# Efficient Parallelized Simulation of Cyber-Physical Systems

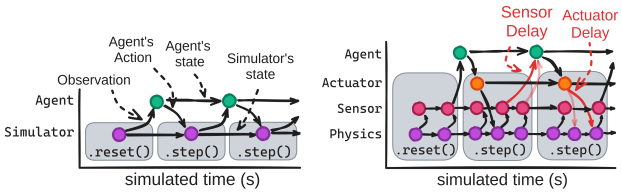
**Anonymous authors**
**Paper under double-blind review**

## Abstract

Advancements in accelerated physics simulations have greatly reduced training times for reinforcement learning policies, yet conventional step-by-step agent-simulator interaction undermines simulation accuracy. In the real-world, interactions are asynchronous, with sensing, acting and processing happening simultaneously; failing to capture this widens the sim2real gap and results in suboptimal real-world performance. In this paper, we address the challenges of simulating realistic asynchronicity and delays within parallelized simulations, crucial to bridging the sim2real gap in complex cyber-physical systems. Our approach efficiently parallelizes cyber-physical system simulations on accelerator hardware, including physics, sensors, actuators, processing components and their asynchronous interactions. We extend existing accelerated physics simulations with latency simulation capabilities by constructing a 'supergraph' that encodes all data dependencies across parallelized simulation steps, ensuring accurate simulation. By finding the smallest supergraph, we minimize redundant computation. We validate our approach on two real-world systems and perform an extensive ablation, demonstrating its superior performance compared to baseline methods.

## 1 Introduction

Physics simulations on accelerator hardware (NVIDIA, 2020; Hu et al., 2020; Freeman et al., 2021; Todorov et al., 2012) has significantly reduced training times for reinforcement learning policies that conform to traditional, sequentially-structured agent-simulator interactions (Rudin et al., 2022). Such interactions lead to clear-cut and predictable execution paths, allowing for efficient parallelization, as shown in Fig. 1a. However, this sequential approach fails to capture the concurrent and dynamic nature of cyber-physical systems—complex networks where computational elements, sensors, and actuators interact asynchronously with the physical world in real-time (Baheti & Gill, 2011).




(a) Sequential interaction (b) Asynchronous interaction

Figure 1: Comparative illustration of computation graphs with and without simulated delays. Vertices represent periodic computations, and edges represent data dependencies. (a) The absence of delay simulation creates consistent blocks of computation, enabling efficient parallelization across simulation steps, yet failing to capture the inherent asynchrony of the real-world. (b) While improving simulation fidelity, simulated delays between various components turn every simulation step into a diverse mix of computation, challenging parallelization efficiency.

In cyber-physical systems, an agent might rely on outdated information if sensor data is delayed, or slow policy evaluation unavoidably extends the effect of previous actions beyond their planned duration. Moreover, the focus has traditionally been on single agents trained end-to-end (Singh et al., 2019). In practice, however, AI systems deployed in real-world settings often rely on a pipeline of models. Accounting for the latency between these models will become crucial as tasks grow in complexity (Nishihara et al., 2017). Finally, physics simulators often bundle physics, sensor, and actuator simulations into a single unit running at a single rate. However, in reality, there are vital asynchronous effects within this block that need to be accounted for. Overlooking these asynchronous effects in simulation widens the sim2real gap and can lead to policies that do not perform well in the real-world.

To represent the asynchronous interactions between components, we advocate the division of the simulator into separate parts. This matches the typical design in robotics, where systems consist of interconnected nodes operating asynchronously at various rates (Quigley et al., 2009). This division enables the creation of computation graphs that accurately represent data flow in real-world situations, including latency effects. Consequently, each simulation step turns into a diverse mix of computation units from various components, as illustrated in Fig. 1b. These must be executed in a sequence that respects the data dependencies outlined by the graph's edges. Simulating with these diverse partitions improves accuracy but complicates parallelization (i.e., simulating multiple copies of the simulation in parallel), as distinct partitions may need to execute simultaneously across GPU threads, hindering GPU efficiency. Such misalignment can happen with independent episodic resets, often initiated based on variable reset criteria. One parallel simulation might reset because the agent reached its goal, while another continues because the agent is still far away. Diverging execution paths can significantly reduce kernel efficiency (Shibata, 2010). When GPU threads take different paths, they must be serialized, leading to more instructions and reduced performance.

The main contribution of this paper is an approach to parallelize cyber-physical system simulations that emulates asynchronicity and delays with minimized computational overhead on accelerator hardware. This allows existing accelerated physics simulations to be extended with efficient latency simulation capabilities. We achieve this by identifying a graph—ideally the smallest one possible—that encodes all the data dependencies outlined by every simulation step's edges. This universal graph, referred to as a *supergraph*, is determined prior to simulation. Sorting the supergraph topologically yields a static execution order for parallel processing of simulation steps without violating data dependencies. By targeting the smallest supergraph, we minimize redundant computation. Finding the smallest supergraph is generally a complex, NP-hard problem (Trimble, 2023); however, our greedy algorithm efficiently approximates this supergraph by leveraging the inherent periodicity in cyber-physical systems.

In sum, we make four key claims: Our approach (i) emulates asynchronicity leading to more accurate simulation, (ii) efficiently handles time-scale differences and asynchronicity resulting in higher parallelized simulation speeds than baseline approaches, (iii) scales to complex system topologies, and (iv) implements simplifications that reduce computational complexity with minimal impact on performance. These claims are supported by an experimental evaluation on two real-world robotic systems, followed by a scalability analysis and an ablation study. A motivational video for our approach is included as supplementary material.

## 2 Related Work

**Accelerated Physics Simulation** Accelerated physics simulators like Brax (Freeman et al., 2021), MJX (Todorov et al., 2012), and PhysX (NVIDIA, 2020) are designed for GPU execution. However, they lack features for simulating delays between their physics engine and other components such as sensors and actuators. Moreover, to mimic complete systems, these simulators must be extended with controllers and perception modules. Yet, these extensions typically interact with simulators sequentially, ignoring the concurrent and asynchronous nature of real-world systems. Our approach builds on this by dividing these simulators into separate components, facilitating the simulation of asynchronous interactions between them.

**Adressing Asynchronicity and Delays** The ORBIT framework (Mittal et al., 2023) and research by Bouteiller et al. (2021) have explored integrating delays into robotic simulations. While ORBIT introduces actuator delays to PhysX, it overlooks the asynchronicity between other system components. Bouteiller et al. (2021)'s work centers on compensating for system delays in the learning algorithm, not addressing the dynamic interactions among delayed components. In contrast, our method extends beyond actuator delays, encompassing asynchrony across all components.

**Minimum Common Supergraph** Our approach addresses the *minimal universal supergraph* (MUG) problem, which seeks the smallest supergraph, i.e., the *minimum common supergraph* (mcs), containing all graphs in a given set as a subgraph (Trimble, 2021). Unlike the brute-force algorithm presented in Trimble (2021), which is suitable only for small graph sets, our greedy algorithm is capable of handling graphs with more than 2000 vertices. In Bunke & Guidobaldi (2003), an iterative update strategy, based on Bunke et al. (2000), is utilized to approximate the mcs. Our method shares similarities but satisfies an additional

constraint: the resulting mcs must remain acyclic post-merge. Furthermore, we discover the set of graphs alongside the mcs, whereas Trimble (2021) and Bunke & Guidobaldi (2003) presume a given graph set. Both Trimble (2021), Bunke & Guidobaldi (2003) and our method solve the *maximum common subgraph* (MCS) problem as a subroutine to find the mcs (McCreesh et al., 2017; McGregor, 1982). However, our focus is on subgraph monomorphisms, which allow for additional edges in the subgraph, rather than induced subgraph isomorphisms, which require a one-to-one correspondence between every node and edge in the subgraph and target graph. To efficiently identify the MCS, we introduce an algorithm that leverages the acyclic nature of our mcs that accelerates the search for an approximate MCS. It is worth noting that our algorithm restricts the MCS search to connected subgraphs, potentially overlooking larger disconnected MCS candidates.

