A Theoretical Approach to Determine the Optimal Size of a Thread Pool for Real-Time Systems

Daniel Casini*†
*TeCIP Institute, Scuola Superiore Sant'Anna, Pisa, Italy
†Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Pisa, Italy

***Abstract*—Parallel workloads most commonly execute onto pools of thread, allowing to dispatch and run individual nodes (e.g., implemented as C++ functions) at the user-space level. This is relevant in industrial cyber-physical systems, cloud, and edge computing, especially in systems leveraging deep neural networks (e.g., TensorFlow), where the computations are inherently parallel. When using thread pools, it is common to implement fork-join parallelism using blocking synchronization mechanisms provided by the operating system (such as condition variables), with the side effect of temporarily reducing the number of worker threads. Consequently, the served tasks may suffer from additional delays, thus potentially harming timing guarantees if such effects are not properly considered. Prior works studied such phenomena, providing methods to guarantee the timing behavior. However, the challenges introduced by thread pools with blocking synchronization cause current analyses to incur a notable pessimism. This paper tackles the problem from a different angle, proposing solutions to determine the optimal size of a thread pool in such a way as to avoid the undesired effects that arise from blocking synchronization.**

I. Introduction

Thread pools are increasingly used in many different applications, including web services, cloud and edge computing, robotics software, and especially in those systems leveraging artificial intelligence and deep neural networks, where computation is inherently parallel. In such settings, it is often far more convenient to schedule a portion of computation in user space as a function (e.g., a C++ function) scheduled by a pool of "worker" threads. For example, in the context of robotics, the multi-threaded executor of ROS 2, a popular middleware framework to manage robotics software, uses a pool of threads to dispatch the workload [1]. In these contexts, each application is commonly represented with a directed acyclic graph (DAG), where nodes denote sequential computation and edges precedence relations.

The advantages of thread-pool scheduling are even more evident in the context of emerging applications based on deep neural networks. To mention a concrete example, TensorFlow[1], a popular framework for deep neural networks (DNNs), makes use of thread pools to schedule the workload due to deep neural networks [2]. Under this configuration, the InceptionV3 [3] DNN running on a multicore platform originates more than $34,000$ sequential nodes [4]. In essence, these nodes are implemented by user-space C++ functions because, with such a massive number of nodes, it would be unviable to create a thread for each of them.

On the other hand, TensorFlow, as well as robotics software, web services, and cloud/edge computing, often require that the executed workload is guaranteed to be completed within an application deadline. Therefore, it is essential to correctly predict the timing behavior of applications executed on behalf of thread pools.

Two principal characteristics distinguish the scheduling of parallel tasks using thread pools from the assumptions used in usual schedulability tests that neglect them: **(i)** individual nodes correspond to *functions* (e.g., C++ functions handled in user-space in the case of TensorFlow) handled by user-space threads and therefore the underlying operating system (OS) is not aware of them, and **(ii)** fork-join parallelism is often implemented through blocking synchronization mechanisms (e.g., condition variables). These differences have significant consequences on the resulting scheduling behavior. First, preemption and migration of nodes (i.e., user-space functions) among threads are commonly not supported. Second, when they use the blocking synchronization primitives provided by the OS, e.g., as extensively discussed in the following, the corresponding blocking system call does not suspend only the individual node calling it, but *the whole thread serving the node*. Clearly, this temporary limits the number of worker threads ready to execute functions, possibly creating additional delays to the other nodes.

Previous work [4] proposed methods to account for these undesired effects in the schedulability analysis, also showing a considerable performance degradation with respect to theoretical bounds that do not consider these practical implementation aspects that may be, however, often be used in practical software systems.

Contribution. In this work, we propose a method to determine the optimal number of worker threads to avoid additional delays arising from the usage of blocking synchronization mechanisms. To this end, this paper first describes the performance degradation problem, also presenting the results of a deep exploration of the thread pool behavior in TensorFlow when used in conjunction with the Eigen mathematical library. Then, we show how to design a thread pool resilient to the concurrency limitation problem by configuring the number of threads in the pool appropriately. In this way, state-of-the-art real-time analyses (e.g., [5]) can still be used to determine

[1]When used in the standard configuration, i.e., in conjunction with the Eigen mathematical library.

schedulability. We solve the problem of determining the pool size in two different ways. First, we model the problem as a flow-cut problem, and we show how this can be solved using an integer linear programming (ILP) formulation. Second, we model the problem as a maximum-weight independent set problem, and we show that it can be solved in polynomial time by converting the graph to a *comparability graph*. The paper targets a global scheduling algorithm [6]. However, Section VIII discusses how our results can be applied to federated [7] and partitioned scheduling [8, 9]. Then, in the evaluation, we compare the proposed method with the state-of-the-art solution with both synthetic tasks and realistic graphs based on state-of-the-art DNN.

II. The Concurrency Limitation Problem

Next, we review the concurrency limitation problem (Section II-A), and we describe its behavior in the context of Eigen and TensorFlow (Section II-B) by presenting the results of a deep code inspection we performed.

A. The Problem

We describe the behavior of a parallel task, modeled as a directed acyclic graph (DAG), executed by a thread pool and using blocking synchronization mechanisms, through the example reported in Fig. 1(a), which consists of a simple fork-join pattern.

The implementation of this example is reported in Alg. 1, where: **(i)** a function $v_1v_5()$ first executes the code logically associated with node v_1v_5; **(ii)** it spawns a concurrent operation consisting of three child nodes (v_2, v_3, v_4); **(iii)** upon termination of the three nodes, it executes the code associated with v_5. Child nodes (v_i, i=2,3,4 in Alg. 1) execute their code and then signal their completion to $v_1v_5()$. Without loss of generality, assume this parallel task to be executed in a pool of two threads, which can execute in parallel in two physical cores. The two threads act as computing elements for the nodes (functions) of the parallel task: therefore, as illustrated in Fig. 1, when function $v_1v_5()$ suspends to wait for v_2, v_3, and v_4, the entire first thread suspends, leaving the three child nodes with just one computing element to execute (i.e., the second thread). This is because blocking synchronization is performed by OS-level system calls, and the OS is not aware of the user-space level scheduling of individual functions executed by the pool, but just of its threads.

It follows that the usage of blocking synchronization mechanisms such as condition variables or synchronization barriers may temporarily reduce the number of available computing elements, i.e., threads, which could otherwise be used to serve the execution of some nodes. Figure 1(c) shows how the number of threads available for the execution of nodes varies with time.

Fig. 1(e) shows another serious issue that may arise when using thread pools with blocking synchronization mechanisms without carefully designing the number of threads in the pool. Indeed, Fig. 1(e) considers the case in which a pool of threads is serving two identical parallel tasks, depicted in Fig. 1(a) and (d), which use the same implementation reported in Alg. 1. As shown in the timeline, it is possible that both threads execute function $v_1v_5()$, *suspending all the threads of the pool*. In this case, as noted in [4], a deadlock can occur, since there is no other thread in which functions v_2, v_3, and v_4 can run, for each of the two parallel task instances.

