ELiSE: A tool to support algorithmic design for HPC co-scheduling

No Author Given

No Institute Given

Abstract. As high-performance computing systems continue to advance, new methodologies for efficient resource usage are emerging. One such technique is co-location, where multiple jobs share the same node-level resources throughout their execution in an attempt to improve overall system throughput and job speedup. This approach can be particularly beneficial for memory-bound HPC applications when more memory bandwidth is available. Optimization focuses on two key aspects: user satisfaction and system utilization, each measured by a variety of related metrics. This is a complex problem and there is a shortage of tools that enable researchers to quickly and easily develop and test co-scheduling algorithms with simplicity and accuracy, without requiring extensive environment configuration. In response to that, we propose ELiSE a Python-based framework designed for the rapid development of scheduling algorithms with co-location capabilities, featuring also current-state practices such as backfilling. ELiSE offers a standalone library and operates both under a CLI for numerous experimentations, and under a GUI for ease of use. Key features include the ability to generate different types of static or dynamic workloads, specify system architecture, use in-built or custom-made (co-)scheduling algorithms, visualize performance metrics, and export results. Leveraging these features, we conducted several experiments and we provide preliminary estimates of potential trade-offs and optimization strategies in the realm of co-scheduling.

Keywords: High-Performance Computing · Co-Scheduling · Simulation

1 Introduction

Supercomputing has now entered the exascale era with massive computational power being delivered to end users. Resource and Job Management Systems (RJMS) typically take over the extremely critical task of appropriately assigning this huge amount of resources to user applications. The evolution and adaptation of RJMS to the scale and architecture of HPC systems is a highly challenging task that requires experimentation at scale and with access privileges that are not available to regular users. For this reason, researchers and practitioners resort to some kind of simulation to test new scheduling algorithms and resource management techniques. Domain-specific simulators like AccaSim [12] and CoSim [18] focus on testing scheduling algorithms under various workloads, toolkit-based

simulators like Batsim [10], IRMASim [15], ElastiSim [2], and Alea [16] leverage versatile frameworks such as SimGrid [9] and GridSim [7] to simulate scheduling scenarios with varying levels of realism and real RJMS-based simulators like the Slurm Simulator [22] and ScSF [19] enable workload-based simulations with Slurm.

Co-scheduling has been shown to be a promising technique for increasing the throughput of HPC systems [3, 5, 6, 14, 26]. The main benefit from co-scheduling comes from the fact that proper co-location of two jobs on HPC nodes may boost the performance of memory-bound applications (a typical family in HPC), without affecting the performance of compute-bound or communication-bound ones, aggregately leading to application speedups and in turn increased system throughput. Nevertheless, despite the strong evidence of the benefits of co-scheduling, the technique has not yet found its way to production systems. We attribute this shortcoming to the fact that although we have concrete data on the benefits of placing HPC applications on the same node (co-location), we are still missing information on how this scheme would evolve in a dynamic and long-running scenario where HPC jobs enter and exit the system (co-scheduling).

Simulation seems an excellent initial step to understand the details of co-scheduling behavior before proceeding to the highly cumbersome and costly task of implementing the scheme in a production RJMS and deploying it in a real system. In this way, we will be able to assess key target metrics like system throughput, turnaround times, fairness, etc., understand any obscure implications like resource fragmentation, and ultimately assess candidate co-scheduling algorithms. Unfortunately, existing simulators are often complex to configure, difficult to extend, and lack support for modeling and simulating co-scheduling under space-sharing.

In this paper, we present the Efficient Lightweight Scheduling Estimator (ELiSE), a tool for prototyping (co-)scheduling algorithms on HPC systems targeted for Message Passing Interface (MPI) applications. ELiSE receives as input a) a simple description of an HPC cluster, b) a dynamic load of HPC jobs, and c) a heatmap of pairwise speedups between the participating applications, and simulates the execution of the load on the cluster under different scheduling policies. ELiSE relies on a simple design that avoids full system simulation and can mostly focus on the behavior of different co-scheduling scenarios. ELiSE is easy to use, provides high flexibility in the production of execution workloads, is straightforward to extend to new scheduling algorithms, and provides a rich palette of visual output to support research in the field.

We validated ELiSE by comparing its simulation results with those of a real RJMS (OAR [8]) on a small cluster and show that they have minimal deviations. Moreover, we executed a large number of simulation scenarios shedding light on the behavior of co-scheduling and we report some initial interesting results, i.e. that based on our current co-location data on the NAS Parallel Benchmarks (NPB)[1] on two real clusters: a) co-scheduling even applied blindly can lead to significant throughput improvements, b) fragmentation under the presence of

[1] https://www.nas.nasa.gov/software/npb.html

jobs with diverse process counts may hinder the benefits of co-scheduling, c) job speedups and good system utilization are important factors when considering an effective co-scheduling algorithm, d) different co-scheduling decisions may lead to a significant trade-off between system throughput and user satisfaction.

We intend to release ELiSE as open source, accompanied by comprehensive documentation on configuration and usage. Additionally, data collected from the HPC systems will be made available through an open data repository.

2 Background and Related Work

2.1 Resource and Job Management in HPC

The Resource and Job Management System (RJMS) in an HPC environment coordinates the available resources, such as CPU, memory, accelerators, etc. across incoming jobs, ensuring that they are utilized efficiently and fairly. The RJMS includes a job scheduler that communicates with a resource manager to obtain information about the queues, the loads of computing nodes, and the resource availability in order to make scheduling decisions for the jobs in its waiting queue. Afterwards, it distributes the necessary resources to the corresponding jobs based on specific allocation schemes. Slurm [25], Flux [1] and OAR [8] are some examples of state-of-the-art RJMS systems. The target metrics of an RJMS may vary and are typically configurable, but they usually involve a combination of system throughput and utilization (system-side metrics), and job waiting times or turnaround times (user-side metrics) in order to keep both parties satisfied. The design and implementation of ELiSE is orthogonal to these metrics that are the responsibility of the scheduling algorithm.