## 3 Preliminaries

Before diving into the details of our approach, we first lay down some basic definitions and notation that will aid in the formalization of our problem and the description of our approach. We consider graphs $\mathcal{G} = \big(V, E\big)$ consisting of a set of vertices $V(\mathcal{G})$ and a set of directed edges $E(\mathcal{G})$. Edge $(u, v) \in E(\mathcal{G})$ denotes an edge from vertex $u$ to vertex $v$. The notation $|\mathcal{G}|$ denotes the number of vertices in $\mathcal{G}$. Any subset of vertices $V' \subseteq V(\mathcal{G})$ induces a unique *subgraph* $\mathcal{G}' \subseteq \mathcal{G}$. The *difference* $\mathcal{G}_2 - \mathcal{G}_1$, where $\mathcal{G}_1 \subseteq \mathcal{G}_2$, yields a graph $\mathcal{G}$ with $V(\mathcal{G}) = V(\mathcal{G}_2) \setminus V(\mathcal{G}_1)$ and $E(\mathcal{G}) = E(\mathcal{G}_2) \setminus E(\mathcal{G}_1)$. The edges that connect $\mathcal{G}_1$ and $\mathcal{G}_2 - \mathcal{G}_1$ are defined as the *embedding* $\mathrm{emb}_{\mathcal{G}_2}(\mathcal{G}_1)$, which is a subset of $E(\mathcal{G}_2)$. The *union* of graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ with respect to a set of edges $E_{12}$ is denoted as $\mathcal{G} = \mathcal{G}_1 \cup_{E_{12}} \mathcal{G}_2$, where $V(\mathcal{G}) = V(\mathcal{G}_1) \cup V(\mathcal{G}_2)$ and $E(\mathcal{G}) = E(\mathcal{G}_1) \cup E(\mathcal{G}_2) \cup E_{12}$. The *addition* $\mathcal{G}_1 + \mathcal{G}_2$, where $\mathcal{G}_1, \mathcal{G}_2 \subseteq \mathcal{G}_3$, yields a subgraph $\mathcal{G}_{12} \subseteq \mathcal{G}_3$, by unifying $\mathcal{G}_1 \cup_{E_{12}} \mathcal{G}_2$ where $E_{12} = \mathrm{emb}_{\mathcal{G}_3}(\mathcal{G}_1) \cap \mathrm{emb}_{\mathcal{G}_3}(\mathcal{G}_2)$. An *edge contraction* on an edge $(u, v) \in E(\mathcal{G})$ yields a new graph $\mathcal{G}'$ such that $V(\mathcal{G}') = V(\mathcal{G}) \setminus \{u, v\} \cup \{w\}$ and

$$
\begin{aligned}
E(\mathcal{G}') = (&E(\mathcal{G}) \setminus \{(u, v), (v, u)\}) \\
&\cup \{(w, x) \mid (u, x) \in E(\mathcal{G}) \text{ or } (v, x) \in E(\mathcal{G})\} \\
&\cup \{(x, w) \mid (x, u) \in E(\mathcal{G}) \text{ or } (x, v) \in E(\mathcal{G})\}.
\end{aligned}
$$

The *ancestors* of a vertex $A_{\mathcal{G}}(u)$ are all vertices $V'(\mathcal{G}) \subseteq V(\mathcal{G})$ that can reach $u$ via a directed path in $\mathcal{G}$. The *roots* of a graph $\mathcal{G}$ are the set of vertices that have no incoming edges, formally $R(\mathcal{G}) = \{u \in V(\mathcal{G}) \mid \forall v \in V(\mathcal{G}), (v, u) \notin E(\mathcal{G})\}$. Similarly, the *leafs* of a graph $\mathcal{G}$ are the set of vertices that have no outgoing edges. A *Directed Acyclic Graph (DAG)* is a directed graph that contains no cycles. A *topological sort* $\tau$ of a directed acyclic graph $\mathcal{G}$ is a linear ordering of its vertices such that for every directed edge $(u, v) \in E(\mathcal{G})$, vertex $u$ comes before $v$ in the ordering. Multiple topological sorts may exist for a given graph $\mathcal{G}$, and the set of all possible topological sorts is denoted by $\mathcal{T}(\mathcal{G})$. A *labeling function* $L : V \to l$ is a function that assigns a label to each vertex. The set of all vertices with label $l$ is denoted by $V_l(\mathcal{G})$ and is arranged as a sorted list consistent with a topological sort of $\mathcal{G}$. We denote the set of topological sorts where the final vertex is of label $l$ in $\mathcal{G}$ as $\mathcal{T}_l^{-1}(\mathcal{G})$. Formally, this is defined as:

$$
\mathcal{T}_l^{-1}(\mathcal{G}) = \{\tau \in \mathcal{T}(\mathcal{G}) \mid I(\tau, u) = |\mathcal{G}|, u \in V_l(\mathcal{G})\},
$$

where $I(\tau, u)$ gives the position of vertex $u$ in the sorted set $\tau$. A *matching function* $f_{\mathrm{m}} : V \times V \to \{\mathrm{True}, \mathrm{False}\}$ is defined as follows:

$$
f_{\mathrm{m}}(u, v) = \begin{cases} \mathrm{True} & \text{if } L(u) = L(v), \\ \mathrm{False} & \text{otherwise.} \end{cases}
$$

A *mapping* between two graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ is a bijective function $M : V'(\mathcal{G}_1) \to V'(\mathcal{G}_2)$ where $V'$ represent a subset of the vertices. Its domain $\mathrm{dom}(M)$ is $V'(\mathcal{G}_1)$ and its range $\mathrm{rng}(M)$ is $V'(\mathcal{G}_2)$. Operations like union $\cup$, intersection $\cap$, and difference $\setminus$ can be applied to both $\mathrm{dom}(M)$ and $\mathrm{rng}(M)$. A mapping $M$ can extend to $M'$ by adding a new vertex pair $(u, v)$ with $M' = M \cup \{(u, v)\}$ where $u \in V(\mathcal{G}_1) \setminus \mathrm{dom}(M)$ and $v \in V(\mathcal{G}_2) \setminus \mathrm{rng}(M)$. A *subgraph monomorphism* $M : V(\mathcal{G}_1) \to V'(\mathcal{G}_2)$ is a specialized mapping that maps each vertex $u$ to $v$ such that $L(u) = L(v)$ and each edge $(u, v)$ corresponds to an edge $(M(u), M(v))$ in $\mathcal{G}_2$.

If such $M$ exists, $\mathcal{G}_2$ is a *supergraph* of $\mathcal{G}_1$ and can be reduced to $\mathcal{G}_1$ by removing vertices and edges in $\mathcal{G}_2$. The transformed set of edges $E_M(\mathcal{G}_1)$ under the mapping $M$ is defined as follows:

$$\begin{aligned} E_M(\mathcal{G}_1) = \{(u', v') \mid (u, v) \in E(\mathcal{G}_1), \\ u' = M(u) \text{ if } u \in \text{dom}(M), u' = u \text{ otherwise}, \\ v' = M(v) \text{ if } v \in \text{dom}(M), v' = v \text{ otherwise}\} \end{aligned}$$

This set includes edges $(u', v')$ where $u'$ and $v'$ are either mapped vertices of $u$ and $v$ under $M$ if they are in the domain of $M$, or are $u$ and $v$ themselves otherwise.

## 4 Our Approach

In our approach to efficiently parallize cyber-physical system simulations, characterized by delays and large time-scale differences, we propose a methodology illustrated in Fig. 2, where vertices represent periodic computations, and edges represent data dependencies. Given a set of target computation graphs that capture the system's asynchronicity, we aim to identify a partitioning of each graph and supergraph that enables parallel execution on accelerated hardware. These graphs might be partially recorded from real-world executions or synthetically created to reflect expected computation and communication delays. Each parallel simulation randomly selects any of the target graphs at the start of an episode, effectively emulating the inherent asynchronicity of the real-world system.