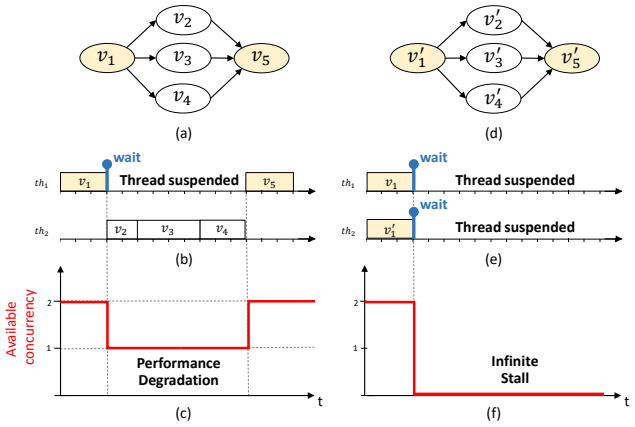

Figure 1: Effects of blocking synchronization mechanisms on the number of available threads. Inset (b) and (e) illustrate two possible execution of the graphs reported in inset (a) and (d), in which the blocking synchronization worsen the performance (inset (c)) or causes a deadlock (inset (f)).

.

However, as extensively discussed later, these shortcomings can be avoided by adequately sizing the pool of threads when adopting a global work-conserving scheduling algorithm. In this way, the concurrency limitation phenomenon can be avoided, making parallel tasks executed by thread pools and blocking synchronization mechanisms prone to be analyzed with state-of-the-art methods.

Algorithm 1 Blocking fork-join parallelism

```
1: procedure v1v5( )
2:     execute v1()
3:     fork v2(), v3(), v4()
4:     wait for v2(), v3(), v4()
5:     execute v5()
6: end procedure
1: procedure vi( )                          ▷ with i=2,3,4
2:     execute vi()
3:     signal completion to v1v5()
4: end procedure
```

B. Thread pools in Eigen

We discuss how Eigen (also adopted by TensorFlow) uses blocking synchronization mechanisms to implement precedence constraints by considering the `parallelFor()` *meta-function*, implemented in the `TensorDeviceThreadPool.h` file [10] of Eigen and also used for implementing several DNN layers in

TensorFlow.[2] A simplified version of such a function is shown in Alg. 2. `parallelFor()` takes five parameters in input: an input array I, its size N, an object describing the computational cost of the operation to be parallelized (e.g., a matrix multiplication or a max-pooling), a pointer to a thread pool, and a pointer `f` to a function to be executed. In essence, `parallelFor()` applies the function `f` to the input `I` in a parallel manner. This is done by enqueueing in the thread pool's queue multiple occurrences of `f`, each one acting on a mutually-exclusive subset of the indexes of `I`. To this end, `computeBlockParams(OpCost, pool.size())` computes the number of elements contained in each of such subsets, denoted as `blockSize` (line 2), and the overall number of blocks (i.e., subsets) `blockCount`. These parameters are computed according to the computational cost of the function `f` (described by the `OpCost` object) and the number of threads in the pool. Then, a barrier object is initialized providing `blockCount` as an input parameter, meaning that a thread calling `barrier.wait()` is awakened after `blockCount` calls to `barrier.notify()` have been issued (line 3). Then, the `handleRange()` function is called, providing the input data `I`, a range $[0, N]$ of indexes of `I`, the thread pool `pool`, and the block size. When `handleRange()` returns, `parallelFor()` blocks on the barrier (line 5). The `handleRange()` function works as follows. Given an interval `[first, last]` of indexes of the input `I`, it recursively divides the interval into two parts, each one handled by a recursive call of `handleRange()`, which are enqueued in the queues of the pool (lines 14-15). The recursion stops when the size of the interval (i.e., `last-first`) is smaller than or equal to `blockSize` (line 8): in this case, `f` is executed on the input range $[first, last]$ of `I`, and `barrier.notify()` is called, meaning that one of the `blockCount` parallel operations have been completed (line 10). When `blockCount` calls of `f` have been issued, the thread that executed `parallelFor()` is awakened and the parallel operation completes.

Fig. 2 illustrates the subgraph generated by a call to `parallelFor` with an input size $N = 100$ and `blockSize` $= 25$. The interval $[0, 100)$ is first divided into two sub-intervals $[0, 50)$ and $[50, 100)$, generating two corresponding nodes. The two intervals are recursively divided into four intervals $[0, 25)$, $[25, 50)$, $[50, 75)$, $[75, 100)$. Hence, four corresponding nodes are enqueued. Since the size of each of the four intervals is equal to 25, the condition at line 8 of Alg. 2 is fulfilled, and the function `f` is called as part of the execution of the four nodes. Then, each of them notifies the barrier object, allowing `parallelFor()` to complete.

To estimate how much blocking synchronization is used in the execution of a DNN in TensorFlow, we refer to the InceptionV3 [3] DNN.

[2]Note that newer versions of Eigen also include an asynchronous version of `parallelFor`. However, all legacy versions of TensorFlow still need to deal with the issues discussed in this paper.

Algorithm 2 Pseudocode of the parallelFor function (from Eigen [10]).

```
1:  procedure ParallelFor(I, N, OpCost, pool, f)
2:      <blockCount, blockSize>  ← computeBlockParams(OpCost, pool.size())
3:      barrier ← createBarrier(blockCount)
4:      handleRange(I, 0, N, f, pool, blockSize)
5:      barrier.wait()
6:  end procedure
7:  procedure HandleRange(first, last, pool, f, I, blockSize)
8:      if last - first ≤ blockSize then
9:          f(I, first, last)
10:         barrier.notify()
11:         return
12:     end if
13:     mid ← midIndex(first, last)
14:     pool->enqueue(handleRange, I, first, mid, f, pool, blockSize)
15:     pool->enqueue(handleRange, I, mid, last, f, pool, blockSize)
16: end procedure
```

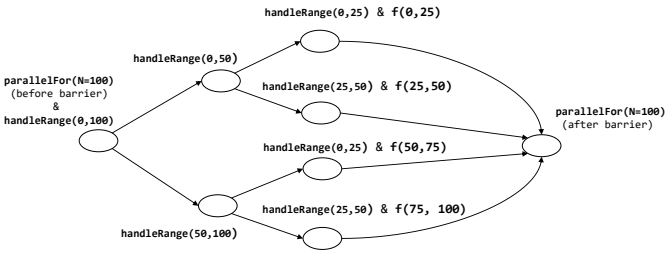

Figure 2: Sub-graph generated by a call to `parallelFor`, with $N = 100$ and `blockSize` $= 25$.

