η κατάλληλη κατανομή εργασιών μεταξύ των δύο ουρών θα μπορούσε να οδηγήσει σε σημαντικές βελτιώσεις στο makespan και τη χρήση πόρων και να επιτευχθεί με βάση τα προαναφερθέντα μοντέλα.

- **Υλοποίηση νέων μοντέλων:** Ανάπτυξη co-location-centric tag-based και Machine Learning Regression μοντέλων αλλά και νέων προσεγγίσεων στα υπάρχοντα μοντέλα της ταξινόμησής μας.
- **Εκ νέου αξιολόγηση σύνθετων ML/NN μοντέλων με πραγματικά δεδομένα:** Με την ενσωμάτωση της συνεκτέλεσης σε production συστήματα, θα καταστεί εφικτή η εκπαίδευση και αξιολόγηση πιο σύνθετων μοντέλων Μηχανικής Μάθησης και Νευρωνικών Δικτύων με μεγαλύτερα και πιο αντιπροσωπευτικά σύνολα δεδομένα. Αυτό θα μπορούσε να οδηγήσει σε πιο αξιόπιστες και ακριβείς προβλέψεις για τη συνεκτέλεση εφαρμογών, δεδομένων και των σύνθετων σχέσεων που συνδέουν τα δεδομένα κατανάλωσης πόρων με την συμπεριφορά κατά την συν-τοποθέτηση που προβλέπεται ότι θα ευνοήσουν τα σύνθετα, μη-γραμμικά μοντέλα Μηχανικής Μάθησης.

- **Καθορισμός πολιτικής πληρωμών σε HPC συστήματα:** Μελέτη σχημάτων πληρωμών που θα λαμβάνουν υπόψη τα πιθανά slowdowns λόγω συνεκτέλεσης. Θα πρέπει να διερευνηθούν μηχανισμοί αποζημίωσης ή διαφοροποιημένης τιμολόγησης που να αντικατοπτρίζουν τις επιπτώσεις της συνεκτέλεσης στις εφαρμογές των χρηστών.
- **Συνεκτέλεση περισσότερων από δύο εφαρμογών ανά κόμβο:** Διερεύνηση της δυνατότητας συνεκτέλεσης περισσότερων από δύο εφαρμογών στον ίδιο υπολογιστικό κόμβο, λαμβάνοντας υπόψη τη χρήση πόρων και τις αλληλεπιδράσεις μεταξύ πολλαπλών εφαρμογών.
- **Καταμερισμός της cache (Cache Partitioning):** Υλοποίηση στρατηγικών καταμερισμού της μνήμης cache ώστε να ελαχιστοποιηθεί η αλληλεπίδραση και ο ανταγωνισμός μεταξύ εφαρμογών που συνεκτελούνται, αυξάνοντας έτσι τη σταθερότητα και την προβλεψιμότητα της απόδοσης.
- **Ενεργειακή αποδοτικότητα:** Μελέτη της επίδρασης της συνεκτέλεσης στην κατανάλωση ενέργειας των HPC συστημάτων και ανάπτυξη τεχνικών που επιτρέπουν εξοικονόμηση ενέργειας χωρίς να υποβαθμίζεται η απόδοση των εφαρμογών.



---

## Bibliography

---

- [1] Alex D Breslow, Leo Porter, Ananta Tiwari, Michael Laurenzano, Laura Carrington, Dean M Tullsen και Allan E Snaveley. *The case for colocation of high performance computing workloads. Concurrency and Computation: Practice and Experience*, 28(2):232–251, 2016.
- [2] University of Florida Research Computing. *Memory: Shared vs Distributed*. Accessed: 2024-11-12.
- [3] Avinash Sodani. *Knights landing (KNL): 2nd Generation Intel® Xeon Phi processor. 2015 IEEE Hot Chips 27 Symposium (HCS)*, σελίδες 1–24, 2015.
- [4] GRNET. *ARIS Documentation*. <https://doc.aris.grnet.gr/>. Accessed: 2024-11-17.
- [5] SchedMD. *Slurm Workload Manager Quick Start Guide*. <https://slurm.schedmd.com/quickstart.html>. Accessed: 2024-11-17.
- [6] Standard Performance Evaluation Corporation (SPEC). *SPEC HPC 2021 Benchmark Suite*. <https://www.spec.org/hpc2021/Docs/>, 2021. Accessed: 2024-11-19.
- [7] Performance Optimisation και Productivity (POP) Centre of Excellence. *Using POP Tools: Score-P and Scalasca*, 2021. Accessed: 2024-11-24.
- [8] Matthew J Koop, Miao Luo και Dhabaleswar K Panda. *Reducing network contention with mixed workloads on modern multicore, clusters. 2009 IEEE International Conference on Cluster Computing and Workshops*, σελίδες 1–10. IEEE, 2009.
- [9] Intel Corporation. *HPC Performance Characterization Analysis*, 2025. Accessed: 2024-12-24.
- [10] GRNET. *Introduction to Supercomputers and ARIS System*, 2024. Accessed: 2024-11-10.