2.2 The basics of co-scheduling

Co-execution of applications on a single device has been discussed in several contexts, i.e. cloud computing [17,24], HPC [4,26] and accelerators [21]. In cloud computing co-location of small units of work (e.g. microservices, functions) in a single server is a strong necessity to achieve decent resource utilization. In HPC the resources (typically cores) needed by schedulable units (jobs) are usually orders of magnitude larger than those provided by a single HPC node. The standard and straightforward approach to placing jobs on HPC systems is to grant them a number of full nodes (we call this scheme *compact allocation*). However, large families of HPC applications are highly memory-bound and do not scale with the number of cores available within the node. With the number of CPU cores consistently growing and the memory bandwidth not being able to keep up the pace, the phenomenon of poor scalability of HPC applications is constantly worsening.

Applications struggling for memory bandwidth would clearly benefit from being granted more memory channels, which in the HPC context could be achieved by spreading them to more compute nodes. Figure 1 presents the speedup of four

NPB benchmarks that request 256 cores executed on the Grid5000-Grvingt (see Table 1) cluster when granted: a) 16 dual-CPU nodes occupying half of the available CPU cores (8 out of 16) and b) 32 dual-CPU nodes occupying a quarter of the available CPU cores (4 out of 16), compared to the standard practice of compact allocation (8 dual-CPU nodes, with full allocation). In all cases, we observe a significant speedup of all applications without modifying the application code. Clearly, this performance improvement comes at the cost of low system utilization. It is then the responsibility of the RJMS to place proper neighboring applications to fill in the available cores so that these benefits are not lost.

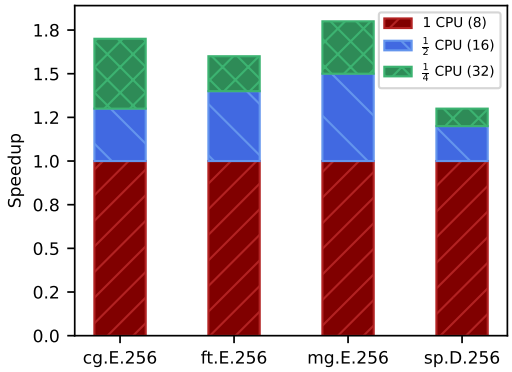

Fig. 1: The effect of spreading HPC applications in more compute nodes

Figure 2 shows a heatmap of speedups from the co-location of eight NPB benchmarks on the Marconi HPC system (see Table 1). In each co-execution scenario, we placed each pair within the same node and repeated the pairing for all the needed number of nodes. The pair was co-executed for 10 minutes, during which any finished job was restarted. We keep the median execution times from this repeated run as the co-execution time of the benchmark. The speedups reported are calculated as the ratio of the original time (when executed with the compact allocation) to the co-execution time. We may notice that for the majority of the co-locations, the scheme was beneficial, leading to an average speedup of 1.12, an initial indication of the potential of co-scheduling. We now need to answer the question of whether this benefit will materialize in a dynamic scenario where applications enter and exit the HPC system.

2.3 Related work

Several HPC simulation frameworks and tools have been proposed with varying scopes. They can be generally grouped into three categories: a) Custom simulators built from scratch, which are in most cases domain-driven, b) simulators built on top of a simulation toolkit like SimGrid [9], GridSim [7] and Structural

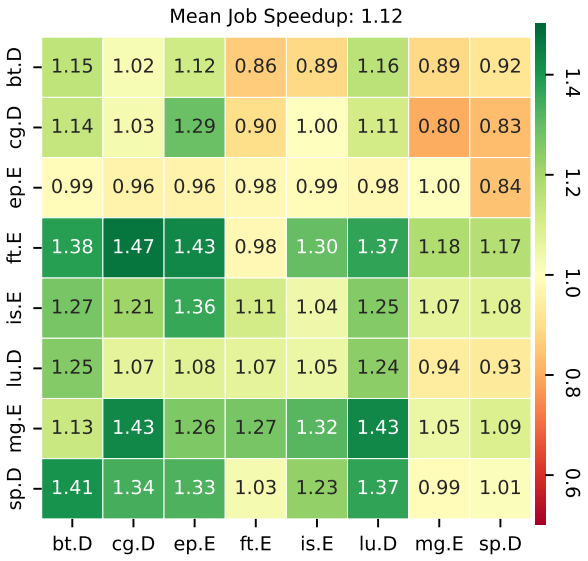

Fig. 2: Heatmap of speedups from the co-execution of eight NPB benchmarks of 256 processes each

Simulation Toolkit (SST) [20], and c) simulators that use production-level RJMS source code as their base.

Domain-specific simulators: AccaSim [12] is a free HPC simulator for job dispatching research. Users can test different scheduling algorithms under a variety of workloads with high speed and minimal memory footprint. Its design principles, however, make it difficult to implement co-location schemes to analyze and test co-scheduling algorithms. CoSim [18] is an ad hoc simulator to analyze trade-offs between opportunistic on-demand jobs delayed and the loss of progress due to preemption of batch jobs. Co-scheduling is being explored in a time-sharing manner, where jobs switch on the same nodes in time intervals, rather than in a space-sharing manner, where jobs are co-executed while sharing those resources.