In aligning with standard simulator interfaces (Brockman et al., 2016) illustrated in Fig. 1, we *partition* these computation graphs into disjoint subgraphs, each corresponding to a simulation step. Crucially, we designate a *supervisor* node in each partition, a pivotal element that dictates the boundaries of these subgraphs. The supervisor node's operating rate sets the simulation time step, ensuring that each partition accurately reflects a discrete segment of the simulation process.

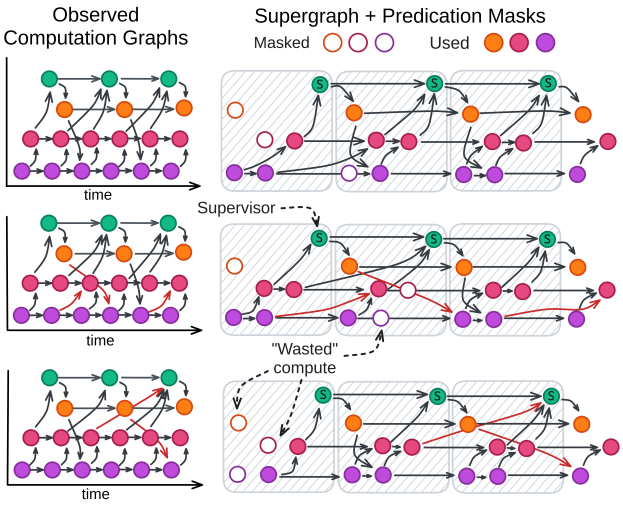



Figure 2: This figure illustrates our approach to efficiently simulating multi-rate asynchronous systems. Given variable delays, computation graphs can differ across experiments (left). We find a supergraph and predication masks, illustrated by the grey shaded blocks, for every computation graph that enables parallel execution across partitions (right). This mask, randomized during simulation, allows us to efficiently emulate asynchronicity and time-scale differences with minimal computational waste.

Our ultimate goal is to enable parallel simulations using accelerator hardware such as GPUs. To achieve this, we introduce an algorithm to identify a minimum common supergraph for the partitions of all computation graphs. A supergraph serves as a template that can be reduced to match any of the partitions by masking (i.e., removing) specific vertices and edges. Using a statically compiled supergraph along with a predication mask—where control flow dependencies are rewritten as data flow dependencies—allows for consistent execution paths for any partition (Taylor & Li, 2011). This setup enables parallel execution of any partition on accelerated hardware. It is important to note that predication, by its nature, executes all vertices in the supergraph, even those that are masked to fit a specific partition. Therefore, minimizing the size of the common supergraph is critical for reducing computational overhead.

### 4.1 Problem Definition

Consider a set of observed computation graphs denoted by $\{\mathcal{G}_0, \mathcal{G}_1, \dots\}$, where each $\mathcal{G}_i$ is a DAG. For a given supervisor label $s$, our goal is to partition each $\mathcal{G}_i$ into disjoint subgraphs $\mathcal{P}_{i,1}, \mathcal{P}_{i,2}, \dots$. Each subgraph corresponds to a discrete simulation step and contains exactly one leaf vertex labeled as $s$. The objective is

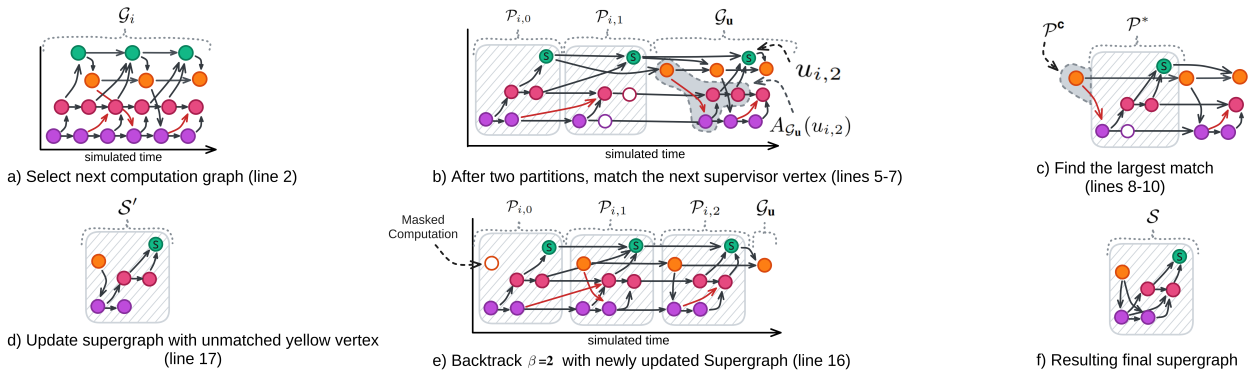



Figure 3: Illustration of the Minimum Common Supergraph Search process (Alg. 1) at a midway point. First, read from left to right, then top to bottom. A partial match is found for $u_{i,2}$, leading to an updated $\mathcal{S}'$ with missing ancestors $\mathcal{P}^\mathbf{c}$ and initiating a backtrack to re-evaluate previous partitions.

to determine these valid partitions along with the smallest DAG, $\mathcal{S}$, that serves as a common supergraph for all partitions. Similar to each partition, a single instance of the designated supervisor vertex in $\mathcal{S}$ must be a leaf vertex. Here, 'smallest' is defined by the number of vertices to minimize computational overhead. We aim to find a subgraph monomorphism $M_{i,j} : V(\mathcal{P}_{i,j}) \to V'(\mathcal{S})$ for each partition $\mathcal{P}_{i,j}$. This mapping allows us to reduce $\mathcal{S}$ into $\mathcal{P}_{i,j}$ using a predication mask. The predication mask is a binary mask applied to $\mathcal{S}$ to selectively remove vertices and edges not present in $\mathcal{P}_{i,j}$. Specifically, the mask is false for vertices and edges not in $\text{rng}(M_{i,j})$ and $E_{M_{i,j}}(\mathcal{P}_{i,j})$, respectively, and true otherwise.

### 4.2 Supergraph Search

**Algorithm 1:** Minimum Common Supergraph Search (mcs)

**Input:** Designated supervisor label $s$
**Input:** Number of steps to backtrack $\beta$
**Input:** A set of observed computation graphs $\{\mathcal{G}_0, \mathcal{G}_1, \dots\}$
**Output:** A set of partitions $\{\mathcal{P}_{0,1}, \mathcal{P}_{0,2}, \dots \mathcal{P}_{i,j} \dots\}$
**Output:** A supergraph $\mathcal{S}$ and mapping $M_{i,j}$ for all partitions $\mathcal{P}_{i,j}$

