POLITECNICO DI TORINO
Repository ISTITUZIONALE

Design and Evaluation of Vertical Scalability Strategies for Deploying Large-scale Co-simulations

Original
Design and Evaluation of Vertical Scalability Strategies for Deploying Large-scale Co-simulations / Valeriano, C.G., Estebsari, A., Bottaccioli, L., Patti, E., Schiera, D.S., Barbierato, L.. - (2025), pp. 2046-2052. (2025 IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC) Toronto (CAN) July 8-11, 2025) [10.1109/compsac65507.2025.00286].

Availability:
This version is available at: 11583/3002675 since: 2025-09-01T11:19:52Z

Publisher:
IEEE

Published
DOI:10.1109/compsac65507.2025.00286

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2025 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

27 June 2026

Design and Evaluation of Vertical Scalability Strategies for Deploying Large-scale Co-simulations

Carlos Gerardo Valeriano*, Abouzar Estebsari‡, Lorenzo Bottaccioli*,
Edoardo Patti*, Daniele Salvatore Schiera*, Luca Barbierato*
*Politecnico di Torino, Turin, Italy. Email: name.surname@polito.it
‡London South Bank University, London, United Kingdom. Email: estebsaa@lsbu.ac.uk

***Abstract*—Multi-Energy Systems (MES) represent a paradigm in which various energy systems, such as buildings, power grids, and heating networks, are integrated to operate in a cohesive manner. These systems provide a significant opportunity to improve technical, economic, and environmental outcomes. However, the complexity of MES poses challenges in accurately capturing their interdisciplinary interactions using standalone simulations, which often do not adequately model their interconnected nature. Co-simulation frameworks have emerged as a promising solution to this challenge, enabling the coordination of multiple simulators within a single scenario. However, computationally intensive simulators can often hinder scalability, especially as the complexity of the simulations increases. This paper proposes a methodology to apply vertical scalability techniques to simulators and optimise each single-node performance in co-simulation frameworks. Using COESI, a Mosaik-based co-simulation platform, it was possible to systematically test these techniques by increasing the number of model entities a simulator class must manage in a complex MES scenario. This study highlights selective parallelisation as a key strategy for optimising computational efficiency in MES co-simulation, offering insights to scale complex energy systems effectively. The results demonstrate significant performance gains, particularly for CPU-intensive tasks with stateless simulators, such as executing intensive numerical computations.**

***Index Terms*—Multi-Energy Systems, Co-Simulation Performance, Vertical Scaling Techniques, Computational Efficiency**

I. Introduction

The urgent need to mitigate climate changes and address energy security challenges has placed energy systems at the forefront of global sustainability efforts. Multi-Energy Systems (MES) have emerged as a key concept to achieve efficient, sustainable, and resilient energy infrastructures by integrating traditionally independent systems, such as electrical grids, heating networks, cooling systems, and transport [1]. MES offer the potential to dynamically balance demand and supply, optimise resource utilisation, and reduce overall system costs and environmental impacts. However, the inherent complexity of MES, driven by their interdisciplinary nature and tightly coupled components, poses significant challenges to their modelling and simulation.

While effective in modelling isolated systems, traditional standalone simulation approaches fail to capture the intricate interdependencies and dynamic interactions that define MES. For instance, the operation of a building's heating system cannot be fully understood without accounting for its interaction with the power and heating networks or local Renewable Energy Source (RES). This oversimplification limits the accuracy and relevance of simulation results, making standalone methods inadequate for designing and optimising modern energy systems.

Co-simulation frameworks, such as Mosaik [2] and HELICS [3], have been developed to address this gap by integrating diverse simulation tools within a unified environment. These frameworks provide a platform for holistic analysis, allowing domain-specific simulators to interact while preserving their unique capabilities. Despite their promise, co-simulation frameworks bring their own set of challenges. One is the issue of computational performance, as the need to coordinate multiple simulators often leads to scalability bottlenecks. To enhance scalability in this context, researchers have proposed various coupling techniques. For example, Li et al. [4] introduced relaxation methods for explicit co-simulation, improving the stability and convergence of simulations that involve disparate subsystems. Similarly, hierarchical co-simulation, as described by Gomes [5], structures complex simulations into layers, optimising the management of synchronisation and performance. This hierarchical approach can enhance both scalability and accuracy by orchestrating the interaction between models more effectively.

Another way to enhance scalability is using dynamic evaluation techniques and tunable models, which balance accuracy and speed. This approach enables designers to rapidly explore different architectural configurations. Lajolo et al. [6] proposed methods for dynamically evaluating system performance at different abstraction levels, allowing co-simulation to adjust its fidelity during runtime. This flexibility is especially valuable in large-scale simulations, where trade-offs between speed and accuracy are crucial for efficient operation.