- [11] Philippe Olivier Alexandre Navaux, Arthur Francisco Lorenzon και Matheusda Silva Serpa. *Challenges in high-performance computing. Journal of the Brazilian Computer Society*, 29(1):51–62, 2023.
- [12] Michael Flynn. *Flynn's Taxonomy*, σελίδες 689–697. Springer US, Boston, MA, 2011.
- [13] Ayesha Afzal, Georg Hager και Gerhard Wellein. *SPEChpc 2021 Benchmarks on Ice Lake and Sapphire Rapids Infiniband Clusters: A Performance and Energy Case Study*. σελίδες 1245–1254, 2023.
- [14] D.H. Bailey, E. Barszcz, L. Dagum και H.D. Simon. *NAS parallel benchmark results. Supercomputing '92: Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, σελίδες 386–393, 1992.
- [15] John D. McCalpin. *Memory Bandwidth and Machine Balance in Current High Performance Computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, σελίδες 19–25, 1995.
- [16] Score P Team. *Score-P User Manual (Version 6.0)*. Jülich Supercomputing Centre. Accessed: February 25, 2025.
- [17] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen και Michael D. Smith. *System support for automatic profiling and optimization. Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles, SOSP '97*, σελίδα 15–26, New York, NY, USA, 1997. Association for Computing Machinery.
- [18] Corey Malone, Mohamed Zahran και Ramesh Karri. *Are hardware performance counters a cost effective way for integrity checking of programs. Proceedings of the Sixth ACM Workshop on Scalable Trusted Computing, STC '11*, σελίδα 71–76, New York, NY, USA, 2011. Association for Computing Machinery.
- [19] Gaddisa Olani Ganfure, Chun Feng Wu, Yuan Hao Chang και Wei Kuan Shih. *DeepWare: Imaging Performance Counters With Deep Learning to Detect Ransomware. IEEE Transactions on Computers*, 72(3):600–613, 2023.
- [20] PerfWiki. *PerfWiki: Performance Benchmarking Resource*. <https://perfwiki.github.io/main/>. Accessed: 2024-12-01.
- [21] Intel Corporation. *Intel Nehalem Performance Monitoring Unit (PMU) Programming Guide*,

2008. Accessed: 2024-12-26.
- [22] Lawrence Livermore National Laboratory. *mpiP: A Lightweight MPI Profiling Library*. <https://software.llnl.gov/mpiP/>. Accessed: 2024-12-16.
  - [23] Robin Boëzennec, Fanny Dufossé και Guillaume Pallez. *Qualitatively analyzing optimization objectives in the design of hpc resource manager*. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 2024.
  - [24] Christiane P Ribeiro, Márcio B Castro, Vania Marangozova-Martin, Jean François Méhaut, Henrique C Freitas και Carlos APS Martins. *INVESTIGATING THE IMPACT OF CPU AND MEMORY AFFINITY ON MULTI-CORE PLATFORMS: A CASE STUDY OF NUMERICAL SCIENTIFIC MULTITHREADED BENCHMARKS*.
  - [25] Chiara Vercellino, Alberto Scionti, Giuseppe Varavallo, Paolo Viviani, Giacomo Vitali και Olivier Terzo. *A machine learning approach for an HPC use case: The jobs queuing time prediction*. *Future Generation Computer Systems*, 143:215–230, 2023.
  - [26] Srividya Srinivasan, Rajkumar Kettimuthu, Vijay Subramani και Ponnuswamy Sadayappan. *Characterization of backfilling strategies for parallel job scheduling*. *Proceedings. International Conference on Parallel Processing Workshop*, σελίδες 514–519. IEEE, 2002.
  - [27] Yunlian Jiang, Kai Tian, Xipeng Shen, Jinghe Zhang, Jie Chen και Rahul Tripathi. *The complexity of optimal job co-scheduling on chip multiprocessors and heuristics-based solutions*. *IEEE Transactions on Parallel and Distributed Systems*, 22(7):1192–1205, 2010.
  - [28] Andreas de Blanche και Thomas Lundqvist. *Terrible twins: A simple scheme to avoid bad co-schedules*. *Proceedings of the 1st COSH Workshop on Co-Scheduling of HPC Applications*, σελίδα 25, 2016.
  - [29] Felipe Vieira Zacarias, Vinicius Petrucci, Rajiv Nishtala, Paul Carpenter και Daniel Mossé. *Intelligent colocation of HPC workloads*. *Journal of Parallel and Distributed Computing*, 151:125–137, 2021.
  - [30] Nikita Mishra, John D Lafferty και Henry Hoffmann. *Esp: A machine learning approach to predicting application interference*. *2017 IEEE International Conference on Autonomic Computing (ICAC)*, σελίδες 125–134. IEEE, 2017.
  - [31] Hannah Lickert, Aleksandra Wewer, Sören Dittmann, Pinar Bilge και Franz Dietrich. *Selection*