1 $\mathcal{S} \leftarrow$ Initialize with $V(\mathcal{S}) = \{u | L(u) = s\}$ and $E(\mathcal{S}) = \emptyset$
2 **for** $\mathcal{G}_i \in \{\mathcal{G}_0, \mathcal{G}_1, \dots\}$ **do**
3 $\quad\mathcal{G}_\text{u} \leftarrow$ Initialize unmatched graph as $\mathcal{G}_i$
$\quad$/* Until all supervisor vertices are matched */
4 $\quad$**while** $V_s(\mathcal{G}_u) \neq \emptyset$ **do**
5 $\quad\quad u_i \leftarrow$ Get next supervisor from sorted set $V_s(\mathcal{G}_\text{u})$
6 $\quad\quad u_{i,j} \leftarrow$ Index $u$ with $j = I(V_s(\mathcal{G}_i), u_i)$
7 $\quad\quad A_\text{u} \leftarrow u_{i,j}$ and its ancestors: $A_{\mathcal{G}_\text{u}}(u_{i,j}) \cup \{u_{i,j}\}$
8 $\quad\quad M^* \leftarrow$ Get largest map: Alg. 2 with $(s, \mathcal{S}, \mathcal{G}_\text{u}, A_\text{u})$
9 $\quad\quad \mathcal{P}^* \leftarrow$ Partition subgraph: $\text{dom}(M^*) \subseteq V(\mathcal{G})$
10 $\quad\quad \mathcal{P}^\mathbf{c} \leftarrow$ Missing subgraph: $A_{\mathcal{G}_\text{u}}(u_{i,j}) \setminus V(\mathcal{P}^*) \subseteq V(\mathcal{G})$
11 $\quad\quad$**if** $V(\mathcal{P}^\mathbf{c}) = \emptyset$ **then**
$\quad\quad\quad$/* All ancestors were matched */
12 $\quad\quad\quad M_{i,j} \leftarrow$ Store subgraph monomorphism $M^*$
13 $\quad\quad\quad \mathcal{P}_{i,j} \leftarrow$ Store partition $\mathcal{P}^*$
14 $\quad\quad\quad \mathcal{G}_\text{u} \leftarrow$ Remove matched partition: $\mathcal{G}_\text{u} - \mathcal{P}_{i,j}$
15 $\quad\quad$**else**
$\quad\quad\quad$/* Partial match */
16 $\quad\quad\quad \mathcal{G}_\text{u} \leftarrow$ Restore $\beta$ partitions in $\mathcal{G}_\text{u} \subseteq \mathcal{G}_i$ with $\mathcal{G}_\text{u} + \mathcal{P}_{i,j-\beta} + \mathcal{P}_{i,j-\beta+1} + \dots + \mathcal{P}_{i,j-1}$
17 $\quad\quad \mathcal{S} \leftarrow$ Update to $\mathcal{S}'$ with missing vertices and edges: $\big(V(\mathcal{S}) \cup V(\mathcal{P}^\mathbf{c}), E(\mathcal{S}) \cup E_{M^*}(\mathcal{P}^* + \mathcal{P}^\mathbf{c})\big)$

Our approach, as outlined in Alg. 1 and illustrated in Fig. 3, aims to simultaneously achieve three main objectives: identifying the supergraph $\mathcal{S}$, determining the partitionings $\mathcal{P}_{i,j}$, and discovering the associated mappings $M_{i,j}$. For each computation graph $\mathcal{G}_i$, the algorithm iterates until all supervisor vertices are matched, as specified in Line 4. In every iteration, the largest partition $\mathcal{P}^*$ and its associated mapping $M^*$ are sought (Line 5-8), following the method detailed in Alg. 2 and explained later on in Sec. 4.3.

Depending on whether all ancestors are matched, the algorithm finds either a complete or a partial match corresponding to the supervisor vertex $u_{i,j}$. In the case of a complete match, both $\mathcal{P}^*$ and $M^*$ are stored (Line 11-14). For partial matches, the algorithm backtracks $\beta$ iterations to reconsider previously matched partitions (Line 16). In either case, the supergraph $\mathcal{S}$ is updated using Eq. (1) to ensure it remains a supergraph of its previous version and incorporates all necessary ancestors $V(\mathcal{P}^\mathbf{c})$ for future matches, as follows:

$$\mathcal{S}' = \big(V(\mathcal{S}) \cup V(\mathcal{P}^\mathbf{c}), E(\mathcal{S}) \cup E_{M^*}(\mathcal{P}^* + \mathcal{P}^\mathbf{c})\big), \tag{1}$$

where $\mathcal{P}^* + \mathcal{P}^\mathbf{c} \subseteq \mathcal{G}_i$. More edges in the updated supergraph $\mathcal{S}'$ effectively constrain the number of possible mappings for subsequent partitions by reducing the number of topological sorts available in the supergraph.

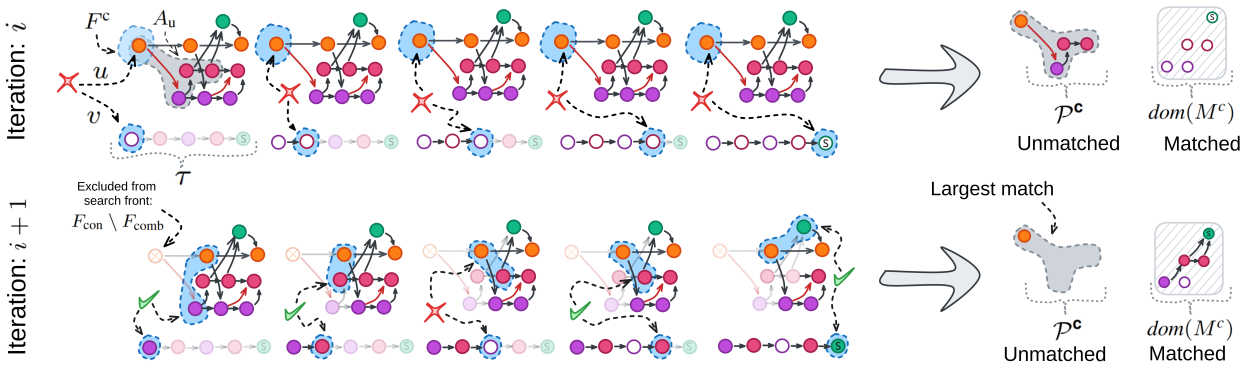



Figure 4: Midway illustration of Line 9-16 in the Largest Match Search (Alg. 2) linked to step (c) in Fig. 3. The ✓ and ✗ indicate whether a vertex in the candidate front $F^c$ is matched in Line 12. The first iteration yields an empty mapping due to the absence of a yellow vertex in the supergraph. In the second iteration, the vertex is excluded from the front, enabling a large partial match. The search is only displayed for a single topological sort.

Conversely, more vertices in $\mathcal{S}'$ increase its expressiveness by increasing the number of vertices that can be mapped to a vertex in subsequent partitions, but also increase the computational overhead of the simulation.

### 4.3 Largest Match Search

**Algorithm 2:** Largest Match Search

**Input:** Designated supervisor label $s$
**Input:** Supergraph $\mathcal{S}$
**Input:** Unmatched computation graph $\mathcal{G}_u$
**Input:** Vertices to be matched $A_u$
**Output:** Largest mapping $M^*$

1 $M^* \leftarrow$ Initialize an empty mapping
2 $\mathcal{G}_{excl} \leftarrow$ Initialize search graph as $\mathcal{G}_u$
3 **while** *True* **do**
4 $\quad F_{excl} \leftarrow$ Initialize search front as roots $R(\mathcal{G}_{excl})$
5 $\quad F_{con} \leftarrow$ Determine constrained front: $F_{excl} \cap A_u$
6 $\quad$ **forall** $F_{comb} \in$ *k-combinations*$(F_{con})$ **do**
7 $\quad\quad$ **forall** $\tau \in \mathcal{T}_s^{-1}(\mathcal{S})$ **do**
8 $\quad\quad\quad \mathcal{G}^c \leftarrow$ Remove $u \in F_{con} \setminus F_{comb}$ from $V(\mathcal{G}_{excl})$
9 $\quad\quad\quad F^c \leftarrow$ Initialize front: $F_{excl} \setminus (F_{con} \setminus F_{comb})$
10 $\quad\quad\quad M^c \leftarrow$ Initialize an empty candidate mapping
11 $\quad\quad\quad$ **forall** $v \in \tau$ **do**
12 $\quad\quad\quad\quad$ **if** $\exists u \in F^c : f_m(u, v) = True$ **then**
13 $\quad\quad\quad\quad\quad u \leftarrow \{u \in F^c : f_m(u, v) = \text{True}\}$
14 $\quad\quad\quad\quad\quad M^c \leftarrow$ Extend mapping: $M^c \cup \{(u, v)\}$
15 $\quad\quad\quad\quad\quad \mathcal{G}^c \leftarrow$ Remove matched $u$ from $V(\mathcal{G}^c)$
16 $\quad\quad\quad\quad\quad F^c \leftarrow$ Update front: $F^c \setminus \{u\} \cup R(\mathcal{G}^c)$
17 $\quad\quad\quad$ **if** $|dom(M^c) \cap A_u| > |dom(M^*) \cap A_u|$ **then**
18 $\quad\quad\quad\quad M^* \leftarrow M^c$ `/* Store largest mapping */`
19 $\quad\quad\quad s_{max} \leftarrow |A_u| - |A_u \setminus V(\mathcal{G}_{excl})| - (|F_{con}| - |F_{comb}|)$
20 $\quad\quad\quad$ **if** $(|dom(M^*) \cap A_u|) \geq s_{max}$ *or* $|dom(M^*)| = |\mathcal{S}|$ **then return** $M^*$
21 $\quad F_{excl} \leftarrow$ Exclude vertices from search graph: $V(\mathcal{G}_{excl}) \setminus F_{con}$