Scalability bottlenecks pose significant challenges when working with computationally intensive simulators in large-

This work is supported by i) COMET, Italian National funded project under PNRR M4C2, Investimento 1.4 - Avviso n. 3138 del 16/12/2021 - CN00000013 National Centre for HPC, Big Data and Quantum Computing; and ii) NEST, Project code PE0000021, Concession Decree No. 1561 of 11.10.2022 adopted by Ministero dell'Università e della Ricerca (MUR), CUP E13C22001890001. Project funded under the National Recovery and Resilience Plan (NRRP), Mission 4 Component 2 Investment 1.3 - Call for tender No. 341 of 15.03.2022 of MUR; funded by the European Union – NextGenerationEU.

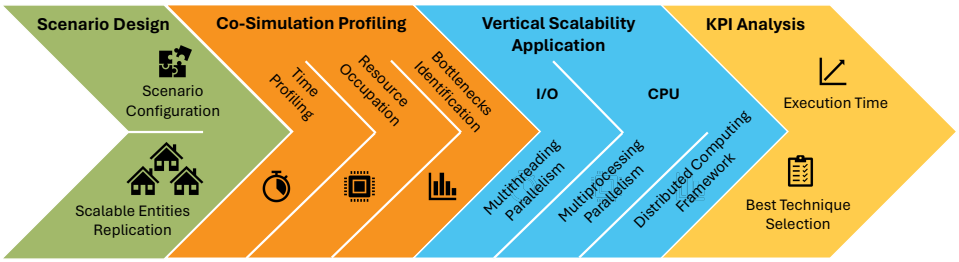

Fig. 1: The methodology flowchart illustrating the four-step process.

scale scenarios. Inefficient resource utilisation can lead to extended execution times and diminish the feasibility of real-world applications. Vertical scalability is recommended to address this issue. This approach involves enhancing the capacity of a single machine by optimising the use of existing resources. In co-simulation, vertical scaling can be achieved through parallelisation techniques such as multiprocessing or multithreading, which fully utilise the CPU cores and memory of a computing node. These methods enable multiple simulations or tasks to run concurrently on a single machine, enhancing efficiency without additional hardware. Vialle [7] illustrated the effectiveness of parallelisation in co-simulation by scaling up to over 1000 Functional Mock-up Units (FMUs) across multi-core architectures, achieving notable improvements in computational performance and simulation speed.

To address these challenges, this paper focuses on three key Research Challenges (**RC1**, **RC2** and **RC3**, respectively). **RC1** is optimising simulation performance in co-simulations. This involves managing the computational demands of diverse simulators while maintaining stability and minimising execution overhead. **RC2** is related to resource utilisation, as co-simulations often suffer from imbalances in CPU, memory, and I/O workloads across simulators, leading to suboptimal efficiency. Finally, **RC3** explores the applicability of optimisation techniques, acknowledging that constraints such as simulator statefulness and reliance on third-party libraries can limit the effectiveness of universal approaches. By systematically addressing these challenges, this work aims to advance the scalability and efficiency of co-simulation frameworks, focusing on MES applications. Using the COESI platform [8], a Mosaik-based co-simulation tool, this study evaluates vertical scaling techniques — including multiprocessing, multithreading, and a distributed computing framework within a complex MES scenario. The findings demonstrate the transformative potential of tailored parallelisation strategies in optimising computational efficiency, enabling the simulation of large-scale energy systems with greater precision and speed.

The rest of the paper is organised as follows: Section II outlines the methodology. Section III describes the simulation scenario. Section IV presents the experimental results. Finally, Section V provides concluding remarks and future research.

II. Methodological Framework

The methodology adopted in this research follows a systematic four-step process, as shown in Fig. 1, enabling the evaluation of different vertical scalability techniques for each simulator involved in the co-simulation. This approach guarantees that combining simulators and their corresponding scalability techniques ensures scalability and efficiency in large-scale co-simulations. Each phase addresses specific challenges of designing, profiling, and optimising a co-simulation scenario while focusing on vertical scalability challenges. The following sections describe each step in Fig. 1.

A. Scenario Design

The *Scenario Design* is crucial for addressing **RC1**, as a well-designed scenario setup ensures that performance bottlenecks are minimised from the beginning, allowing the system to handle increasing computational loads efficiently and scale appropriately. This process is achieved by following two steps: *i)* the *Scenario Configuration*, and *ii)* the *Scalable Entities Replication*.