Such a network is mainly composed of average-pooling, max-pooling, concat, softmax, and convolutional layers. The first four types of layers directly use the `parallelFor()` function described in Alg. 2, hence making use of blocking synchronization mechanisms through the barrier object. The remainder of the operations is mainly convolutions, which do not use directly `parallelFor()`, but they instead rely on more complex mechanisms for tensor contraction [11] (see the file `TensorContractionThreadPool.h` [12] of Eigen). However, at the beginning of each convolution, a node starts the operation by queuing some initial nodes in the thread pool's queues. Then, the source node of the convolution waits on a barrier, hence using again blocking synchronization mechanisms. Such a node is awakened when the convolution terminates. Internally, the convolution has several internal fork-join operations. Such operations are not based on blocking synchronization mechanisms but on *atomic variables* used as counters, therefore combining blocking and

non-blocking synchronization.

This example highlights how blocking synchronization mechanisms are used in almost every layer of a deep neural network executed by TensorFlow and Eigen. Non-blocking synchronization is also used in ubiquitous layers, i.e., convolutions. Hence, we conclude that providing a method for determining the size of thread pools that allows re-using existing results for studying the timing properties of parallel tasks in the presence of both blocking and non-blocking precedence constraints is of the utmost importance to provide guarantees for DNNs executed by TensorFlow.

Which is the relevance of blocking synchronization beyond Eigen? In this section, we reported a detailed overview of how blocking synchronization can be harmful to real-time performance in the context of Eigen and TensorFlow. But the relevance of the problem goes beyond these two libraries. Indeed, both thread pools and blocking synchronization mechanisms such as barriers and condition variables are commonly used by programmers in many applications[3] with some form of timing requirements (e.g., soft real-time). Clearly, with this problem in mind, a developer designing a new application using thread pools from scratch to achieve timing predictability will never use blocking synchronization, opting for a solution with non-blocking synchronization. However, most applications are not written - as new - from scratch. Instead, it is common that pre-implemented and easy-to-use functionalities are integrated into new systems (e.g., coming from software libraries). These might use mechanisms such as condition variables to synchronize since they are widely used programming paradigms. Therefore, our efforts aim to generalize the problem beyond Eigen and TensorFlow and provide application integrators with a principled way of determining the thread pool size when they require to use it to run pre-existing software modules that make use of blocking synchronization.

Which is the relevance of finding the *optimal* size of a thread pool?

Then one may argue about the relevance of finding the *optimal* size of a thread pool, while a conservative approximation could just suffice. However, creating additional threads is not for free, as it implies the allocation of memory and induces CPU multiplexing overheads. Several prior works highlighted the relationship between the performance and the pool size, concluding that significant degradation of the average-case performance can be observed with oversized thread pools [17, 18]. Other works also show that oversized pools are problematic for performance due to system-level overheads caused by the congestion at I/O interfaces and scheduling [19].

Therefore, avoiding wasting resources on unneeded threads is essential, especially on resource-constrained embedded/edge platforms. Thus, a designer faces conflicting requirements: on the one hand, having enough threads to allow bounding response times at design time by using existing techniques but, on the other hand, avoiding introducing too many threads to minimize the overhead. Next, we propose configuring the thread pool based on this reasoning: create additional threads to enable real-time analysis, but no more than needed by *minimizing* their number.

[3]For example, thread pools are used by the Tasking Framework by the DLR, the German research center for aeronautics and space [13]–[15], and by the Task Management API (MTAPI) used as part of the Embedded Multicore Building Blocks (EMBB) [16] by Siemens).

III. System Model

This paper studies a set of parallel directed acyclic graph (DAG) tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$ running on a multi-core platform with m identical processors, where a dedicated pool of threads schedules each task. For each task $\tau \in \Gamma$, we assume at most one of its instances to be pending at the same time[4]; for this reason, for brevity, we often refer to the (only) pending instance of τ by referring to τ itself. As discussed more extensively later in the paper, the problem addressed hereafter is an *intra-task* phenomenon, i.e., it can be solved by considering each thread in isolation. Therefore, for simplicity, in the rest of the paper, we omit the notation that would be needed for denoting multiple tasks, considering an arbitrary task τ under analysis.

Task Model. Each DAG task is scheduled by a dedicated pool of threads $\Phi = \{\phi_k : k = 1, \ldots, |\Phi|\}$. Each task is described by a DAG $G = \{V, E\}$, with V denoting the set of all nodes $v_j \in V$ in the graph, and each edge $(v_p, v_s) \in E \subseteq V \times V$ denoting a precedence constraint between nodes, meaning that v_s cannot start before v_p is completed. Each node v_i represents a sequential computation. A node without incoming edges is called *source node*, whereas a node without outgoing edges is called *sink node*. A task can release a potentially infinite sequence of instances. Each instance is said to be pending from its release to when it completes. A task completes when all its sink nodes are completed. A node v is ready when the corresponding task is ready and all precedence constraints of v are satisfied.

Given an edge $(v_p, v_s) \in E$, v_p is a *direct predecessor* of v_s and v_s is a *direct successor* of v_p. We denote with the symbols $\texttt{pred}(v_i)$ and $\texttt{succ}(v_i)$ the set of predecessors and successors of v_i, which include precedence relations that are either direct (by means of an edge) or indirect (by means of multiple edges and intermediate nodes).

A thread ϕ_k is said to be *ready* when it has pending nodes to execute. Similarly, a node $v_i \in V$ is said to be *ready* when it is eligible to execute according to precedence constraints.

Scheduling. We consider threads to be scheduled under a global work-conserving scheduling algorithm. A *work-conserving* scheduler is a scheduler that never idles a core if there exists a ready thread to execute. For example, a configuration compatible with this assumption can be obtained by executing TensorFlow on Linux, e.g., configuring each thread to have a fixed priority using its `SCHED_FIFO` scheduling class. The workload executed within each thread pool Φ is also globally scheduled; namely, the workload is enqueued in

[4]This assumption is, for example, fulfilled by any schedulable task set with constrained deadlines.

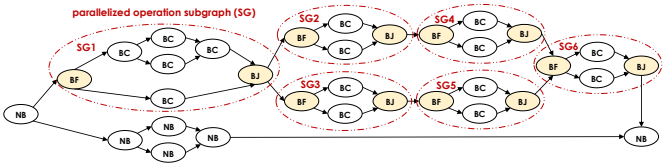

Figure 3: Example of graph compliant with the system model of Section III.

a single (*logical*) queue, which can be accessed by all the threads ϕ_k of the pool Φ.