Our approach to identify the largest valid mapping $M^*$ for each supervisor vertex $u_{i,j}$ in $\mathcal{S}$ is outlined in Alg. 2 and illustrated in Fig. 4. Initially, it tries to match all ancestor vertices in candidate subgraphs $\mathcal{G}^c$, extracted from the unmatched graph $\mathcal{G}_{excl}$ (Line 2). The search is refined by iteratively excluding ancestors in a breadth-first manner from $\mathcal{G}^c$ (Line 8, and 21).

To this end, a refined search front $F^c$ is formed for each iteration (Line 4-6, Line 9). The algorithm explores all $k$-combinations of $F_{con}$ in descending order of $k$ (Line 6). For each combination, the largest candidate mapping $M^c$ is sought by traversing all valid topologies of $\mathcal{S}$, starting from the refined search front $F^c$ (Line 7-10). Everytime a match is found, the mapping is extended and the search front is updated (Line 11-16). Our approach assumes that vertices are stateful, i.e., vertices of similar labels are connected with one another, ensuring the uniqueness of the match (Line 12-13). If a larger mapping is found, $M^*$ is updated (Line 17-18). The algorithm halts the search if no larger match can be found (Line 19-20).

By design, Alg. 2 only considers mappings $M^c$ that ensure that collectively contracting all edges $(u, v) \in E(\mathcal{P}^c)$ in $\mathcal{G}_u$ will not introduce a cycle in the resulting graph $\mathcal{G}'_u$. Note here that $\mathcal{P}^c$ is the induced subgraph $\text{dom}(M^c) \subseteq V(\mathcal{G}_u)$. This constraint is critical to ensure that the updated supergraph $\mathcal{S}'$, as governed by equation Eq. (1), remains acyclic. This

sets us apart from previous work (Bunke et al., 2000; Bunke & Guidobaldi, 2003), which do not consider this constraint. A code example is provided as supplementary material that implements Alg .1 and 2.

### 4.4 Limitations and Approximations

The efficacy of our approach is contingent on a set of assumptions. Firstly, best performance is achieved when the computation graphs exhibit a recurring topological structure. Secondly, the model assumes substantial time-scale differences between what we term the *supervisor vertex* and other vertices. Finally, our approach assumes that vertices are stateful, i.e., vertices of similar labels are connected with one another. These assumptions are particularly well-suited for cyber-physical systems where components are stateful and run at fixed frequencies, and where the supervisor vertex often takes the form of a slower, learning agent or an outer-loop controller. Moreover, the algorithm assumes that the computation graphs' structure does not depend on the data processed by the vertices. Specifically, we assume delays in the system are not a function of the internal states, outputs, or incoming inputs.

Identifying the minimal common supergraph is an NP-hard problem (Trimble, 2023). To manage this complexity, we make several approximations to Alg. 2, and ablate their impact in Sec. 5. If all vertices are assumed to be stateful, then the constrained front $F_{\text{con}}$ can contain at most one vertex for each label, i.e. $|F_{\text{con}}| = |\text{rng}(L)|$. Then, the worst-case time complexity for considering all topological sorts of the supergraph $\mathcal{S}$ and all combinations of $F_{\text{con}}$ is $\mathcal{O}(2^{|\text{rng}(L)|} + |V|!)$ (Line 6-7 in Alg. 2). We alleviate this by considering only a single topological sort of $\mathcal{S}$ and a single combination per combination size $k$, reducing the worst-case time complexity to $\mathcal{O}(|\text{rng}(L) + 1| + |V|)$. Lastly, the sequence in which computation graphs are processed can affect the resultant supergraph. Similar to Bunke & Guidobaldi (2003), this has not proven to have a significant impact in our evaluations.

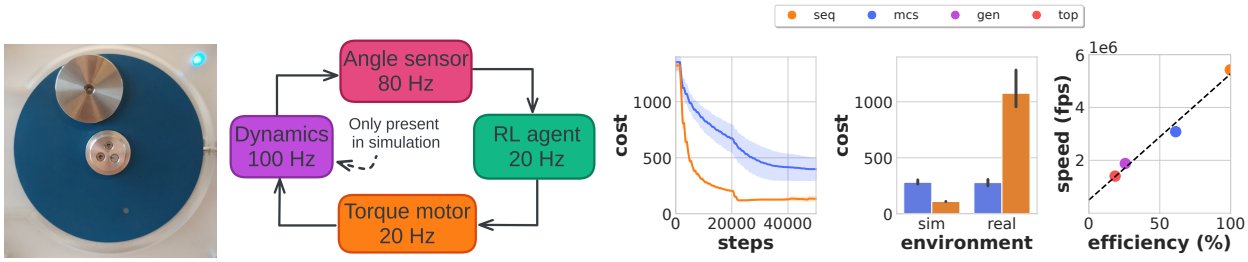
## 5 Experimental Evaluation

The main focus of this work is an efficient approach to simulate delays in parallelized simulation on accelerator hardware. We present our experiments to show the capabilities of our approach and to support our key claims, that our approach (i) emulates asynchronicity leading to more accurate simulation, (ii) efficiently handles time-scale differences and asynchronicity resulting in higher parallelized simulation speeds than baseline approaches, (iii) scales to complex system topologies, and (iv) implements simplifications that reduce computational complexity with minimal impact on performance. In the remainder of this section, we will use *mcs* to refer to our proposed method.

### 5.1 Baselines

We outline three baseline methods for our experimental evaluation. The *sequential* baseline (*seq*) assumes no delays in computation graph processing, illustrating a conventional approach as shown in Fig. 1a. This baseline serves as a reference for evaluating the impact of realistic delays in simulations.

We then introduce two baselines that incorporate delays by randomizing predication masks in parallelized simulations, but differ in supergraph construction. Given the absence of existing methods that can handle the DAG constraint and partitioning requirements for our supergraph (as discussed in Sec. 4.3), these baselines represent straightforward strategies for supergraph construction. Both baselines sequentially stack $K$ layers in the supergraph, with each layer containing a vertex for every non-supervisor label and concluding with a final layer of a single supervisor vertex. This structure ensures the supergraph is a DAG and with its size as $|\mathcal{S}| = K \times (|\text{rng}(L)| - 1) + 1$, thereby ensuring subgraph monomorphisms across partitions with an adequate number of layers. The *topological* baseline (*top*) sets $K$ equal to the number of vertices in the largest partition. While this method guarantees a subgraph monomorphism with each partition, it can lead to disproportionately large supergraphs with sparse layer utilization. The *generational* baseline (*gen*), on the other hand, sets $K$ as the maximum path distance across partitions. This approach is more space-efficient, but also tends to over-include vertices, as it does not account for time-scale differences between vertices. Consequently, each layer incorporates every vertex label, even those infrequently used.