The *Scenario Configuration* involves setting up the energy scenario through the COESI configuration file, which is a well-defined structured schema based on the YAML standard, which contains all the information needed to set up the entire co-simulation environment as well as the individual integrated model [9]. This approach simplifies the composition of scenarios by using the default settings of the objects and focusing only on changing the desired parameters in a plug-and-play way and automates the scenario compiling process from the Mosaik orchestrator. This stage is critical because it defines the simulation scenario and establishes connections between the simulators and their respective models. Indeed, Mosaik must instantiate simulators to ensure proper functionality, considering their operational parameters, dependencies, and data exchange requirements. The scenario also outlines how these simulators will interact throughout the simulation. For a scenario to be scalable, the COESI setup must incorporate a mechanism that allows the simulators to manage an adjustable number of entities effectively. These entities, representing individual model instances within a simulator, are the core components for testing scalability. By utilising configuration parameters or scripting techniques, the number of entities per simulator can be dynamically modified during execution. This approach provides flexibility and prepares the system for scalability experiments in later stages.

After completing the Scenario Configuration, the simulators must be modified to accommodate a scalable number of entities. The process of *Scalable Entities Replication* guarantees

that replicating entities is straightforward and computationally efficient. A typical scalable entity setup utilises parameterisation within the simulator's configuration code, allowing users to easily specify the desired number of entities. This modular approach eliminates the need to create entities manually, reducing the possibility of human error and ensuring a smooth transition when testing various scenario sizes.

B. Co-simulation Profiling

This step directly addresses **RC2**, as it ensures that CPU, memory, and I/O workloads are effectively distributed, preventing inefficiencies that could hinder scalability. It is composed of three parallel stages: *i) Time Profiling*, *ii) Resource Occupation*, and *iii) Bottlenecks Identification*.

With the scenario in place, the *Time Profiling* step focuses on profiling the base co-simulation setup. The base scenario is run with the minimum number of entities per simulator (i.e. one entity) to identify performance bottlenecks without the influence of scalability factors. Time profiling is performed by recording the execution time of each simulator within the overall simulation. This profiling offers insights into the time contribution of each simulator relative to the total simulation runtime. Simulators that take up a disproportionate amount of execution time are flagged for further analysis, as they may pose challenges when scaled up.

In addition to Time Profiling, the *Resource Occupation* profiling metrics, including CPU and memory usage, are monitored for each simulator. These metrics help identify simulators that may consume excessive computational or memory resources, hindering vertical scalability. For example, a simulator that performs intensive I/O operations, such as frequent file read/write or network communication, might show high resource usage even with a small number of entities.

Time Profiling and Resource Occupation analysis results are processed in the *Bottlenecks Identification* to select simulators that could be potential bottlenecks for the overall co-simulation execution. Simulators with high time consumption or resource usage are evaluated with an increased number of entities to assess their scalability limits. If the runtime or resource usage increases disproportionately as the entity count rises, these simulators will receive further attention in the next phase, where suitable scalability techniques will be implemented.

C. Vertical Scalability Application

This step tackles **RC3**, evaluating different scalability methods while considering real-world constraints such as simulator statefulness and dependencies on third-party libraries. Vertical scalability strategies can be effectively applied across various programming languages, including Python, Java, and C++, and various simulation tools such as MATLAB Simulink, Modelica, and EnergyPlus with FMU encapsulation. The methodology tests multiple optimisation techniques and determines which approaches are best suited for co-simulation frameworks applying three main vertical scalability strategies: *i) Multithreading Parallelism*, *ii) Multiprocessing Parallelism*, and *iii) Distributed Computing Frameworks*.

In scenarios where simulators experience bottlenecks due to I/O-bound operations, *Multithreading Parallelism* is a compelling strategy applicable across various programming languages. Conversely, this approach is not beneficial for simulation tools that typically depict the physical dynamics of a model, as it primarily impacts CPU-bound operations. This strategy enables the concurrent execution of multiple threads, thereby minimising idle wait times associated with I/O operations. In Python, the `concurrent.futures` library offers a high-level interface for efficiently managing thread pools. Some libraries, such as `PyArrow`, provide integrated multithreaded operations for reading and loading data from large CSV files. This ensures that I/O operations no longer limit the simulator's performance when managing multiple entities. Java leverages the `ExecutorService` framework, which facilitates the concurrent execution of tasks, allowing for optimal resource utilisation. C++ programmers can utilise the `std::thread` library, which provides fine-grained control over thread management, and can employ programming libraries such as OpenMP or Intel Threading Building Blocks (TBB), which provide standardised interfaces for parallel multithreaded execution. By decoupling I/O processes from computational tasks, this strategy significantly enhances the throughput and responsiveness of simulators, irrespective of the underlying programming paradigm.