Node types. Following [4], we model precedence constraints obtained with and without blocking synchronization mechanisms with a node type $x_i \in \mathcal{X} = \{\texttt{BF}, \texttt{BJ}, \texttt{BC}, \texttt{NB}\}$ associated with each node v_i, with the following meaning:

- A node of type `BF` (*blocking fork*) exhibits the behavior shown for node `v1` of Alg. 1 (i.e., lines 2,3, and 4 of v_1v_5`()`), i.e., **(i)** it possibly perform some computation, then **(ii)** spawns some child nodes and **(iii)** as its last operation, it waits on a synchronization barrier, causing the thread that served it to *suspend*. The set of all nodes of type `BF` is defined as $V^{\texttt{BF}} = \{v_i \in V : x_i = \texttt{BF}\}$.
- A node of type `BJ` (*blocking join*) is associated with a node of type `BF` and executes when all the child nodes spawned by the corresponding `BF` node signaled their completion (e.g., node `v5` of Alg. 1, line 5 of `v1v5()`). For each $v_i \in V^{\texttt{BF}}$ the function $\mathcal{J}(v_i)$ denotes the associated `BJ` node.
- The type `BC` (*child of blocking nodes*) is assigned to each node $v_i \in V : x_i \notin \{\texttt{BF}, \texttt{BJ}\}$ in a sub-graph delimited by a pair of nodes of type `BF` and `BJ`, respectively. For each of these nodes, the function $\mathcal{F}(v_i)$ is defined to return its `BF` node, and given a `BF` node $v_f \in V^{\texttt{BF}}$, $V^{\texttt{BC}}(v_f)$ denotes the set of the associated `BC` nodes.
- Other nodes are of type `NB` (*non-blocking*).

Each pair of `BF` and `BJ` nodes then delimits a *subgraph* of nodes. The set of subgraphs using blocking synchronization is denoted as $\mathcal{S}$, and the i-th subgraph is denoted as SG_i. Subgraphs originated by pairs of nodes of type `BF` and `BJ` cannot be nested. Furthermore, it is assumed that internal nodes of each subgraph are not connected with the remainder of the graph. Specially, the `BF` and `BJ` node of a subgraph can have only incoming and outgoing edges from/to nodes outside the subgraph, respectively. Nodes of type `BC` can be connected either with other `BC` nodes of the same subgraph, with the `BF` node by means of incoming edges (to the `BJ` node), and with the `BJ` node through outgoing edges.

An example of a parallel task is reported in Fig. 3, where the node type is reported for each node. As it is shown in Fig. 3 each subgraph SG that is delimited by each pair of nodes of type `BF` and `BJ` contains nodes that are not directly connected with the remainder of the DAG, as it happens in the Eigen-level parallel implementation of a DNN layer. The model also supports hybrid graphs where both fork-join structures using blocking and non-blocking synchronization co-exist (as in the lower part of Fig. 3).

Notation. The main notation adopted throughout the paper is summarized in Section I.

Table I: Main notation adopted throughout the paper.

Symbol	Description
m	number of cores
τ	an arbitrary task under analysis
Φ	the threadpool that manages τ
$\lvert\Phi\rvert$	the number of threads of Φ
ϕ_k	the k-th thread of Φ
G	the DAG of τ_i
V	the set of nodes in G
E	the set of edges in G
v_i	a node $v_i \in V$
(v_i, v_j)	an edge in E
$\texttt{pred}(v_i)$	the set of predecessors of v_i
$\texttt{succ}(v_i)$	the set of successors of v_i
$\mathcal{X}$	the set of node types ($\mathcal{X} = \{\texttt{BF}, \texttt{BJ}, \texttt{BC}, \texttt{NB}\}$)
x_i	the node type of v_i
$V^{\texttt{BF}}$	the set of nodes of type `BF`
$V^{\texttt{BC}}(v_f)$	the set of `BC` nodes of $v_f \in V^{\texttt{BF}}$
$\mathcal{J}(v_f)$	the `BJ` node associated with $v_f \in V^{\texttt{BF}}$
$\mathcal{F}(v_i)$	the `BF` node v_f associated with $v_i \in V^{\texttt{BC}}(v_f)$
$\mathcal{S}$	the set of subgraphs using blocking synchronization
SG_i	the i-th subgraph in $\mathcal{S}$
$p(\tau, t)$	current parallelism of a task τ at time t
$d(\Phi, \tau, t)$	desired concurrency of the thread pool Φ at t
$l(\tau, t)$	available concurrency of the thread pool Φ of τ at t

IV. Problem Formalization

We start by providing some definitions useful to formalize the problem addressed in this paper. We begin with the concept of *current parallelism of a DAG task*.

Definition 1 (Current parallelism of a task τ)**.** *The current parallelism of a task τ at a time t is the number $p(\tau, t)$ of nodes of τ that can run in parallel according to precedence constraints at time t in an arbitrary schedule.*

It is worth noting that the notion of parallelism is, in general, different from the notion of concurrency because the latter is subject to the additional constraint that the task runs on a physical machine with a bounded number of cores. This leads to the following definition.

Definition 2 (Desired concurrency of the thread pool Φ)**.** *The desired concurrency $d(\Phi, \tau, t)$ of a thread pool Φ at a time t is the maximum number of threads that would be able to run in parallel with m cores given the current parallelism of the graph if no `BF` node suspended any thread of Φ, i.e., $d(\Phi, \tau, t) = min(p(\tau, t), m)$.*

Intuitively, the desired concurrency $d(\Phi, \tau, t)$ is the maximum allowed parallelism that τ can have at t on a physical platform with m core when all the threads of its pool are

simultaneously executing, *and none of them is suspended due to blocking synchronization*. Note that, in the presence of multiple tasks handled by different pools, the actual concurrency can be lower than $d(\Phi, \tau, t)$ even without using blocking synchronization mechanisms. However, such a reduction is due to the scheduling of threads at the operating system level, whose effect is already accounted for in state-of-the-art real-time analyses (e.g., [20]).

However, the number of worker threads that can actually be used to serve nodes of τ may be less than $d(\Phi, \tau, t)$ due to the usage of blocking synchronization. Therefore, we introduce the concept of *available concurrency*.

Definition 3 (Available Concurrency)**.** *The available concurrency $l(\tau, t)$ of a pool Φ at time t is defined as the number of threads in Φ that are not suspended due to blocking synchronization, i.e., those which are ready to execute workload at t. Threads in Φ that are not suspended due to blocking synchronization are said to be* available threads *of Φ.*

The available concurrency is graphically shown in insets (c) and (f) of Fig. 1.

Therefore, a *performance degradation* occurs when the desired and available concurrency does not match.

Definition 4 (Performance Degradation due to Limited Concurrency)**.** *A task $\tau \in \Gamma$ managed by a pool Φ is said to suffer performance degradation due to limited concurrency at a time t when $l(\tau, t) \leq d(\Phi, \tau, t)$, i.e., when the number of available threads at t is less than the one that would have been available when blocking synchronization is not used.*

From a real-time analysis standpoint, the condition $l(\tau, t) \leq d(\Phi, \tau, t)$ has further implications. Indeed, if not satisfied, it means that the parallel application *cannot* be analyzed with state-of-the-art real-time analysis techniques for DAG tasks under global scheduling because they do not account that a computational unit (in this case, a thread) might be unavailable due to the usage of blocking synchronization mechanisms.

Therefore, the key research question of this paper is defined as follows.

Research Question. *How can the proper thread pool size $|\Phi|$ be selected to avoid, by design, the performance degradation phenomenon due to limited concurrency?*

The key observation is the following. Suppose the thread pool is properly (over)-dimensioned. In this case, the reduced-concurrency phenomenon can be eliminated: this is because when thread blocks due to a `BF` node, there is always another thread ready to run in place of the blocked one, if needed, since the global scheduler will select it to run due to the work-conserving property. Therefore, we show next how to determine the optimal (i.e., minimal) number of threads to configure a thread pool for its parallel task.