(a) Pendulum (b) Control diagram (c) Training curve (d) Sim2real (e) Speed

Figure 5: Sim2real evaluation of an RL policy trained to swing up a pendulum with (*mcs*) and without delays (*seq*). Panels (a) and (b) depict the experimental setup and control diagram, respectively. Panels (c) and (d) show that *mcs* outperforms *seq* in the real-world evaluation, despite a slower convergence and lower simulated performance. Panel (e) shows that *mcs* achieves a higher mean simulation speed with a compacter supergraph (*mcs*: $|\mathcal{S}| = 13$) than baseline approaches that also consider delays (*gen*: $|\mathcal{S}| = 31$ and *top*: $|\mathcal{S}| = 43$).

To evaluate these methods, we introduce the *supergraph efficiency* metric ($\eta$):

$$\eta = 100 \times \frac{1}{N} \sum_{i,j} \frac{|\mathcal{P}_{i,j}|}{|\mathcal{S}|}$$

Here, $N$ denotes the total number of partitions, with $\eta$ indicating the proportion of each partition's size $\mathcal{P}_{i,j}$ relative to the total supergraph size. This metric effectively quantifies the proportion of vertices actively utilized (unmasked) in emulating the computation graphs across episodes. Note that a 100% efficiency may not be achievable in practice, as it would imply that all partitions are equal.

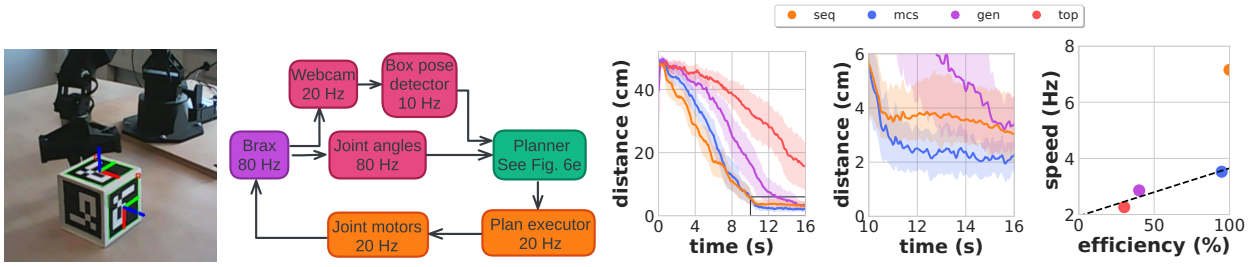
### 5.2 Performance

In these set of experiments we aim to validate that randomizing predication masks during training enhances the fidelity of robotic simulations and our approach to identifying the supergraph leads to more efficient parallelized simulations. We validate the performance on two real-world robotic systems: a pendulum swing-up task and a vision-based manipulation task. We use two different control strategies, reinforcement learning (RL) and model predictive control (MPC), to demonstrate the utility of our approach in different real-world settings.

#### 5.2.1 Pendulum swing-up task

The pendulum swing-up task is a well-known RL benchmark with nonlinear, unstable, and underactuated dynamics sensitive to delays (Derner et al., 2020). The experimental setup and control diagram are depicted in Fig. 5a and Fig. 5b, respectively. A failure to emulate the asynchronous real-world interactions between components makes a simulation-trained policy ineffective when transferred to a real-world setting. Policies were trained using soft actor-critic (SAC) (Haarnoja et al., 2018) in two simulators: one emulating delays (our approach: *mcs*) and another without delays (sequential approach: *seq*). Note that the *gen* and *top* baselines are not included in the sim2real evaluation. This exclusion is due to their replication of the same effective computation graphs as *mcs*, leading to identical policy outcomes. Hence, we only consider these baselines later on in the simulation speed evaluation within this section. We record 10 computation graphs from the real-world system to identify a supergraph, partitioning and corresponding predication masks that were randomized during training. Each experiment was replicated five times with different random seeds and the results are presented in Fig. 5. Though the sequential approach exhibits quicker convergence and superior simulated performance, it underperforms in real-world tests compared to our delayed simulation. A smaller performance gap between simulation and reality suggests that our approach leads to more accurate simulation, yielding more effective real-world policies.

To establish the link between efficiency and simulation speed, we carried out a parallelized performance evaluation of the swing up-task on an RTX 3070 GPU. We compiled the supergraph with JAX (Frostig et al.,




(a) Box pushing (b) Control diagram (c) Peformance (d) Final accuracy (e) Planning rate

Figure 6: A comparison of four MPC strategies for a task where a manipulator moves a box to a target: three consider delays (*mcs*, *gen*, *top*) and one does not (*seq*). Panels (a) and (b) depict the experimental setup and control diagram, respectively. Panels (c) and (d) depict mean convergence rate and final accuracy over 10 episodes with 95% confidence intervals, respectively, while (e) correlates these with the achieved replanning rate. The *seq* strategy, although faster initially, leads to less accurate movements due to ignoring delays. The *mcs* method, while replanning less frequently, achieves approximately 40% higher accuracy. Moreover, *mcs* exhibits the highest replanning rate with a smaller supergraph (*mcs*: $|\mathcal{G}| = 54$) compared to *gen* ($|\mathcal{G}| = 139$) and *top* ($|\mathcal{G}| = 223$) that also consider delays.

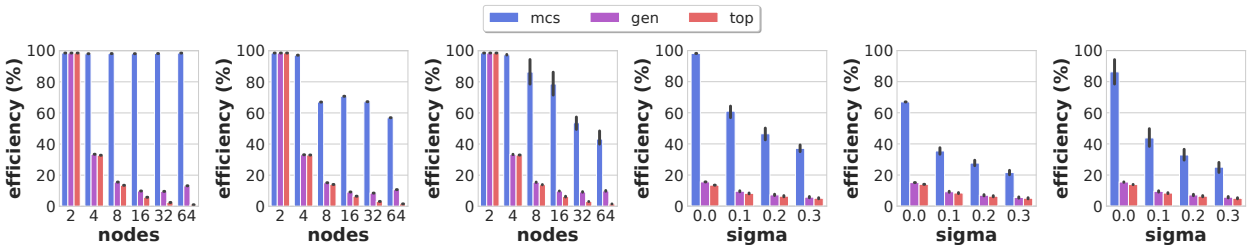
2018) and randomized the predication masks across 1000 parallelized episodes. We used the supergraphs produced by our approach with backtracking $\beta = 5$ and both baseline methods, and recorded the simulation frames per second (fps). As indicated in Fig. 5, our method notably outperforms other baselines that include delays, achieving an approximate simulation speed of 3 million fps. This improvement is largely attributed to a more compact supergraph. We observed a clear linear relationship between $\eta$ and simulation fps, which is consistent with the inverse proportionality between simulation fps and supergraph size.

#### 5.2.2 Manipulation task

In the manipulation task, a Viper 300x robotic manipulator moves a box to a target based on streaming webcam images. Our experimental setup and control diagram are shown in Fig. 6a and Fig. 6b. Emphasizing the importance of delay simulation, we use a consumer-grade Logitech C170 webcam, chosen for its low resolution, modest frame rate, and high latency, to track the box's position and orientation.

We adopt the MPC approach from Yang et al. (2020), planning actions based on the most recent robot observations using the Cross Entropy Method (CEM) (Rubinstein & Kroese, 2004). CEM, known for its efficient, derivative-free optimization, is particularly advantageous due to its parallelizability. Considering the contact-rich nature of box pushing, we opt for Brax (Freeman et al., 2021) as our dynamics model within the MPC framework, instead of learning complex contact dynamics. Brax, a differentiable physics simulator, is optimized for GPU acceleration and effectively handles contact-rich tasks. In a similar approach, Pezzato et al. (2023) recently used PhysX (NVIDIA, 2020) to solve a box pushing task. Our implementation employs CEM for three iterations, involving 75 samples per iteration and a planning horizon of two control steps, each lasting 0.15 seconds. We implement our approach using JAX (Frostig et al., 2018) and execute it on an RTX 3070 GPU.