When simulators are constrained by CPU-bound operations, *Multiprocessing Parallelism* emerges as an essential technique for enhancing computational efficiency. This approach exploits the capabilities of multi-core architectures, enabling the distribution of workloads across multiple CPU cores. In Python, the `multiprocessing` and `joblib` libraries facilitate the creation of independent processes that can operate concurrently, effectively bypassing the Global Interpreter Lock (GIL). In Java, the `ForkJoinPool` framework allows for the efficient execution of parallel tasks by recursively dividing computations into smaller subtasks. In C++, developers often resort to Inter-Process Communication (IPC) mechanisms like MPI for high-performance computing applications that demand true parallel execution. Unlike multithreading tools, multiprocessing approaches provide better scalability for compute-intensive simulations. These techniques enable simulators to distribute tasks across physical cores efficiently, thereby reducing execution time and supporting higher model complexity. Regarding simulation tools encapsulated in FMU, they can leverage the aforementioned programming strategies, as the FMU containing the model can be run in COESI using Python, Java, or C++ interfaces.

For Python-coded simulators that require more advanced scalability solutions, *Distributed Computing Frameworks* like Ray are considered. Ray [10] is an open-source Python framework providing a unified interface to implement parallelism across threads, processes, and distributed clusters. Although this research focuses on vertical scalability within a single computer system, Ray's scalability features are evaluated for their ability to further optimise Python code. Simulators benefit from task scheduling and resource allocation by adopting

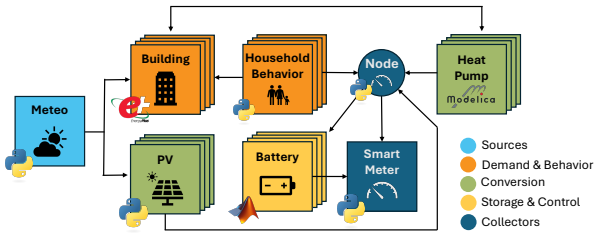

Fig. 2: Scenario diagram illustrating the interaction between different simulators and their data flow.

Ray, improving their performance under high-entity scenarios.

D. KPI Analysis

The final step involves analysing the impact of the scalability techniques that were implemented on the KPIs of the co-simulation. The scenario is executed multiple times with increasing numbers of entities, limited by the hardware capabilities of the testing machine. The *Execution Time* of the scaled scenario, both with and without the applied techniques, is compared to quantify the improvements, as it is the primary KPI under analysis. Resource usage patterns are monitored to ensure that scalability techniques do not introduce inefficiencies or unintended side effects. The insights gained from this analysis guide the *Best Technique Selection* for optimising co-simulation performance.

III. Scenario Design

The MES scenario analysed represents a building energy system that comprises multiple simulators, each representing different energy system components. These simulators have been integrated into the COESI platform [9], enabling interaction and data exchange through the co-simulation framework. This section overviews the simulators, their roles, and how they interact within the scenario. Refer to [9] for a detailed description of the simulators. A visual representation of the high-level interactions among simulators is shown in Fig. 2. This diagram illustrates the simulators, their respective model instances, their connections, and which simulation software or programming language was used to develop them.

The *Battery Simulator* model was designed and implemented in MATLAB Simulink and exported into the COESI platform through an FMU. The model depicts the behaviour of a standalone battery system, simulating its charge and discharge cycles based on input power levels controlled by an internal battery management system. It balances supply and demand with the building and energy network components.

The *Heat Pump Simulator*, developed with Modelica and integrated as an FMU, models a heating system that converts electricity into heat by extracting thermal energy from the external environment, even when temperatures are low, and transferring it indoors to provide warmth. Indeed, the system operates based on external temperature conditions and an internal control logic that dynamically adjusts the heat output to maintain the desired indoor setpoint temperature.

1) Building Simulator: The Building Simulator evaluates the heating and cooling demand of a building envelope. Inputs from Household Behaviour and Meteo simulators drive its calculations, and outputs are relayed to other models to reflect real-time energy consumption patterns. This simulator model was designed using EnergyPlus as a development tool and was also exported to COESI as an FMU.

The *Photovoltaic (PV) Simulator* is developed in Python and it models PV power generation based on meteorological data. It provides real-time generation data to the energy network model.

The *Node Simulator* is an intermediary between generation, consumption, and storage units, ensuring energy flows are adequately accounted for. Additionally, the *Smart Meter simulator* provides energy consumption and production monitoring. Both these simulators were developed in native Python code.