V. Determining the size of the pool

First, note that since each task $\tau_i \in \Gamma$ is assigned to a dedicated pool Φ, the concurrency limitation is an *intra-task* phenomenon, i.e., the execution of `BF`, `BC`, and `BJ` nodes of τ_i affect only the number of available threads of Φ. Then, it follows that we can study the concurrency limitation problem separately for each task $\tau_i \in \Gamma$.

Based on previous observations, Lemma 1 states a condition to avoid performance degradation due to limited concurrency.

Lemma 1. *If*

$$\forall t \geq 0,\ l(\tau, t) \geq d(\Phi, \tau, t) \tag{1}$$

then τ_i does not suffer performance degradation during its execution.

Proof. It directly follows by applying Definition 4 $\forall t \geq 0$. □

Therefore, the next step is to compute $l(\tau, t)$. To this end, we leverage the following two observations.

Observation 1. *The available concurrency $l(\tau, t)$ decreases by one every time a `BF` node terminates its execution.*

Observation 2. *The available concurrency $l(\tau, t)$ increases by one every time a `BJ` node becomes ready.*

This leads to the following definition.

Definition 5. *Given an arbitrary schedule at a time t, $n^B(t)$ denotes the number of `BF` nodes $v_j \in V^{BF}$ of $\tau_i \in \Gamma$ that are completed at time t while the corresponding `BJ` node $\mathcal{J}(v_j)$ is not ready at t.*

We can now derive a closed-form formula for $l(\tau, t)$.

Lemma 2. *Given an arbitrary schedule and a time t, the available concurrency $l(\tau, t)$ is*

$$l(\tau, t) = \max(0, |\Phi| - n^B(t)), \tag{2}$$

Proof. Note that the available concurrency is always greater than or equal to zero. Given an arbitrary schedule and a time t, there are $n^B(t)$ threads blocked (Definition 5), where each one reduces the available concurrency by one (by Observation 1 and Observation 2). Therefore, with a thread pool of size $|\Phi|$, there are $|\Phi| - n^B(t)$ available threads at t. □

However, using Lemma 1 and Eq. (2) requires dealing with an impractical continuum due to the presence of the universal quantifier and the dependency on time t. Therefore, we seek to find for a value $l(\tau)$, independent of the time, such that $l(\tau) \leq l(\tau, t)$ holds for any possible schedule and $\forall t \geq 0$. Given Eq. (2), $l(\tau)$ is found by searching the maximum number n^B of `BF` nodes $v_j \in V^{\text{BF}}$ of $\tau \in \Gamma$ that completed while the corresponding `BJ` node $\mathcal{J}(v_j)$ is not ready yet *in any possible schedule* and then using

$$l(\tau) = \max(0, |\Phi| - n^B). \tag{3}$$

The derivation of n^B is extensively discussed next in Section VI. If $l(\tau)$ is known, the thread pool size $|\Phi|$ can be set in such a way to guarantee the condition of Lemma 1.

However, note that also $d(\Phi, \tau, t) = min(p(\tau, t), m)$ depends on t through $p(\tau, t)$. This dependency with the time

can be solved in two ways: **(i)** by assuming that the minimum is always resolved to m, thus assuming that the parallel task is parallel enough to always occupy all the physical core of the platform; or **(ii)** by finding a bound to the maximum parallelism of the DAG that does not depend on the time (a polynomial time method is discussed in Section VI-B).

Independent on which of these methods is used to compute the maximum parallelism, let $d(\Phi, \tau)$ be a bound on $d(\Phi, \tau, t)$ such that $\forall t \geq 0, d(\Phi, \tau) \geq d(\Phi, \tau, t)$.

Then, Theorem 1 states a condition on how to set the thread pool size $|\Phi|$ to avoid performance degradation due to limited concurrency.

Theorem 1. *If*

$$|\Phi| \geq d(\Phi, \tau) + n^B, \tag{4}$$

then the corresponding task $\tau \in \Gamma$ does not suffer performance degradation due to limited concurrency.

Proof. By Lemma 1, $\tau \in \Gamma$ does not suffer performance degradation due to limited concurrency if $\forall t \geq 0,\ l(\tau, t) \geq d(\Phi, \tau, t)$. By definition of $l(\tau)$ and $d(\Phi, \tau)$, $\forall t \geq 0,\ l(\tau) \geq l(\tau, t)$ and $\forall t \geq 0, d(\Phi, \tau) \geq d(\Phi, \tau, t)$ Therefore, by Eq. (3), it follows $l(\tau) = |\Phi| - n^B \geq d(\Phi, \tau)$. The theorem follows. □

Theorem 1 provides a condition for setting the size of the thread pool in such a way to avoid performance degradation due to the usage of blocking synchronization. This is obtained by possibly having more threads in the pool than total concurrency. In turn, this may imply having more threads than the number of cores: in this case, if $d(\Phi, \tau, t)$, at most m threads can run in parallel, with the other $|\Phi| - m$ threads in a ready (but not executing) state that are set to replace any running thread that suspends on a blocking synchronization barrier, thus avoiding the available concurrency to fall below the desired concurrency.

VI. Deriving the exact n^B

Next, we propose two possible problem formulations to derive the exact n^B. The first one directly leverages Observation 1 and Observation 2, modeling the problem as a *flow-cut*, and proposing a mixed-integer linear formulation that finds a solution without requiring any graph transformation. The second solution instead formulates the problem as a *maximum weighted independent set problem*, and it allows finding n^B in polynomial-time after transforming the graph to a comparability graph.

A. Modeling the problem as a flow-cut

We start defining the concept of *flow-cut* of a DAG.

Definition 6 (Flow-cut)**.** *A flow-cut of a DAG G is defined as a partition of the set of nodes V_i into two disjoint sets V_1 and V_2, such that $\forall v_i \in V_2,\ \ succ(v_i) \cap V_1 = \emptyset$, with $V_1 \cap V_2 = \emptyset$ and $V_1 \cup V_2 = V$.*

Next, we show that n^B can be found by modeling the problem as a flow-cut. We start with a key observation.

Observation 3. *A subgraph $SG_i \in \mathcal{S}$ causes a reduction of the available concurrency if and only if: **(i)** the BF node completed its execution, and **(ii)** at least one of its BC nodes are pending.*

In Observation 3, (i) follows from Observation 1, and (ii) from Observation 2, as the BJ node of a subgraph SG_i becomes ready only when *all* the BC nodes of the same subgraph SG_i completed.

Based on the previous observation, Theorem 2 shows that the problem of finding $n^B(t)$ can be solved as a flow-cut problem.