We evaluate four MPC strategies: three accounting for delays (*mcs*, *gen*, *top*) and one ignoring them (*seq*). Delay-inclusive strategies, following Yang et al. (2020), use past plans to predict future box positions and orientations at action time. This prediction is based on the recorded computation graphs of the system that are used to identify a supergraph, partitioning, and corresponding predication masks. Due to its computational load, these strategies have a lower replanning rate compared to the delay-agnostic *seq*. The slower the replanning rate, the further into the future the planner must predict, increasing the likelihood of inaccurate predictions. As Fig. 6d shows, *mcs* achieves 40% higher accuracy than *seq*, despite less frequent replanning. Moreover, *mcs* method also results in smoother operations than *seq*. The larger supergraphs in *gen* and *top* result in excessively slow replanning, significantly reducing convergence rates, and final accuracy. This illustrates the trade-off between accuracy and efficiency, where the improved accuracy must justify the additional computational load.




(a) Unidirectional (b) Bidirectional (c) Unirandom (d) Unidirectional (e) Bidirectional (f) Unirandom

Figure 7: Efficiency comparison of *mcs* (our approach), *top*, and *gen*. In panels (a-c) the number of nodes is varied with no asynchronicity ($\sigma = 0$), while in panels (d-f) the asynchronicity levels are varied with a fixed size of $N = 8$ nodes.

### 5.3 Scalability

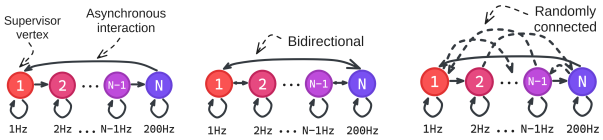
The next set of experiments support the claim that our approach scales to complex system topologies. We focus on artificially generated computation graphs, as they allow us to systematically vary the number of nodes, the level of asynchronicity, and the topology of the graph. We consider three different topologies: *unidirectional*, *bidirectional*, and *unirandom*, depicted in Fig. 8.

The nominal sampling time of each node is set according to the node's index $i$ as $\Delta t_i = \frac{1}{i}$ s, except for the last node's sampling time which is set to $\Delta t_N = \frac{1}{200}$ s. We argue that these topologies resemble cascaded control schemes that are common in robotic systems, with slower learning-based nodes and faster simulator nodes with intermediate controllers, estimators, sensors and actuators. The effective sampling time of each node is computed as $\Delta t_{i,k} = \Delta t_{i,k} + \max(0, x_k \Delta t_i)$, where $x_{i,k}$ is computation delay of node $i$ at sequence $k$ scaled with the node's nominal sampling time. An Ornstein-Uhlenbeck (OU) process (Bibbona et al., 2008) is used to model every node's delay to reflect the temporal correlation of delays, defined as follows:

$$x_k = \theta x_{k-1} + \sigma \nu,$$

where $\theta$ is a correlation coefficient, $\sigma$ is the standard deviation, and $\nu$ is a Gaussian random variable with zero mean and unit variance. The standard deviation of an OU process is related to the standard deviation of a Gaussian distribution with $\sigma_g = \sqrt{\frac{\sigma^2}{2\theta}}$.

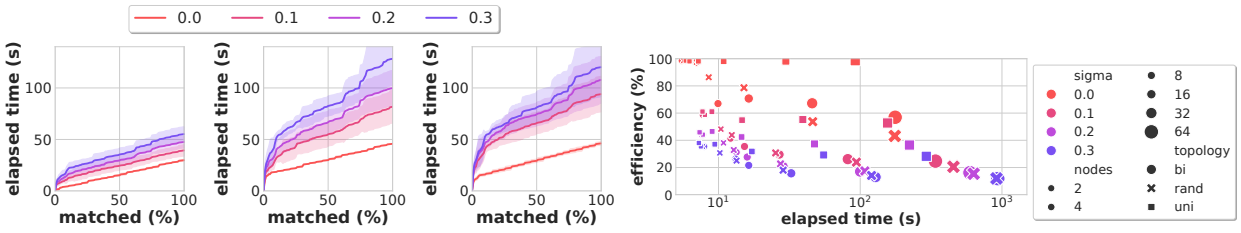
We consider the topologies in Fig. 8 for a different number of nodes $N \in \{2, 4, 8, 16, 32, 64\}$, and a varying levels of asynchronicity $\sigma \in \{0, 0.1, 0.2, 0.3\}$. We replicate each experiment 5 times using different random seeds. For each configuration, we generate 20 computation graphs, each running for a duration of 100 seconds. We employ Alg. 1 to identify a supergraph on a single core of an Intel Core i9-10980HK and compare its performance with two baseline approaches.




(a) Unidirectional (b) Bidirectional (c) Unirandom

Figure 8: Three topologies resembling cyber-physical systems to evaluate the scalability of our approach. (a) *Unidirectional*: each node has a single outgoing connection. (b) *Bidirectional*: each node has two outgoing connections. (c) *Unirandom*: akin to *Unidirectional*, but with an extra random outgoing connection per node.

Figures 9a, 9b, and 9c illustrate the performance of our algorithm for different numbers of nodes when there is no asynchronicity ($\sigma = 0$). Our approach achieves a 100% efficiency for the unidirectional topology, whereas the efficiency of baseline approaches declines rapidly as the number of nodes increases. The superior efficiency of our approach in the unidirectional topology is attributable to its fewer connections (i.e., constraints). Figures 9d, 9e, and 9f demonstrate the performance of our algorithm for different levels of asynchronicity when the network comprises $N = 8$ nodes. As asynchronicity increases, partitions become more dissimilar, and the efficiency of our approach does decline, yet it remains multiples higher than that of the baseline approaches.




(a) Unidirectional (b) Bidirectional (c) Unirandom (d) Complexity analysis

Figure 9: Performance analysis of computational complexity and efficiency. Panels (a-c) show the elapsed time for completion with $N = 32$ nodes across various asynchronicity levels and topologies. Initial episodes are time-intensive due to numerous preliminary partial matches, followed by a consistent linear time scaling in processing time. Panel (d) shows the computational complexity versus efficiency for different topologies, asynchronicity levels, and node counts, highlighting their impact on performance.

Fig. 9 presents an analysis of our method's computational complexity in constructing the supergraph, considering both the computation graph's characteristics and topology, and the scaling of supergraph search complexity over all recorded computation graphs. Figures 8a, 8b, and 8c detail our algorithm's complexity under varying asynchronicity levels through time as it processes all recorded computation graphs. The initial episodes incur higher computational costs due to the increased computational overhead of handling numerous partial matches (Line 6-7 in Alg. 2), while subsequent episodes demonstrate linear scaling in time.

In Fig. 8d, we observe that efficiency is inversely related to both the asynchronicity level and the number of connections per node within a topology. Specifically, the unidirectional topology outperforms the bidirectional and unirandom topologies due to its fewer edges, which equates to fewer constraints. A decrease in efficiency correlates with an increase in computation time, primarily because fewer complete matches are found. While the most substantial contributor to computation time is the number of nodes in the topology, it does not affect efficiency as similar efficiency is achieved with different numbers of nodes. Nevertheless, the one-time upfront cost of identifying the supergraph is minor when compared to the overall simulation time, substantiating our claim that our approach scales effectively to complex system topologies.