Time-series simulators supply real-world input data to the scenario. The *Household Behaviour Simulator* provides household power consumption and occupancy profiles. Moreover, it defines indoor temperature setpoints for the Building simulator to control the heating and cooling of the building. Instead, the *Meteo Simulator* provides weather conditions such as temperature and solar radiation. Data to these simulators is provided through CSV local files. Both simulators are also Python-based.

These simulators communicate with each other through defined data connections, allowing dynamic interactions between different components of the MES. The key connections include: *i)* The Building Simulator interacts with the heat pump simulator, exchanging indoor temperature values and heating power requirements; *ii)* The Heat Pump Simulator connects to the Node Simulator to report power consumption; *iii)* The PV Simulator provides solar power generation data to the Node simulator; *iv)* The Battery Simulator exchanges power flow information with the Node Simulator, ensuring proper charge and discharge cycles; *v)* The Household Behaviour Simulator provides household power demand values to the Building and Node Simulators. Finally, *vi)* The Meteo Simulator supplies weather data to the Building and PV Simulators, influencing heating demand and solar power generation.

IV. Experimental Results

The tests were conducted on a virtual machine hosted on Google Cloud to evaluate the performance of different vertical scalability techniques. This environment provided the necessary computational resources to handle the increasing demands of the simulation. The virtual machine used for the experiments was configured with an x86_64 architecture and ran on Ubuntu 22.04 as the operating system. It was equipped with 8 virtual CPUs (vCPUs) and 30 GB of RAM, ensuring sufficient memory for running multiple simulators and handling large datasets. The benchmarking process involved executing the co-simulation under controlled conditions, ensuring reproducibility and consistency across all tested configurations.

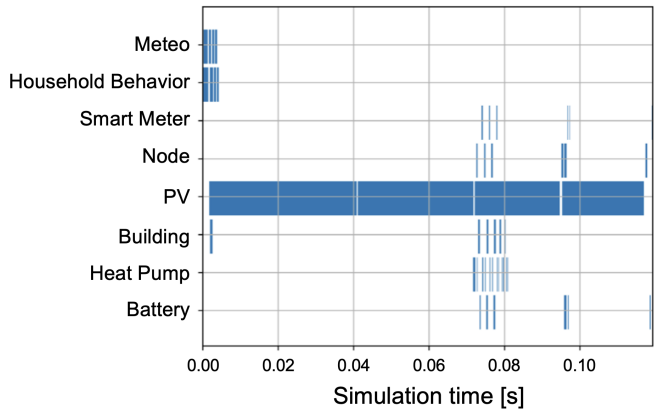

Fig. 3: Execution time profiling for the one entity per simulator configuration of the MES scenario. The chart highlights the active time frames for each simulator, showing periods when each simulator performs tasks.

A. Co-simulation Profiling

To understand the computational characteristics of each simulator, a baseline simulation was performed with one entity per simulator. A detailed visualisation of execution times is provided in Fig. 3, which shows the dominant factors in the performance of each simulator. The execution time of each simulator was analysed, leading to the following observations:
i) PV Simulator: Identified as CPU-bound, making it a suitable candidate for multiprocessing and Ray-based parallelisation.
ii) Meteo and Household Behaviour Simulators: Characterised as I/O-bound, benefiting from efficient data handling techniques rather than traditional parallelisation.
iii) Building, Heat Pump, and Battery Simulators: FMU-based simulators where caching mechanisms could help reduce redundant computations.

B. Vertical Scalability Application

Given the results discussed in Section IV-A, the next step was to test vertical scaling techniques on the PV simulator, identified as the main bottleneck of the simulation. The main focus of the parallelisation efforts was the `step` method of the simulator class, which iterates through all entities and steps them sequentially. Two parallelisation approaches were tested to enhance its performance and scalability: multiprocessing and Ray-based parallelism.

Ray was selected due to its flexibility in handling distributed execution, offering *Ray Tasks* and *Ray Actors* as two primary parallelisation models. *Ray Tasks* are stateless, lightweight functions executing in parallel across available workers. They are well suited for scenarios where each computation is independent, such as concurrently stepping multiple PV simulator instances. *Ray Actors*, unlike tasks, maintain an internal state across multiple function calls. This makes them more suitable for simulators requiring persistent memory between steps.

Although these two methodologies typically address distinct application patterns, the analysis presented here intentionally compares these models within a broader context beyond simple code execution. The focus is on exploring their fundamental effects when applied to parallelisation at the simulator level within a vertical scaling scenario. The rationale for comparing these foundational concepts lies in an incremental approach: mastering the core functionalities offered by Ray is essential before progressing to more complex, horizontally scalable scenarios, such as applying containerization and orchestration platforms like Docker and Kubernetes.