Toolkit based simulators: Batsim [10] is a general-purpose simulation tool built on top of SimGrid [9], a well-known, versatile toolkit designed for simulating distributed applications on various platforms. It enables seamless analysis and testing of custom and production-level scheduling algorithms using an event-driven communication protocol. It also supports multiple levels of realism for different research needs, by applying simple or complex models for various phenomena (e.g. energy consumption) to each component of the system. An extension of Batsim, IRMASim [15], further contributes by adding a memory contention model and a more in-depth characterization of the processors-cores architecture. ElastiSim [2] also uses SimGrid and studies scenarios with mal-

leable workloads. Both simulators benefit in accuracy by the network and I/O models provided by SimGrid but cannot emulate large-scale systems with multiple MPI applications executing concurrently using Simulated MPI (SMPI). This can be avoided by using the execution traces of applications collected from a workload. This approach is similar to ours, but we avoid differentiating the computation and communication details of a job, and thus abstain from defining and configuring extra details of the system architecture. The Alea [16] simulator, built on the extended GridSim 5 [7] toolkit, provides an environment for simulating diverse job scheduling problems and testing advanced techniques like local search and classical algorithms such as FCFS and EASY Backfilling. GridSim is a Java-based simulation toolkit for modeling and evaluating large-scale distributed systems like grids. It supports the simulation of heterogeneous resources, users, and applications, enabling performance analysis of scheduling algorithms and resource management in controlled, repeatable scenarios. Although Alea is a robust tool, the development of co-scheduling algorithms requires a comprehensive understanding of its complex architecture, making it difficult to expand.

Real RJMS based simulators: The Slurm Simulator [22] is a framework that wraps around the actual Slurm Workload Manager with the ability to provide a workload as input based on historical data. ScSF [19] is an extension of the Slurm Simulator for heterogeneous systems, that also improves the simulation's time. The main drawbacks of both frameworks are the limited freedom to implement new scheduling algorithms without using Slurm's interface and the inability to use more abstract representations for an HPC system.

3 ELiSE

3.1 Supported features

In the current version, ELiSE supports the following features regarding workloads, target platforms, scheduling algorithms, and output. Note that users can interact with ELiSE both with a graphical and a command-line interface, while independent simulations (e.g. for different scheduling algorithms) can be distributed to different physical machines.

Creating workloads: To generate a workload, users must provide the desired number of applications (jobs) N, an *application seed*, i.e. a set of n applications together with their execution times with compact allocation (*compact time*) and a n^2 co-execution matrix (*heatmap*), i.e. a matrix containing the speedups from the co-execution of each pair of applications. From that point on, ELiSE can generate a workload of N applications properly selecting representatives from the seed randomly, with user-defined frequencies, or with a specific list. Consequently, the user can define the interarrival time of the applications with available options being, constant, random, Poisson, and Weibull.

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

Available schedulers: The currently implemented schedulers contain a simple FCFS scheduler, FCFS with EASY backfill, FCFS with conservative Backfill [23], FCFS co-scheduler with EASY Backfill, and a smart co-scheduler (called Filler). The latter two are described in Section 3.6. One of the main features of ELiSE is its straightforward extension to new schedulers which is also described in the same section.
Output: ELiSE provides as output a detailed log of the workload's execution, including the arrival, start, and end times of each job and more, as well as different visualizations including Gantt charts, system throughput, queue size, system utilization as a function of time, application speedups in boxplots and others. Indicative outputs of ELiSE are shown in Figure 3.

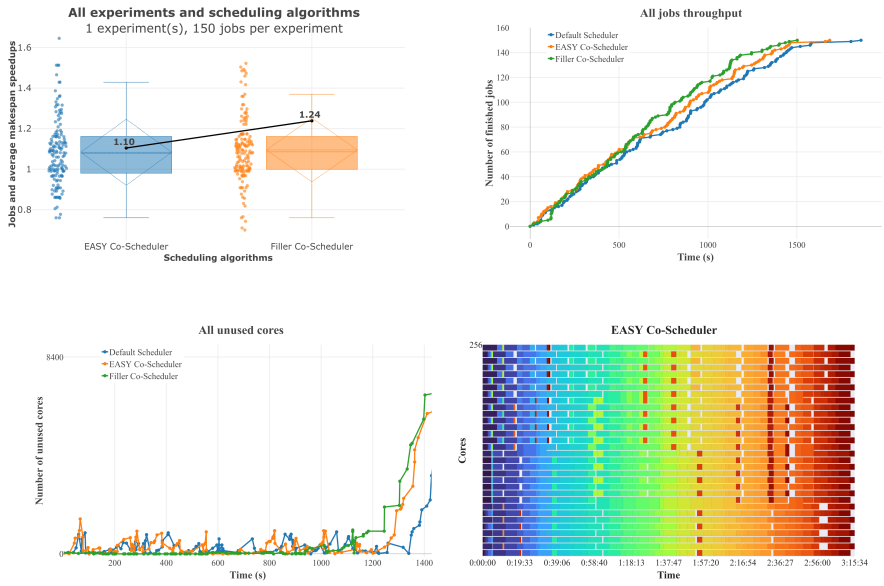

Fig. 3: Visualization features of ELiSE that help determine various scheduling factors such as utilization via the Gantt chart

3.2 Design principles

Before delving into more details on the internals of ELiSE, we discuss here some design principles. First, the mission and focus of ELiSE is to assess different scheduling policies with a special focus on co-scheduling, and as such it is not a full system simulator. It utilizes results from executions and pairwise executions (co-executions) of jobs. These results can arise from real experiments, partial experiments and interpolations, simulations, models, and any other possible

process that is considered orthogonal and complementary to ELiSE. This separation of responsibilities leads to a rather simple design that avoids the need for over-configuration and tuning, ultimately enabling rapid development and testing of scheduling and co-scheduling algorithms. Another important design point is the parallel nature of ELiSE. A simulation run can host multiple workloads and schedulers that are executed in parallel thus returning faster the results for analysis.

3.3 High-level logic

The simulator resembles a finite state machine with jobs moving across four states as shown in Figure 4: The *Future* state contains jobs that have been generated according to user requests but their arrival time has not been reached by the simulator yet, *Waiting* contains jobs that have arrived and wait to be executed in the system, *Executing* are those that are currently running in the cluster, and *Finished* are those that have exited the system. The *Scheduler* decides which job to move from the waiting queue to the running state and which nodes/cores of the cluster to grant to the job. Further simulation logic described in the next sections orchestrates the rest of the process, i.e. handling future jobs, advancing time, calculating finishing times, etc.