- of suitable machine learning algorithms for classification tasks in reverse logistics. Procedia CIRP*, 96:272–277, 2021.
- [32] *High Performance Edge Computing - The Frontier for Supercomputing and Acceleration*. Accessed: November 16, 2024.
- [33] J. L. Bez και S. Byna. *April 2019 Darshan counters from the Cori supercomputer (1.0.0)*. Zenodo, 2022. Data set.
- [34] J. Dongarra, T. Sterling, H. Simon και E. Strohmaier. *High-performance computing: clusters, constellations, MPPs, and future directions. Computing in Science & Engineering*, 7(2):51–59, 2005.
- [35] MPI Forum. *MPI: A Message-Passing Interface Standard, Version 4.0*, 2021. Accessed: 2024-11-19.
- [36] Evelyn Duesterwald και Vasanth Bala. *Software Profiling for Hot Path Prediction: Less is More*. τόμος 35, σελίδες 202–211, 2000.
- [37] T. Ball και J.R. Larus. *Efficient path profiling. Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, σελίδες 46–57, 1996.
- [38] Jaspal Subhlok, Shreenivasa Venkataramaiah και Amitoj Singh. *Characterizing NAS benchmark performance on shared heterogeneous networks. Proceedings 16th International Parallel and Distributed Processing Symposium*, σελίδες 9–pp. IEEE, 2002.
- [39] Pouya Kousha, Kamal Raj Sankarapandian Dayala Ganesh Ram, Mansa Kedia, Hari Subramoni, Arpan Jain, Aamir Shafi, Dhabaleswar Panda, Trey Dockendorf, Heechang Na και Karen Tomko. *INAM: cross-stack profiling and analysis of communication in MPI-based applications. Practice and Experience in Advanced Research Computing*, σελίδες 1–11. 2021.
- [40] Kevin Menear, Ambarish Nag, Jordan Perr-Sauer, Monte Lunacek, Kristi Potter και Dmitry Duplyakin. *Mastering HPC Runtime Prediction: From Observing Patterns to a Methodological Approach. Practice and Experience in Advanced Research Computing*, σελίδες 75–85. 2023.
- [41] Di Zhang, Dong Dai, Youbiao He, Forrest Sheng Bao και Bing Xie. *RLScheduler: an automated HPC batch job scheduler using reinforcement learning. SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, σελίδες 1–15. IEEE, 2020.

- [42] HPC Wiki Community. *Admin Guide: Scheduling Algorithms*. [https://hpc-wiki.info/hpc/Admin\\_Guide\\_Scheduling\\_Algorithms](https://hpc-wiki.info/hpc/Admin_Guide_Scheduling_Algorithms), 2025. Accessed: 2025-02-17.
- [43] Peter J Keleher, Dmitry Zotkin και Dejan Perkovic. *Attacking the bottlenecks of backfilling schedulers*. *Cluster Computing*, 3(4):245–254, 2000.
- [44] Lizhe Wang, Gregor Von Laszewski, Jay Dayal και Fugang Wang. *Towards energy aware scheduling for precedence constrained parallel tasks in a cluster with DVFS*. *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, σελίδες 368–377. IEEE, 2010.
- [45] Kevin Brown και Satoshi Matsuoka. *Co-locating graph analytics and hpc applications*. *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, σελίδες 659–660. IEEE, 2017.
- [46] David K Newsom, Olivier Serres, Sardar F Azari, Abdel Hameed A Badawy και Tarek El-Ghazawi. *Energy efficient job co-scheduling for high-performance parallel computing clusters*. *2015 IEEE International Conference on Smart City/SocialCom/SustainCom (SmartCity)*, σελίδες 550–556. IEEE, 2015.
- [47] David M Russinoff. *SSE Floating-Point Instructions. Formal Verification of Floating-Point Hardware Design: A Mathematical Approach*, σελίδες 241–246. Springer, 2021.
- [48] Simon Pickartz, Jens Breitbart και Stefan Lankes. *Co-scheduling on upcoming many-core architectures*. *Proceedings of the Joined Workshops COSH 2017 and VisorHPC 2017*, 2017.
- [49] Sakshi Aggarwal. *Machine Learning algorithms, perspectives, and real-world application: Empirical evidence from United States trade data*. 2023.
- [50] Guillaume Aupy, Anne Benoit, Brice Goglin, Loïc Pottier και Yves Robert. *Co-scheduling HPC workloads on cache-partitioned CMP platforms*. *The International Journal of High Performance Computing Applications*, 33(6):1221–1239, 2019.