Theorem 2. *$n^B(t)$ can be computed by counting the number of nodes $v_f \in V^{BF}$ in the first partition V_1 that have at least one of the corresponding BC nodes $v_i \in V^{BC}(v_f)$ in V_2, with V_1 and V_2 defined as in Definition 6.*

Proof. We consider an arbitrary instance of τ, an arbitrary time t, and an arbitrary flow-cut, with the following meaning for sets V_1 and V_2: V_1 includes all the nodes that have already completed in the considered instance of τ at an arbitrary point in time t, and V_2 all the nodes that are not completed yet. For any possible schedule following the precedence constraints, this definition of V_1 and V_2 is a valid flow-cut according to Definition 6 and includes all possible valid schedules according to precedence constraints.

By Definition 6, $n^B(t)$ counts the number of BF nodes $v_j \in V^{\text{BF}}$ of τ that completed at time t while the corresponding BJ node $\mathcal{J}(v_j)$ is not ready at t. Therefore, leveraging Observation 3, $n^B(t)$ is found in one of the flow-cuts where: **(a)** the BF node completed, **(b)** at least one of the corresponding BC nodes not completed yet, thus preventing the BJ node from becoming ready for execution. This means that a subgraph $\text{SG}_x \in \mathcal{S}$ needs to have at least one BC node in V_2 to be included in $n^B(t)$. Therefore, it follows that $n^B(t)$ can be found by counting the number of BF nodes in V_1 (i.e., completed) that have at least one of its BC nodes in V_2 (i.e., not completed). □

By leveraging Theorem 2, the following corollary follows.

Corollary 1. *n^B can be computed by counting the number of nodes $v_f \in V^{BF}$ in V_1 in the flow cut that maximizes the number of BF nodes in V_1 that have at least one of its BC nodes in V_2.*

A solution based on an ILP formulation. The flow-cut problem presented in Theorem 2 can be solved with an integer linear programming formulation, using the following variables:

- For each edge $(v_j, v_x) \in E$, $\text{CT}_{j,x} \in \{0, 1\}$ is equal to 1 if and only if $v_j \in V_1$ and $v_x \in V_2$.
- For each node $v_j \in V$, $\text{V1}_j \in \{0, 1\}$ is equal to 1 if and only if $v_j \in V_1$.
- For each node $v_f \in V^{\text{BF}}$, for each node $v_j \in V^{\text{BC}}(v_f)$, $\text{BN}_{f,j} \in \{0, 1\}$ is equal to 1 if and only if $v_f \in V_1$ and $v_j \in V_2$.
- For each node $v_f \in V^{\text{BF}}$, $\text{BG}_f \in \{0, 1\}$ is equal to 1 if and only if $v_f \in V_1$ and, for at least one of its BC nodes $v_j \in V^{\text{BC}}(v_f)$, it holds $v_j \in V_2$.

The objective function maximizes the number of BF nodes that are in V_1 while at least one of its BC node is in V_2, i.e., **maximize** $\sum_{v_f \in V^{\text{BF}}} \text{BG}_f$.

The optimization problem is subject to the following set of constraints. We use the so-called big-M formulation, where the symbol M is defined to denote a large constant representing infinity.

Constraints. A variable $\text{CT}_{j,x}$ is equal to one if and only if the edge connecting v_j and v_x are literally *cutting* the graph into the two disjoint sets, i.e., if $v_j \in V_1$ and $v_x \in V_2$: differently, V1_j is equal to one for any node belonging to V_1, also if it does not have an edge connecting to a node in V_2. Constraint 1 enforces the definition of $\text{CT}_{j,x}$ and V1_j.

Constraint 1. *For each* $(v_j, v_x) \in E$,

$$CT_{j,x} \geq V1_j - V1_x. \tag{5}$$

For each $(v_j, v_x) \in E$ *and* $v_l \in \texttt{pred}(v_j) \cup v_j$,

$$V1_l \geq CT_{j,x}. \tag{6}$$

For each $(v_j, v_x) \in E$ *and* $v_s \in \texttt{succ}(v_x) \cup v_x$,

$$V1_s \leq 1 - CT_{j,x}. \tag{7}$$

Proof. For each edge $(v_j, v_x) \in E$, the first constraint ensures that, if $v_j \in V_1$ and $v_x \in V_2$, then (v_j, v_x) is in the cut. This occurs when $\text{V1}_j = 1$ and $\text{V1}_x = 0$, which enforces $\text{CT}_{j,x} \geq 1$. In all other cases, the first constraint has no effect as it enforces either $\text{CT}_{j,x} \geq -1$ or $\text{CT}_{j,x} \geq 0$.

For each edge $(v_j, v_x) \in E$ and for each $v_l \in \texttt{pred}(v_j) \cup v_j$, we enforce that if (v_j, v_x) is in the cut, then v_j and all its predecessors are in V_1. This is guaranteed by the second constraint, which enforces $\text{V1}_l \geq 1$ to all predecessors if (v_j, v_x) is in the cut, and it has no effect otherwise ($\text{V1}_l \geq 0$). Similarly, for each edge $(v_j, v_x) \in E$ and for each $v_s \in \texttt{succ}(v_x) \cup v_x$, we enforce that if (v_j, v_x) is in the cut, then v_x and all its successors are *not* in V_1. This is guaranteed by the third constraint, which enforces $\text{V1}_s \leq 0$ to all successors if (v_j, v_x) is in the cut, and it has no effect otherwise ($\text{V1}_s \leq 1$). □

A variable $\text{BN}_{f,j}$ is equal to one if and only if a BF node $v_f \in V^{\text{BF}}$ is in V_1 and the other node $v_j \in V^{\text{BC}}(v_f)$ is in V_2. Constraint 2 enforces this definition.

Constraint 2. *For each* $v_f \in V^{BF}$, *for each* $v_j \in V^{BC}(v_f)$,

$$BN_{f,j} \geq V1_f - V1_j, \quad BN_{f,j} \leq V1_f, \quad BN_{f,j} \leq 1 - V1_j, \tag{8}$$

Proof. The constraint enforces $\text{BN}_{f,j} = 1$ if v_f is in $V_{i,1}$ and v_j is not in V_1. This occurs when $\text{V1}_f = 1$ and $\text{V1}_j = 0$, which enforces $\text{BN}_{f,j} = 1$. For all other values of V1_f and V1_j, the constraint enforces $\text{BN}_{f,j} = 0$. □

Finally, Constraint 3 enforces the definition of BG_f, which is the set of variables subject to the maximization. Variables BG_f extends variables $\text{BN}_{f,j}$ by selecting only nodes $v_f \in V^{\text{BF}} \cap V_1$ that have *at least one* of their BC nodes in V_2. For the last constraint, we define the following decision variable.

- For $v_f \in V^{\text{BF}}$ and for each $v_j \in V^{\text{BC}}(v_f)$, the boolean variable $\text{A}_{f,j}$ is defined such that $\sum_{v_j \in V^{\text{BC}}(v_f)} \text{A}_{f,j} = 1$.

Variables $\text{A}_{f,j}$ serve the purpose to count *only once* the reduction of concurrency due to a node $v_f \in V^{\text{BF}}$ through variables $\text{BN}_{f,j}$.