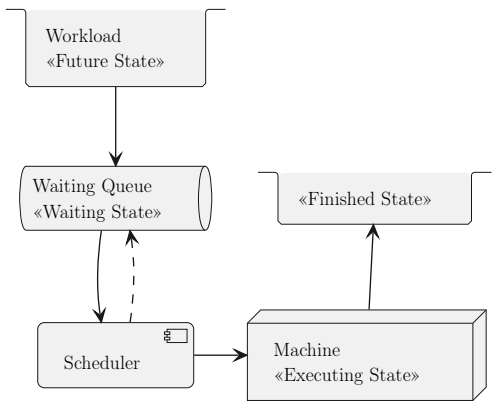

Fig. 4: High-level design logic of the framework

3.4 Life-cycle of a simulation step

The life-cycle of a simulation step of ELiSE is depicted in Figure 5. The process repeats while there are still jobs in any of the future, waiting and executing states, and in each pass, it moves jobs appropriately between the various states (according to the scheduling criteria), calculates the remaining time of each executing job, and the simulation time step. The latter is computed by selecting

the minimum of a) the arrival time of each future job, b) the starting time of each waiting job, and c) the remaining execution time of each executing job.

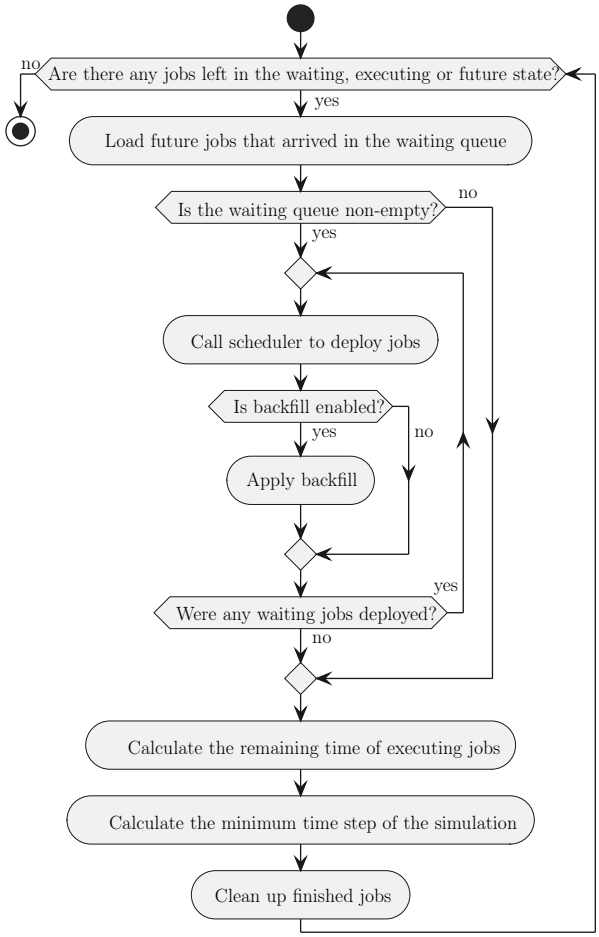

Fig. 5: Life-cycle of a simulation step

3.5 Adaptation for co-scheduling

As the main motivation for ELiSE is to shed light on the behavior of co-scheduling, we discuss in this paragraph the appropriate adaptations to this direction, that mainly affect the calculation of the remaining execution time of each executing job. Recall that the co-location of applications within the same node potentially changes the throughput of each application and thus may lead to different execution times, higher or lower than the ones provided by user input (compact allocation time). Note also that: a) during its execution lifetime an application may have different neighbors that enter and exit the neighboring

cores within the node and b) at a given time, an application may have different neighbors in the various nodes that it has been allocated to.

For the second case, we make the reasonable assumption that the execution time of application A when concurrently co-located with applications B and C (in different nodes) is determined by the slowest case, since processes in HPC applications typically interact with each other in time steps, and the slowest process determines the pace of the entire execution. Thus, for each executing job, the framework returns all its neighboring jobs and then calculates the minimum pair-wise speedup from the relevant heatmap provided by the user. The new remaining execution time of each job is computed using the following formula:

$$remExecTime_{new} = \frac{speedup_{old} \times remExecTime_{old}}{speedup_{new}}, where$$

$$speedup_{new} = \min_{\forall co-scheduled(job')} \{getSpeedupWith(job')\}$$

The expression $speedup_{old} \times remExecTime_{old}$ transforms the current remaining execution time of the job to the equivalent remaining time if it was running isolated with a compact allocation policy. The compact execution time is used as the reference execution time for any co-execution. The above rationale is repeated in each simulation step and is adapted to the neighbors of the application that may change in time. Note, finally, that ELiSE includes all the mechanisms to support beyond pairwise co-locations, however, since this requires more advanced schedulers and speedup predictions, further exploration is left for future work.

3.6 Adding a new scheduler

In ELiSE , all scheduling algorithms create a class hierarchy as shown in Figure 6. Algorithm 1 describes the main operations a scheduler performs, which is the abstract root class of the hierarchy. It provides methods for allocating, co-locating, deploying jobs, and backfilling jobs, together with reordering the waiting queue. The two abstract methods, **deploy** and **backfill**, must be defined for the scheduling algorithm to be usable by ELiSE. All the well-known scheduling algorithms such as FCFS, Shortest-Job First (SJF), EASY backfill, conservative backfill, and more, can be implemented using this interface as a base. We have followed this scheme to implement a straightforward co-scheduler that pairs jobs in an FCFS order and supports EASY backfilling. Algorithm 2 presents the *Filler Co-Scheduler* that inherits the backfill method from the EASY Co-Scheduler. As illustrated, we define **waiting_queue_reordering** which breaks the arrival order of jobs in an attempt to increase system utilization.