Both *Ray Tasks* and *Ray Actors* were integrated into the original MES scenario, with an increasing number of entities per simulator to evaluate the scalability of these techniques. Since the PV simulator operates as a CPU-based simulator, this setup allowed for testing both Ray Tasks and Ray Actors, comparing the two approaches. Additionally, the `num_cpus` parameter was utilised to specify the number of cores allocated to each task/actor. In an 8-core environment, the `num_cpus` parameter could be set to 1 for entity counts up to 8, assigning one core per task/actor. When there are more than 8 entities, Ray's scheduling algorithm queues the additional ones and executes them as computing cores become available. Alternatively, the `num_cpus` parameter could be set to fractional values (e.g., 0.5) to allow more entities to run concurrently by allocating fewer computational resources per task/actor.

Additional tests were conducted to evaluate the impact of fractional `num_cpus` configurations on simulation performance for *Ray Tasks* and *Ray Actors*.

TABLE I: Simulation times with different `num_cpus` configurations for *Ray Tasks*

# Entities / simulator	num_cpus	Simulation Time
16	1	00:00:17,781
16	0.5	00:00:19,351
32	1	00:00:30,923
32	0.5	00:00:33,086
32	0.25	00:00:34,773

The first set of tests involved Ray Tasks. As shown in Table I, fractional CPU assignments slightly increased simulation times. Comparing 16 entities, using `num_cpus=0.5` resulted in an 8.8% increase (17.781s to 19.351s). Similarly, for 32 entities, reducing CPU allocation from 1 to 0.5 increased simulation time by approximately 7% (30.923s to 33.086s), and further reducing to 0.25 added an additional increase of about 5% (33.086s to 34.773s). This indicates that fractional CPU allocation negatively impacts performance, likely due to additional scheduling overhead.

TABLE II: Simulation times with different `num_cpus` configurations for *Ray Actors*

# Entities / simulator	num_cpus	Simulation Time
16	0.5	00:00:20,501
32	0.25	00:00:37,772
64	0.125	00:01:04,744

The second set of tests focused on Ray Actors. As detailed in Table II, Ray Actors require the instantiation of a dedicated

class per virtual core, thus constraining CPU division: to run 16 entities on 8 cores, CPUs must be divided into two groups (0.5 per actor); for 32 entities, CPUs must be divided further (0.25 per actor), and so forth. The resulting simulation times show incremental performance penalties. For instance, simulation times with 64 entities and `num_cpus=0.125` reached over 1 minute (64.744s), illustrating diminishing returns.

These observations highlight that fractional CPU allocation consistently introduces a performance overhead, albeit relatively modest. The overhead results primarily from the increased complexity and scheduling latency of managing smaller CPU partitions. Consequently, given these findings, it was determined that the optimal configuration for subsequent results and analyses for Ray Tasks would use `num_cpus=1`, offering the best performance outcomes.

Figure 4 provides a comparative plot of the total simulation times for each configuration, including the non-optimised baseline, multiprocessing, Ray Tasks, and Ray Actors setups. This figure illustrates the impact of each parallelisation technique on execution time, enabling a detailed comparison of their effectiveness on the overall simulation.

As shown in Fig. 4, the optimised configuration consistently achieved lower total simulation times, with significant reductions as the entity count increased. For low entity counts (e.g. 2 and 4 entities), the differences between the baseline and optimised configurations are minimal, as illustrated by the closely aligned values in Fig. 4.

However, as the entity count increases, the benefits of the parallelisation techniques become more substantial. For 8 entities, the improvements are of about 36% for multiprocessing (from 20.55s to 13.12s), around 46% for both Ray Tasks and Ray Actors (down to 11.14s). This trend continues as we scale up when the cosimulation includes 32 entities with time reduction around 54% for multiprocessing (from 73.39s to 33.48s) and about 58% for Ray Tasks (down to 30.92s). The Ray Actors' configuration performs slightly less efficiently than Ray Tasks yielding an improvements of about 49% compared to the non-optimised baseline (i.e. down to 37.77s). The performance gains are particularly relevant at the largest scale of 64 entities. The multiprocessing setup reduced the total time by over half, from 145.77s to 66.51s, i.e. almost 55%. This substantial improvement highlights the scalability benefits of multiprocessing for CPU-bound tasks in the PV simulator, allowing the simulation to manage a larger workload more efficiently. Ray Tasks achieved even better performance, reducing total time to 57.78s, resulting in approximately a 60.3% improvement over the baseline scenario. Ray Actors showed notable but slightly less pronounced gains, with total execution time at 64.74s, equating to roughly a 55.6% improvement. Although the new distributed computing frameworks clearly improve simulation time, their impact on small-scale simulations remains limited—since even without optimization, the process takes just over two minutes. However, optimisation becomes crucial when scaling to hundreds or thousands of simulators. For instance, extrapolating the curves from Fig. 4 to a scenario with four thousand buildings