Constraint 3. *For each* $v_f \in V^{BF}$, *for each* $v_j \in V^{BC}(v_f)$,

$$BG_f \leq BN_{f,j} + (1 - A_{f,j}) \cdot M, \quad BG_f \geq BN_{f,j} - (1 - A_{f,j}) \cdot M, \tag{9}$$

Proof. The constraint enforces BG_f to be equal (i.e., both $\geq$ and $\leq$) to one (since $\sum_{v_j \in V^{\text{BC}}(v_f)} \text{A}_{f,j} = 1$) of the $\text{BN}_{f,j}$, i.e., the one with $\text{A}_{f,j} = 1$. In all the other cases, the constraint has no effect (i.e., it is equivalent to enforce $\text{BG}_f \leq \infty$ and $\text{BG}_f \geq -\infty$). The solver is forced to selected a value $\text{BN}_{f,j} = 1$, if any, due to the maximization in the objective function. The constraint follows. □

It is worth noting that the solution obtained by the proposed ILP formulation returns the *exact* n^B (as $n^B = \max \sum_{v_f \in V^{\text{BF}}} \text{BG}_f$) and not an upper bound: indeed, as noted in the proof of Theorem 2, there exist an actual possible schedule according to the precedence constraints leading to the computed n^B. This allows setting the optimal size for the thread pool. The proposed solution is flexible, and it can be applied directly on the DAG G with no transformation and allows extending the problem with additional modeling features by just adding other constraints. However, it can suffer from scalability issues in the case of large DAGs. Therefore, we discuss next a different solution to compute n^B using a graph transformation.

B. Computing n^B as a Max-Weight Independent Set problem

This section shows how to map the problem of computing n^B to a *maximum weight independent set* [21] problem by means of a graph transformation, with the key advantage of enabling a polynomial-time solution.

Given a subgraph $\text{SG}_i \in \mathcal{S}$, we define the *concurrency-reduction block* CR_i as a special type of node that is pending if at least one BC node $v_j \in \text{SG}_i$ is pending.

Definition 7 (Transformed Graph)**.** *Given a DAG* $G = (V, E)$, *the transformed graph* $G' = (V', E')$ *is obtained by substitution of all the* BC *nodes of each subgraph* $SG_i \in \mathcal{S}$, *with a corresponding concurrency-reduction block* CR_i.

By construction, when a concurrency-reduction block is pending, the available concurrency reduces by one. Therefore, the problem reduces to finding the independent set of concurrency-reduction blocks with maximum cardinality in G'. To start, we recall the definition of independent set.

Definition 8 (Independent Set)**.** *A set* $\mathcal{I} \subseteq V$ *is an independent set if no two nodes in* $\mathcal{I} \subseteq V$ *are connected in V, either directly (by means of an edge), or indirectly (via intermediate nodes and multiple edges).*

By associating a weight equal to *one* to each concurrency-reduction block and a weight equal to zero to each other node

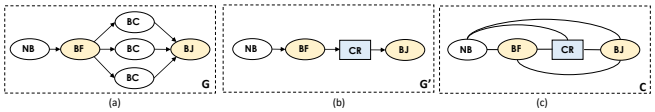

Figure 4: Graph transformations to compute n^B as a max-weight independent set problem in polynomial time.

in the graph G', the independent set of concurrency-reduction blocks with maximum cardinality can be found as an instance of the maximum-weight independent set problem.

Theorem 3. *n_i^B can be found as the sum of the weights of the set resulting from a maximum-weight independent set problem after assigning to each $CR_i \in G'$ a weight equal to* one*, and a weight equal to* zero *to each other node.*

Proof. By Observation 3, a subgraph $\text{SG}_i \in \mathcal{S}$ reduces the available concurrency by one when **(i)** the BF node completed its execution, and **(ii)** at least one of its BC nodes are pending. By definition, a concurrency reduction block CR_i of a subgraph $\text{SG}_i \in \mathcal{S}$ is pending when at least one of its BC nodes is pending. Therefore, the maximum number of n_i^B is reached when the maximum number of concurrency-reduction blocks are pending at the same time according to precedence constraints. Since each $\text{CR}_i \in G'$ has a weight equal to *one*, and all other nodes a weight equal to *zero*, the problem is equivalent to maximizing the weighted independent set (to count only concurrency-reduction blocks that can be pending at the same time) of nodes in G'. n_i^B can then be computed as the sum of the weights in the resulting set. The theorem follows. □

This problem is NP-hard [6, 22] for arbitrary graphs. However, it can be solved in polynomial time for *comparability graphs* by applying the Ellipsoid method [23]. Therefore, we next recall the definition of comparability graph and we show how to convert G' into a comparability graph.

Definition 9 (Comparability Graph)**.** *A comparability graph C is an undirected graph in which it is possible to orient each edge such that the resultant graph has the following properties:*

1) ***Anti-symmetry:** if an edge (v_i, v_j) exists, then an edge (v_j, v_i) does not.*
2) ***Transitivity:** if two edges (v_i, v_j) and (v_j, v_l) exist, then also (v_i, v_l) exist.*

Both G and G' can be converted into comparability graphs: indeed, anti-symmetry is respected by the definition of DAG (because it is acyclic), and transitivity is satisfied by adding "dummy edges" from each node to all its successors, without altering the edge's semantics of a precedence constraint. Therefore, the comparability graph C of G' is obtained by applying the transitive closure and removing orientation.

Fig. 4 summarizes the transformations required to apply the method proposed in this section. Once C is obtained, it is possible to solve the maximum weight independent set problem (via the Ellipsoid method [23]) in polynomial time and computing n^B as the sum of the weights in the set.

The maximum parallelism of a DAG. In Section V, we noted that also $d(\Phi, \tau, t) = min(p(\tau, t), m)$ depends on t through $p(\tau, t)$, which is needed for applying Lemma 1. Similar to the problem just discussed, also the problem of finding the maximum parallelism can be solved as a maximum independent set problem, which is NP-hard for general graphs [22]. Also in this case, however, the problem can be solved using comparability graphs for the DAGs considered in this paper in polynomial time. Alternatively, other polynomial-time algorithms have been proposed for specific cases, e.g., fork-join tasks [6].