4 Evaluation

In this section, we proceed with the evaluation of ELiSE which is split into three parts: in the first part, we validate the results of ELiSE by comparing them with

Algorithm 1: Interface of the Scheduler abstract class

```
class Scheduler is
    def host_alloc_condition(hostname : str, job : Job) → orderingnumber:
    ...
    def allocation(job : Job, socket_cores_binding : tuple) → req_fulfilled:
    ...
    def waiting_queue_reordering(job : Job) → orderingnumber:
    ...
    abstract def deploy() → boolean:
    ...
    abstract def backfill() → boolean:
    ...
end
```

Algorithm 2: Filler Co-Scheduler

```
class Filler Co-Scheduler inherits from EASY Co-Scheduler is
    def waiting_queue_reordering(job : Job) → orderingnumber:
        sys_free_cores ← cluster.get_idle_cores();
        if sys_free_cores > 0 then
            diff ← sys_free_cores − job.num_of_processes;
            if diff ≥ 0 then
                factor0 ← 1 − diff / sys_free_cores;
            else
                factor0 ← −1;
            end
        else
            factor0 ← 1;
        end
        factor1 ← (job.Id+1) / maxId;
        return factor0 / factor1;
    end
    def deploy() → boolean:
        deployed_any_jobs ← false;
        waiting_queue′ ← copy(waiting_queue);
        waiting_queue′.descending_sort(by waiting_queue_reordering());
        while waiting_queue′ is not empty do
            job ← get_head(waiting_queue′);
            request ← allocation(job, half_socket_binding);
            if request can be fulfilled then
                dispatch(job);
                deployed_any_jobs ← true;
            else
                break;
            end
        end
        return deployed_any_jobs;
    end
end
```

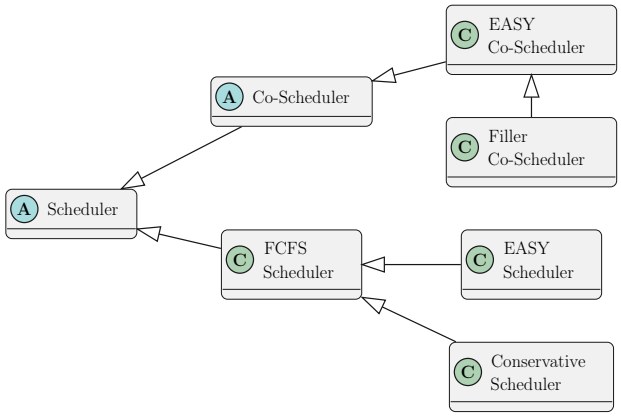

Fig. 6: Hierarchical view of the Schedulers as implemented in ELiSE. Abstract and Concrete classes are represented as *A* and *C*, respectively

those of a real, small-scale execution. In the second part, we utilize ELiSE to collect initial results on co-scheduling and draw some preliminary conclusions. In the third part, we provide results on the simulation time of ELiSE.

4.1 Experimental Setup

HPC systems utilized: For our experiments, we used three systems as shown in Table 1 with their specifications. We have collected pairwise co-scheduling results from these systems and fed them to ELiSE, while we additionally use Grid5000-Grvingt for our validation step.
Benchmarks: We used the NPB suite as representative applications for HPC systems. We have constructed application seeds using jobs of 64, 128, 256, 512, and 1024 process counts from the D and E classes of NPB.
Evaluation metrics: We explain herein the various metrics that are used in the remainder of this section. *Makespan* is the total time required to complete a set of jobs, starting from the submission of the first job to the completion of the last one. *Job turnaround time* is the total time taken for a job, from submission to completion, i.e. the sum of the waiting time and the execution time. *Job speedup* is the ratio of the baseline compact execution time to the execution time of the job when co-scheduled. *System utilization* refers to the percentage of the system that is in use until the waiting queue is empty.

4.2 Validation

The first question we intend to answer is whether ELiSE provides simulation results close to reality. We test ELiSE against the results collected from OAR 3 [8], which we extended to feature a co-location mechanism[2]. We deployed the

[2] https://github.com/hidden_for_blind_review

Table 1: HPC Clusters Specifications

	ARIS
CPU Type	Intel Xeon E5-2680v2 (Ivy Bridge), 2.8 GHz
CPUs per Node	2
Cores per CPU	10
Cores per Node	20
Hyperthreading	OFF
Memory per Node	64GB
Network	Infiniband FDR, 56 Gb/s
	Marconi
CPU Type	Intel Xeon 8160 (SkyLake), 2.10 GHz
CPUs per Node	2
Cores per CPU	24
Cores per Node	48
Hyperthreading	OFF
Memory per Node	196 GB
Network	Intel Omni-Path, 100 Gb/s
	Grid5000 (Grvingt)
CPU Type	Intel Xeon Gold 6130, 2.10 GHz
CPUs per Node	2
Cores per CPU	16
Cores per Node	32
Hyperthreading	OFF
Memory per Node	192 GiB
Network	Intel Omni-Path, 100 Gb/s

extended OAR on a sub-cluster of eight nodes in Grid5000-Grvingt. For this purpose, we used NixOS-Compose [13] to set up ephemeral distributed systems thanks to Nix functional package manager and Nixos. For our evaluation, we deployed the modified version of OAR using the g5k-nfs-store flavor which can be presented as an adaptation of the classic nfs-root, where in NixOS is referred to as a configuration that allows a system to boot from a Network File System (NFS) share. We tested two scenarios, one for simple FCFS scheduling and one for co-scheduling, discussed in the next paragraphs.

FCFS validation For our evaluation, we conducted six experiments. Two experiments were of 48 jobs each with a makespan of about one hour. The rest of the experiments were of 136 jobs each, where two of them had a makespan of about two to three hours, and the other two had a makespan of about three to four hours. In Figure 7 we present a comparison of the per job turnaround time between real execution (OAR) and simulated execution (ELiSE) together with the Mean Absolute Percentage Error (MAPE) both for the per job turnaround time and the total experiment makespan. In all cases, we observe that the sim-

ulated and actual times are close and any differences can be attributed to differences in the execution times (note that ELiSE utilizes execution times from prior executions that may be slightly different from subsequent ones), system, and scheduler overheads that are not accounted for in ELiSE.