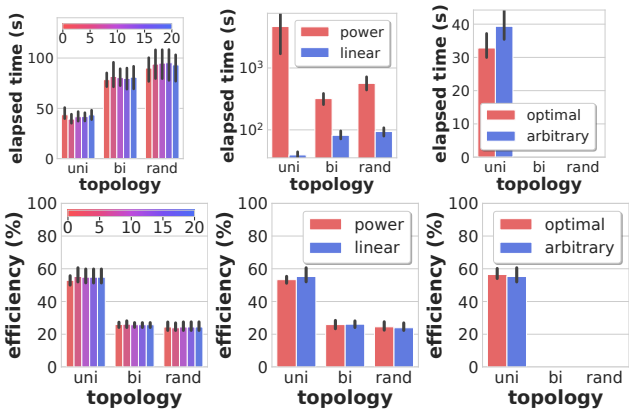
### 5.4 Ablation Study

Finally, our goal is to substantiate that our approach employs simplifications discussed in Sec. 4.4 that reduce computational complexity without significantly affecting performance. For this ablation study, we focus on different topologies with $N = 32$ and $\sigma = 0.1$, considering several simplifications.

Fig. 10 illustrates that the benefits of *backtracking* are limited. However, it neither increases the computational complexity of our approach nor adversely affects efficiency.

We also analyzed the effect of considering only a single combination for each size $k$, as opposed to exploring all combinations. Fig. 10 demonstrates that this simplification has negligible impact on efficiency but considerably reduces the computational complexity (note the log-scale).

Lastly, we explored the implications of using a single topological sort. Rather than exhaustively considering all topological sorts—an approach that would




(a) Backtrack (b) Combination (c) Sort

Figure 10: Ablation study on topologies with $N = 32$ and $\sigma = 0.1$, examining computational complexity and efficiency. Sub-figures show: (a) Effects of varying $\beta$; (b) Efficiency-impact of considering one (*linear*) vs. all combinations per size $k$ (*power*); (c) Comparison of *arbitrary* and *optimal* topological sorts.

be computationally prohibitive—we compared the
effects of using an arbitrary sort versus an optimal one. The optimal sort of the supergraph is defined as one that accommodates the maximum number of potential edges (i.e., constraints). Due to the inherent unidirectionality of the unidirectional topology, the optimal sort arranges vertices of lower indices before those of higher indices. Since we lack optimal sorting criteria for bidirectional and unirandom topologies, we limited this part of the study to the unidirectional topology. Fig. 10 shows that this simplification has negligible impact on efficiency.

In summary, our evaluation suggests that our method successfully emulates asynchronicity, offering more accurate and faster parallelized simulations compared to baseline approaches. At the same time, our method scales well to complex system topologies and utilizes algorithmic simplifications to reduce computational complexity without substantially affecting performance. Thus, we have substantiated all our key claims through this experimental evaluation.

## 6 Conclusion

In this paper, we introduced a method for efficiently simulating inherently asynchronous systems on accelerator hardware. Our approach leverages recorded computation graphs from real-world operations to accurately model asynchronicity and time-scale differences. The experiments suggest that our approach provides a scalable, efficient, and accurate means for simulating cyber-physical systems. We evaluated our method in two real-world scenarios against baselines and confirmed its efficacy in emulating asynchronicity and handling time-scale differences efficiently. Our work opens avenues for developing fast and accurate cyber-physical system simulations, particularly well-suited for parallel computing on accelerator hardware.

## References

Radhakisan Baheti and Helen Gill. Cyber-physical systems. *The Impact of Control Technology*, 12(1): 161–166, 2011.

Enrico Bibbona, Gianna Panfilo, and Patrizia Tavella. The Ornstein–Uhlenbeck process as a model of a low pass filtered white noise. *Metrologia*, 45(6):S117, 2008.

Yann Bouteiller, Simon Ramstedt, Giovanni Beltrame, Christopher Pal, and Jonathan Binas. Reinforcement learning with random delays. In *Proc. of the Int. Conf. on Learning Representations (ICLR)*, 2021.

G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. OpenAI Gym. *arXiv preprint*, 2016.

Horst Bunke and C Guidobaldi. Graph clustering using the weighted minimum common supergraph. *Graph Based Representations in Pattern Recognition*, pp. 235–246, 2003.

Horst Bunke, Xiaoyi Jiang, and Abraham Kandel. On the minimum common supergraph of two graphs. *Computing*, 65:13–25, 2000.

Erik Derner, Ji***r***í Kubalík, Nicola Ancona, and Robert Babu***s***ka. Constructing parsimonious analytic models for dynamic systems via symbolic regression. *Applied Soft Computing*, 94:106432, 2020.

C Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax-A Differentiable Physics Engine for Large Scale Rigid Body Simulation. In *Proc. of the Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 4(9), 2018.

T. Haarnoja et al. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *Proc. of the Int. Conf. on Machine Learning (ICML)*, pp. 1861–1870. PMLR, 2018.

Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. DiffTaichi: Differentiable Programming for Physical Simulation. In *Proc. of the Int. Conf. on Learning Representations (ICLR)*, 2020.

Ciaran McCreesh, Patrick Prosser, and James Trimble. A partitioning algorithm for maximum common subgraph problems. *Proc. of the Intl. Conf. on Artificial Intelligence (IJCAI)*, pp. 712–719, 2017.

James J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software: Practice and Experience*, 12, 1982.

M. Mittal, C. Yu, Q. Yu, J. Liu, N. Rudin, D. Hoeller, J. L. Yuan, R. Singh, Y. Guo, H. Mazhar, A. Mandlekar, B. Babich, G. State, M. Hutter, and A. Garg. Orbit: A Unified Simulation Framework for Interactive Robot Learning Environments. *IEEE Robotics and Automation Letters (RA-L)*, pp. 1–8, 2023. doi: 10.1109/LRA.2023.3270034.

Robert Nishihara, Philipp Moritz, Stephanie Wang, Alexey Tumanov, William Paul, Johann Schleier-Smith, Richard Liaw, Mehrdad Niknami, Michael I Jordan, and Ion Stoica. Real-time machine learning: The missing pieces. In *Proc. of the Workshop on Hot Topics n Operating Systems (HTOS)*, pp. 106–110, 2017.

NVIDIA. NVIDIA PhysX, 2020. URL `\url{https://developer.nvidia.com/physx-sdk}`.

Corrado Pezzato, Chadi Salmi, Max Spahn, Elia Trevisan, Javier Alonso-Mora, and Carlos Hernandez Corbato. Sampling-based model predictive control leveraging parallelizable physics simulations. *arXiv preprint*, 2023.

M Quigley et al. ROS: an open-source Robot Operating System. *Proc. of the IEEE Intl. Conf. on Robotics & Automation (ICRA)*, 3:5, 2009.

Reuven Y Rubinstein and Dirk P Kroese. The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation, and machine learning. *Springer*, 133, 2004.

Nikita Rudin, David Hoeller, Philipp Reist, and Marco Hutter. Learning to Walk in Minutes Using Massively Parallel Deep Reinforcement Learning. *Proc. of the Conf. Robot Learning (CoRL)*, 164:91–100, 2022.

Naoki Shibata. Efficient Evaluation Methods of Elementary Functions Suitable for SIMD Computation. *Computer science-Research and development*, 25:25–32, 2010. doi: 10.1007/s00450-010-0108-2.

Avi Singh, Larry Yang, Kristian Hartikainen, Chelsea Finn, and Sergey Levine. End-to-end robotic reinforcement learning without reward engineering. *arXiv preprint*, 2019.

Ryan Taylor and Xiaoming Li. Software-Based Branch Predication for AMD GPUs. *SIGARCH Comput. Archit. News*, 38(4):66–72, 2011. ISSN 0163-5964. doi: 10.1145/1926367.1926379.

Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pp. 5026–5033, 2012.

James Trimble. Induced universal graphs for families of small graphs. *arXiv preprint*, 2021.

James Trimble. *Partitioning algorithms for induced subgraph problems.* PhD thesis, University of Glasgow, 2023.

Yuxiang Yang, Ken Caluwaerts, Atil Iscen, Tingnan Zhang, Jie Tan, and Vikas Sindhwani. Data efficient reinforcement learning for legged robots. *Proc. of the Conf. Robot Learning (CoRL)*, 2020.