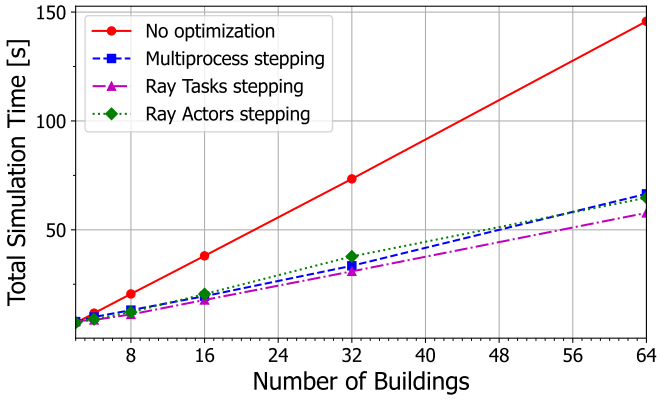

Fig. 4: Comparison of total execution times between the non-optimised configuration, the multiprocessing configuration, and both Ray Tasks and actors configurations for the PV simulator `step` method in the MES scenario.

shows that optimization using Ray Tasks yields a performance gain exceeding 64%, reducing execution time from about two and a half hours to just about 53 minutes.

The Ray Tasks configuration is the most efficient approach for the PV simulator, especially as the entity count increases. The flexibility offered by Ray Tasks, including fine-grained control over CPU allocation (e.g., fractional core usage with the `num_cpus` parameter), allows for effective parallelisation that surpasses both Ray Actors and multiprocessing in terms of performance. While beneficial for stateful scenarios, Ray Actors introduce a slight overhead that makes them less efficient than Ray Tasks.

These results align with expectations for CPU-bound tasks where parallelisation reduces the computational load on each processor core, effectively distributing the workload. The reduction in simulation time at higher scales suggests that the PV simulator's parallelised `step` method allows the simulation to utilise the available processing power more effectively, minimising bottlenecks and enhancing overall performance. These findings underscore the value of targeted optimisations based on initial profiling and highlight the importance of scaling techniques for CPU-intensive components in distributed co-simulation environments.

redIt is important to highlight that the implemented tests explicitly targeted overcoming Python's GIL. By employing `multiprocessing` and Ray frameworks, simulations are executed across multiple independent processes, effectively bypassing the GIL constraint. Each simulator instance operates as a separate process, and parallelisation techniques manage multiple simulator entities concurrently at various abstraction levels. The observed improvements in execution times thus confirm successful distribution of computational workloads beyond Python's inherent single-thread limitations.

C. I/O optimisation

I/O-bound simulators, responsible for managing large datasets stored in CSV files, provide meteorological and household behaviour data in the proposed MES scenario, as

TABLE III: Loading times for CSV files using the standard serial approach with Pandas and PyArrow engine with multi-threaded I/O in the MES scenario.

CSV file	size	column	row	Load Time	
				Pandas	PyArrow
meteo.csv	20.6 MBytes	9	315742	00:00:00,227	00:00:00,046
family.csv	1.8 MBytes	3	52561	00:00:00,043	00:00:00,021

discussed in Section III. Given the substantial data volumes involved, optimising data retrieval and loading times is crucial for improving overall simulation efficiency. To achieve this purpose, the adopted I/O optimisation strategy uses both Pandas and PyArrow libraries. Pandas, a powerful data-handling library in Python, provides robust tools for reading and processing CSV files. When combined with the PyArrow engine, Pandas can utilise multithreading to read data in parallel across multiple CPU cores, providing a potential solution to improve I/O efficiency without introducing significant overhead.