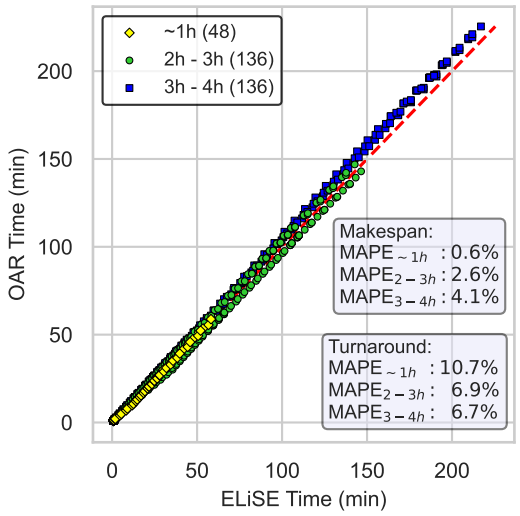

Fig. 7: Per job turnaround times. Each point corresponds to the turnaround time of one job

Co-Scheduling validation In this experiment, we extended OAR to enable a co-scheduler with a conservative backfill mechanism. We created six mixes with varying sizes and "co-scheduling friendliness". We empirically characterize a workload seed as co-scheduling (un)friendly if its heatmap has a high (low) mean job speedup. Figure 8 shows the makespan of five different shuffles for the six size/friendliness workloads. Again in this simple co-scheduling case, we observe that the simulated and real results are very close with the MAPE not exceeding 15% in the worst case.

4.3 Initial observations on the behavior of co-scheduling

In our view, ELiSE unleashes a great potential to study the behavior of co-scheduling, towards ultimately guiding its implementation and deployment in a production system. In this section, we show how by using ELiSE, we made some important initial observations. Firstly, we observed that the diversity in the process count of a workload mix had a high impact on the benefits of co-scheduling. We experimented with four types of workloads of 500 jobs each from the ARIS

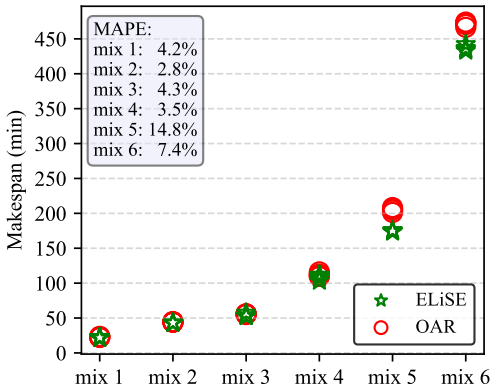

Fig. 8: Makespans of OAR Conservative Co-Scheduler with regards to ELiSE EASY Co-Scheduler

system. Each workload type consisted of jobs with an increasing diversity of process counts, whereas each experiment was conducted shuffled four times. Figure 9 shows the improvement of the non-sophisticated EASY Co-Scheduler over the default EASY scheduler. At first, we observe that when dealing only with jobs requesting 256 processes, EASY co-scheduling reaches a makespan improvement of almost 12.6% which designates a significant throughput speedup attributed to co-scheduling. However, as the workload is becoming more diverse in process counts, naive co-scheduling struggles to keep its benefit, actually losing it when facing the entire process diversity. With the help of ELiSE's visualization tools, we were able to easily understand that this was due to low utilization, as naive co-scheduling led to fragmentation and unused system resources. We then implemented the Filler Co-Scheduler that crudely fills in unutilized resources with the best-fit waiting jobs. Filler is able to maintain the co-scheduling improvements, at the cost however, of breaking the FCFS principle of the EASY schedulers, thus actually being unfair. For the moment, however, we just need to illustrate the capabilities of ELiSE and not propose new co-scheduling algorithms. We leave for future work the detailed experimentation of the co-scheduling behavior and the evaluation of new algorithms that can be simultaneously fair and lead to high performance.

Table 2 demonstrates the high makespan improvements of workloads containing jobs with equal process counts and with varying mean 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 mean speedup case, we conducted four shuffled experiments on a cluster with 100 nodes and 48 cores. These results lead us to two observations: First, we note that the improvement by co-scheduling in this favorable case is rather impressive as it reaches almost 31%. Second, there is a correlation between the mean speedup of all job pairs in an application pool and the makespan improvement achieved

through co-scheduling, a fact that is not surprising since accelerated jobs are expected to lead to makespan improvements.

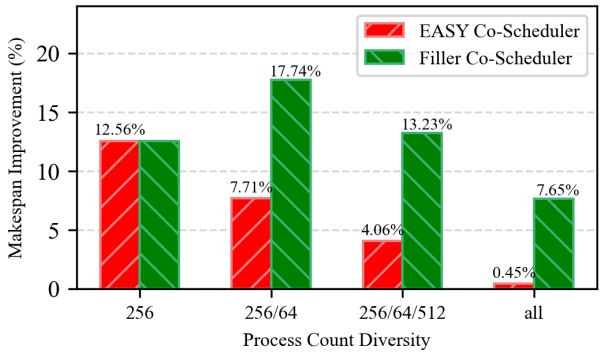

Fig. 9: Makespan improvement for two co-scheduling algorithms with regards to the EASY Scheduler and for diverse process count mixes

Table 2: Makespan Improvement by utilizing the EASY Co-Scheduler with regards to the mean job speedup of the used sub-heatmap

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

We have seen so far that good system utilization and job speedups can lead to makespan (system throughput) improvement when co-scheduling is employed. However, a system scheduler needs to also consider metrics that express user satisfaction. In the case of co-scheduling, we argue that if such a scheme is activated and some users experience execution slowdowns, administrators should be aware of this fact. We conducted 100 experiments for a particular application pool using ELiSE, the heatmap from the ARIS supercomputer, and a 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. Figure 10 illustrates the makespan improvement and its correlation with mean job speedup and system utilization. While both metrics exhibit a clear trend, job speedups show a more significant impact, at least in the context of this specific experiment.

These observations can inform future algorithmic design. Additionally, the lower section of the figure highlights that varying job shuffles resulted in a

wide range of outcomes for the EASY Co-Scheduler across two critical metrics: makespan improvement (system-wise) and the percentage of jobs experiencing slowdowns relative to compact execution (user satisfaction). This variability further emphasizes the need for a more advanced co-scheduling algorithm.

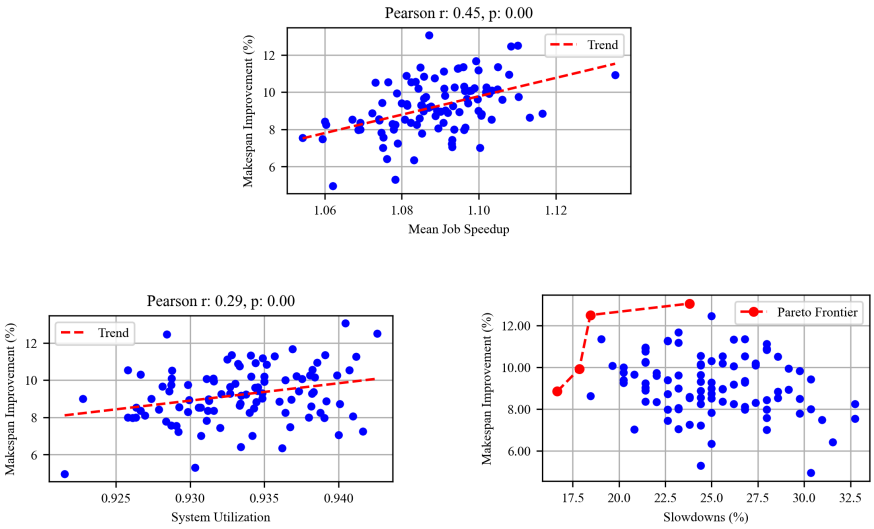

Fig. 10: Correlation between Makespan Improvement by utiziling the EASY Co-Scheduler and various metrics related to system throughput and user satisfaction

4.4 Simulation time

Tables 3 and 4 summarize the simulation time as the number of execution days per simulation hour for synthetic workloads and real workloads taken from the Parallel Workload Archive (PWA) [11]. We may observe that ELiSE can be utilized to simulate small and medium cases extremely fast and very large cases at tolerable times, i.e. in the worst case of ELiSE, we can indicatively simulate 10k jobs on a system of 2.5M cores for 14 hours of makespan time in one hour of simulation time. The results were obtained on a desktop machine equipped with an AMD Ryzen 5 1600 Six-Core Processor and 16GB of RAM.

5 Conclusion and Future Work

This paper presented a comprehensive simulation framework for evaluating co-scheduling techniques in HPC systems. We have demonstrated the effectiveness of our framework by analyzing the performance of various scheduling and co-scheduling algorithms under diverse workload and system configurations. Our

Table 3: Days of makespan time per simulation hour (synthetic workloads)

FCFS Scheduler

Workload Size	**Small Cluster** 256 cores	**Medium Cluster** 25600 cores	**Large Cluster** 2560000 cores
100	2038.66	147.75	226.52
1000	838.74	14.31	12.56
10000	45.84	0.58	0.83

EASY Scheduler

Workload Size	**Small Cluster** 256 cores	**Medium Cluster** 25600 cores	**Large Cluster** 2560000 cores
100	900.98	297.82	240.10
1000	295.04	12.77	10.34
10000	20.07	0.63	0.59

Table 4: Days of makespan time per simulation hour (workloads from PWA)

Workload	**FCFS Scheduler**	**EASY Scheduler**
SDSC-SP2-1998 (128 nodes, 64 ppn, 73496 jobs)	116.87	117.77
KIT-FH2-2016 (1152 nodes, 20 ppn, 114335 jobs)	28.34	27.19

results also highlight the high potential of co-scheduling to improve system performance and resource utilization. For future work, we plan on focusing on three main axes; expanding modeling capabilities, enhancing user experience and improving the performance and scalability of the framework.

Expanding modeling capabilities: We plan to generalize co-scheduling to capture more co-location scenarios. We also intend to include detailed models for power consumption, dissipation, and I/O-bound applications. Furthermore, we intend to support a wider range of workload types, incorporating more adaptive techniques to simulate real-world user activity. Lastly, we also plan to introduce models for host failures and recovery mechanisms, to simulate fault-tolerant systems. These inclusions to the framework will allow us to delve deeper into the intricacies of co-scheduling and explore a wider range of use cases and scenarios.

Enhancing user experience: We intend to further develop user-friendly tools for configuring and automating simulations. This includes automating parameter tuning for scheduling algorithms to streamline optimization analysis. Additionally, we plan to enrich the framework with a broader range of visualizations and plots to provide deeper insights into simulation results.

Improving performance: We are committed to further optimizing the framework's scalability to efficiently handle large-scale HPC simulations.