To evaluate the performance of parallelised I/O operations, this experiment specifically focuses on profiling the time required to load CSV files into the simulator. Two configurations were tested: the standard serial loading approach using Pandas alone and the multithreaded approach enabled by PyArrow. This allowed for the direct comparison of the performance gains from parallel I/O operations. Table III reports the time to load the CSV files under both configurations. These results demonstrate whether the PyArrow engine's multithreaded I/O capability provides a tangible advantage over the traditional serial approach, effectively reducing the data loading bottleneck for the Time Series simulators. The aim is to determine whether leveraging this optimised library can reduce file loading times and, by extension, improve the overall efficiency of the simulation pipeline. The PyArrow-optimised approach significantly reduces load times for both files, particularly for the larger dataset (i.e. meteo.csv), highlighting the benefits of multithreading for I/O-bound tasks. As reported in Table III, the small *family.csv* file (1.8 MB) showed a 51.2% reduction in load time, decreasing from 0.043s to 0.021s with PyArrow. While the improvement is notable, the difference is less dramatic than the larger file, likely because the smaller data size minimises the benefits of parallel I/O. For the larger *meteo.csv* file (20.6 MB), PyArrow's impact is even more pronounced, with the load time dropping from 0.227s to 0.046s, i.e. a 79.7% improvement. This indicates that PyArrow's multithreading is particularly effective when handling larger datasets, as the increased data volume maximises the advantages of parallel processing. These results suggest that PyArrow's multithreaded I/O can significantly reduce file load times in data-intensive scenarios, especially when dealing with larger files.

V. Conclusion

This paper presented a novel methodology for applying vertical scalability techniques to a large-scale co-simulation framework for MES. Our investigation focused on three primary research challenges: optimising simulation performance (**RC1**), improving resource utilisation (**RC2**), and exploring the applicability of optimisation techniques (**RC3**). By systematically analysing these aspects within the context of the COESI platform, we have uncovered significant insights that advance the state of the art in co-simulation methodologies. The implementation of the methodology fosters a standardised application of vertical scaling techniques, including multiprocessing, multithreading, and a distributed computing framework, that proves effective in overcoming the scalability bottlenecks typically encountered in large-scale co-simulations. Results applied to a simple MES scenario demonstrated that multiprocessing and Ray-based approaches significantly reduced execution time for CPU-bound simulators, with Ray showing additional flexibility in workload distribution. Furthermore, optimisations for I/O-bound simulators were explored, leveraging efficient data handling techniques to minimise retrieval overhead. The findings highlight the importance of selecting appropriate parallelisation techniques based on the characteristics of each simulator. While multiprocessing and Ray improved performance for CPU-bound workloads, I/O-bound simulators required different optimisation approaches. These results provide valuable insights for designing and deploying scalable co-simulation infrastructures.

Future work will explore horizontal scaling across distributed systems, implement some of these techniques on individual simulators to further enhance scalability, and refine optimisation techniques for energy-based simulators.

References

[1] P. Mancarella, "Mes (multi-energy systems): An overview of concepts and evaluation models," *Energy*, vol. 65, pp. 1–17, 2014.

[2] C. Steinbrink, M. Blank-Babazadeh, A. El-Ama, S. Holly, B. Lüers, M. Nebel-Wenner, R. P. Ramírez Acosta, T. Raub, J. S. Schwarz, S. Stark, A. Nieße, and S. Lehnhoff, "Cpes testing with mosaik: Co-simulation planning, execution and analysis," *Applied Sciences*, vol. 9, no. 5, 2019.

[3] T. D. Hardy, B. Palmintier, P. L. Top, D. Krishnamurthy, and J. C. Fuller, "Helics: A co-simulation framework for scalable multi-domain modeling and analysis," *IEEE Access*, vol. 12, pp. 24 325–24 347, 2024.

[4] Y. Li, J. Chen, Z. Hu, H. Zhang, J. Lu, and D. Kiritsis, "Co-simulation of complex engineered systems enabled by a cognitive twin architecture," *International Journal of Production Research*, 2021.

[5] C. Gomes, "Foundations for continuous time hierarchical co-simulation." 2016.

[6] M. Lajolo, C. Passerone, and L. Lavagno, "Scalable techniques for system-level cosimulation and coestimation," 2003.

[7] S. Vialle, J.-P. Tavella, C. Dad, R. Corniglion, M. Caujolle, and V. Reinbold, "Scaling fmi-cs based multi-simulation beyond thousand fmus on infiniband cluster," no. 132. Linköping University Electronic Press, 2017, pp. 673–682.

[8] L. Barbierato, D. Salvatore Schiera, M. Orlando, A. Lanzini, E. Pons, L. Bottaccioli, and E. Patti, "Facilitating smart grids integration through a hybrid multi-model co-simulation framework," *IEEE Access*, vol. 12, pp. 104 878–104 897, 2024.

[9] D. S. Schiera, L. Barbierato, A. Lanzini, R. Borchiellini, E. Pons, E. Bompard, E. Patti, E. Macii, and L. Bottaccioli, "A distributed multi-model platform to cosimulate multienergy systems in smart buildings," *IEEE Transactions on Industry Applications*, vol. 57, no. 5, pp. 4428–4440, 2021.

[10] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: a distributed framework for emerging ai applications," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 561–577.