References

1. Ahn, D.H., Garlick, J., Grondona, M., Lipari, D., Springmeyer, B., Schulz, M.: Flux: A next-generation resource management framework for large hpc centers. In: 2014 43rd International Conference on Parallel Processing Workshops. pp. 9–17. IEEE (2014)
2. Almaaitah, N.O., Singh, D.E., Özden, T., Carretero, J.: Performance-driven scheduling for malleable workloads. The Journal of Supercomputing pp. 1–29 (2024)
3. Blanche, A.d., Lundqvist, T.: Terrible twins: A simple scheme to avoid bad co-schedules. In: Proceedings of the 1st COSH Workshop on Co-Scheduling of HPC Applications. p. 25 (2016)
4. Breitbart, J., Weidendorfer, J., Trinitis, C.: Case study on co-scheduling for hpc applications. In: 2015 44th International Conference on Parallel Processing Workshops. pp. 277–285. IEEE (2015)
5. Breitbart, J., Weidendorfer, J., Trinitis, C.: Automatic co-scheduling based on main memory bandwidth usage. In: Desai, N., Cirne, W. (eds.) Job Scheduling Strategies for Parallel Processing. pp. 141–157. Springer International Publishing, Cham (2017)
6. Breslow, A.D., Porter, L., Tiwari, A., Laurenzano, M., Carrington, L., Tullsen, D.M., Snavely, A.E.: The case for colocation of high performance computing workloads. Concurrency and Computation: Practice and Experience **28**(2), 232–251 (2016)
7. Buyya, R., Murshed, M.: Gridsim: a toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. Concurrency and Computation: Practice and Experience **14** (2002)
8. Capit, N., Da Costa, G., Georgiou, Y., Huard, G., Martin, C., Mounié, G., Neyron, P., Richard, O.: A batch scheduler with high level components. In: CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005. vol. 2, pp. 776–783. IEEE (2005)
9. Casanova, H.: Simgrid: A toolkit for the simulation of application scheduling. In: Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid. pp. 430–437. IEEE (2001)
10. Dutot, P.F., Mercier, M., Poquet, M., Richard, O.: Batsim: a realistic language-independent resources and jobs management systems simulator. In: Job Scheduling Strategies for Parallel Processing: 19th and 20th International Workshops, JSSPP 2015, Hyderabad, India, May 26, 2015 and JSSPP 2016, Chicago, IL, USA, May 27, 2016, Revised Selected Papers 19. pp. 178–197. Springer (2017)
11. Feitelson, D.G., Tsafrir, D., Krakov, D.: Experience with using the parallel workloads archive. vol. 74, pp. 2967–2982 (2014). https://doi.org/https://doi.org/10.1016/j.jpdc.2014.06.013
12. Galleguillos, C., Kiziltan, Z., Netti, A., Soto, R.: Accasim: a customizable workload management simulator for job dispatching research in hpc systems. Cluster Computing **23**(1), 107–122 (2020)
13. Guilloteau, Q., Bleuzen, J., Poquet, M., Richard, O.: Painless transposition of reproducible distributed environments with nixos compose. In: 2022 IEEE International Conference on Cluster Computing (CLUSTER). pp. 1–12. IEEE (2022)

14. Hall, J., Lathi, A., Lowenthal, D.K., Patki, T.: Evaluating the potential of coscheduling on high-performance computing systems. In: Workshop on Job Scheduling Strategies for Parallel Processing. pp. 155–172. Springer (2023)
15. Herrera, A., Ibáñez, M., Stafford, E., Bosque, J.: Irmasim: a simulator for intelligent workload managers in heterogeneous clusters (04 2023)
16. Klusáček, D., Tóth, Š., Podolníková, G.: Complex job scheduling simulations with alea 4. In: Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques. pp. 124–129 (2016)
17. Mars, J., Tang, L., Hundt, R., Skadron, K., Soffa, M.L.: Bubble-Up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In: Proceedings of the Annual International Symposium on Microarchitecture, MICRO. pp. 248–259 (dec 2011). https://doi.org/10.1145/2155620.2155650
18. Maurya, A., Nicolae, B., Guliani, I., Rafique, M.M.: Cosim: A simulator for co-scheduling of batch and on-demand jobs in hpc datacenters. In: 2020 IEEE/ACM 24th International Symposium on Distributed Simulation and Real Time Applications (DS-RT). pp. 1–8. IEEE (2020)
19. Rodrigo, G.P., Elmroth, E., Östberg, P.O., Ramakrishnan, L.: Scsf: A scheduling simulation framework. In: Job Scheduling Strategies for Parallel Processing: 21st International Workshop, JSSPP 2017, Orlando, FL, USA, June 2, 2017, Revised Selected Papers 21. pp. 152–173. Springer (2018)
20. Rodrigues, A.F., Hemmert, K.S., Barrett, B.W., Kersey, C., Oldfield, R., Weston, M., Risen, R., Cook, J., Rosenfeld, P., Cooper-Balis, E., et al.: The structural simulation toolkit. ACM SIGMETRICS Performance Evaluation Review **38**(4), 37–42 (2011)
21. Saroliya, U., Arima, E., Liu, D., Schulz, M.: Hierarchical resource partitioning on modern gpus: A reinforcement learning approach. In: IEEE International Conference on Cluster Computing, CLUSTER 2023, Santa Fe, NM, USA, October 31 - Nov. 3, 2023. pp. 185–196. IEEE (2023). https://doi.org/10.1109/CLUSTER52292.2023.00023, https://doi.org/10.1109/CLUSTER52292.2023.00023
22. Simakov, N.A., Innus, M.D., Jones, M.D., DeLeon, R.L., White, J.P., Gallo, S.M., Patra, A.K., Furlani, T.R.: A slurm simulator: Implementation and parametric analysis. In: High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 8th International Workshop, PMBS 2017, Denver, CO, USA, November 13, 2017, Proceedings 8. pp. 197–217. Springer (2018)
23. Srinivasan, S., Kettimuthu, R., Subramani, V., Sadayappan, P.: Characterization of backfilling strategies for parallel job scheduling. In: Proceedings. International Conference on Parallel Processing Workshop. pp. 514–519. IEEE (2002)
24. Xu, R., Mitra, S., Rahman, J., Bai, P., Zhou, B., Bronevetsky, G., Bagchi, S.: Pythia: Improving datacenter utilization via precise contention prediction for multiple co-located workloads. In: Proceedings of the 19th International Middleware Conference, Middleware 2018. pp. 146–160. Association for Computing Machinery, Inc, New York, New York, USA (nov 2018). https://doi.org/10.1145/3274808.3274820, http://dl.acm.org/citation.cfm?doid=3274808.3274820
25. Yoo, A.B., Jette, M.A., Grondona, M.: Slurm: Simple linux utility for resource management. In: Workshop on job scheduling strategies for parallel processing. pp. 44–60. Springer (2003)
26. Zacarias, F.V., Petrucci, V., Nishtala, R., Carpenter, P., Mossé, D.: Intelligent colocation of hpc workloads. Journal of Parallel and Distributed Computing **151**, 125–137 (2021)