VII. Previous work on upper-bounding n_i^B

Next, we discuss two methods to obtain an upper-bound on n_i^B. The first one comes from [4]. The authors of [4], while solving the problem of guaranteeing the absence of deadlocks, noted that, for each arbitrary node $v_i \in V$, the BF nodes that can affect the execution of v_i are those contained into the set:

$$\mathcal{X}(v_j) = \begin{cases} \{v_x \in V_i^{\text{BF-par}}(v_j)\} \cup \mathcal{F}(v_j) & \text{if } x_j = \texttt{BC} \\ \{v_x \in V_i^{\text{BF-par}}(v_j)\} & \text{otherwise,} \end{cases} \quad (10)$$

where $V^{\text{BF-par}}(v_i) = \{v_j \in V^{\text{BF}} \setminus (\texttt{pred}(v_i) \cup \texttt{succ}(v_i))\}$. Then, they propose to bound n^B as the maximum cardinality of $\mathcal{X}(v_i)$ for each $v_i \in V$. However, this method (which is referred to as UB−1 hereafter in this paper) may provide a much larger estimate of n^B with respect to what can be actually observed in a real schedule. As an example, consider the DAG of Fig. 3: while the approach proposed in this paper (which is referred to as EXACT hereafter) returns $n_i^B = 2$, UB−1 bounds n_i^B with 6, thus causing a significant over-dimensioning of the thread pool using Theorem 1. This is because subgraphs originated by pairs of BF and BJ nodes connected in series are subject to a precedence constraint: for example, subgraphs SG2 and SG4 (or SG3 and SG5) cannot have simultaneously their BF nodes completed while the corresponding BJ node is not ready yet. Therefore, they cannot reduce the available concurrency simultaneously as there can be only at most one pending instance of each task. An improved bound, not presented in [4], (called UB−2) on n_i^B is obtained by observing that subgraphs connected in series can never have more than *one* BF node that is completed with its corresponding BJ node is not ready. By leveraging this observation, we can bound n_i^B similarly as for UB−1 but accounting only once for sequences of BF-BJ subgraphs connected back-to-back (i.e., the BJ node of a subgraph has only one outgoing edge connecting to the BF node of a following BF-BJ subgraph, which in turn has only one incoming edge). In this way, with UB−2 n_i^B is bounded with 4 for the graph of Fig. 3, since the subgraphs pairs SG2 and SG4, and SG3 and SG5, are counted only once for each pair, respectively. Both UB−1 and UB−2 have a polynomial-time complexity in the number of nodes and edges. These methods are used for comparison in the evaluation.

VIII. Applying the proposed solution

This section presents potential cases that can benefit by applying the proposed solution.

A. Federated Scheduling

Background. The federated scheduling paradigm [7] divides a set of periodic parallel implicit-deadline tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$ with period T_i and deadline $D_i = T_i$ that need to run in a multicore platform into two disjoint sets: *heavy* and *light* tasks. A task τ_i is characterized by its *volume* C_i, i.e., the sum of the WCETs of all its nodes, and its *critical-path length* L_i, i.e., the largest sum of WCETs over a path of the graph of τ_i. A task τ_i is a heavy task if its utilization $U_i = C_i/T_i$ is greater than one; otherwise, it is a light task. Each heavy task is allocated to a dedicated cluster of processors, where global scheduling is applied. Light tasks run sequentially on a cluster composed of the remaining cores. For each heavy task τ_i, the number of dedicated cores to be assigned is computed as

$$h_i = \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil \tag{11}$$

A task set is then deemed schedulable if there are enough cores to accommodate all heavy tasks and light tasks are schedulable on the remaining cores according to a multiprocessor scheduling algorithm [7].

Federated Scheduling with Thread Pools. Next, we discuss how to apply federated scheduling when using thread pools and blocking synchronization. For simplicity, in the definition of $d(\Phi, \tau, t)$ in Section V we considered the number of cores m. However, the approach can be generalized to federated scheduling. We start by considering heavy tasks first. A thread pool Φ handling a heavy task τ_i using blocking synchronization can be configured as follows:

1) Compute the number of required cores h_i with Eq. (11).
2) Compute the thread pool size $|\Phi_i|$ using Theorem 1 after redefining the desired concurrency (Definition 2) to account for a subset of h_i cores instead of all the m cores available in the platform, i.e., $d(\Phi_i, \tau_i, t) = min(p(\tau_i, t), h_i)$.
3) Create a thread pool with $|\Phi_i|$ threads and configure them to run exclusively on h_i cores in an exclusive manner. For example, in Linux, this can be done by using `sched_setaffinity` system call.

Since light tasks are treated as sequential tasks, we can apply the same procedure reported above, considering $h_i = 1$, and configuring the threads of the pool to run (non-exclusively, in this case) on the remaining cores not used by heavy tasks.

B. Deadlock Avoidance

As shown in Section II-A, the concurrency reduction can also lead to deadlocks in some cases. Clearly, guaranteeing the absence of performance degradation, as proposed in this paper, automatically guarantees also the absence of deadlocks. Prior work [4] provided specific conditions to ensure the absence of deadlocks at design time, which rely on determining n^B. However, [4] provided only a coarse upper-bound on n^B. Combining the conditions in [4] with the exact method for determining n^B proposed in Section VI is also beneficial to better dimensioning thread pools of non-real-time or soft-real-time systems using blocking synchronization to implement fork-join parallelism.

C. Partitioned scheduling

Under partitioned scheduling, exactly one thread for each task is statically assigned to a specific core, and also nodes of a parallel task are statically assigned to run on a particular thread [4] (and hence core). Likewise, in this case, the reduced concurrency phenomenon may cause both deadlocks and performance degradation. We conjecture that a proper thread pool design allows avoiding these issues even under partitioned scheduling. However, this would require modifications to the thread pool implementation, which should allow allocating multiple threads sharing the same work queue to each processor. This research direction is left as future work.

IX. Evaluation

This section reports on the results of two experimental studies we performed to compare the proposed solution (which is referred to as `EXACT` hereafter in this paper) with `UB-1` and `UB-2`. The first experimental study targets synthetic workloads, designed to stress the reduced concurrency phenomenon and to highlight the differences between `EXACT`, `UB-1` [4], and `UB-2`. The second experimental study compares the same algorithms but on realistic deep neural network workloads. In both of them, we aim at answering the question: *how much do we need to over provision our system, in terms of threads, to avoid performance degradation due to limited concurrency?* To this end, we define (according to Eq. (4)) and evaluate the *overprovisioning factor* $\text{OF}(y) = (n^B/y) \cdot 100$, expressed in percentage, which specifies how much the thread pool size needs to be oversized to always guarantee a total concurrency equal to y, when n^B is computed with each of the three proposed methods. In the reported experiments, we consider w.l.o.g. a platform with eight cores (i.e., $y = 8$). Furthermore, we report the running times of the proposed methods, executing them on a machine equipped with an Intel Core i7-6700K @ 4.00GHz, and using the Microsoft VC++2015 compiler. The ILP formulation has been solved with IBM CPLEX.

Synthetic Workloads. In this experimental study, we synthetically generated graphs compliant with the system model of this paper in such a way to stress the reduced concurrency phenomenon and the differences between the three methods to bound n^B. The generated graphs build upon the structure of Figure 3, where each of the parallel branches containing subgraphs SG2-SG4 and SG3-SG5 in the figure have been substituted with x^{BL} corresponding *blocks* connected in sequence. Two types of blocks have been considered: *serial blocks*, composed of a single subgraph, and *parallel blocks*, composed of four subgraphs. Subgraphs in each block are connected as shown in Fig. 5. For example, Fig. 3 represents a graph generated using serial blocks with $x^{\text{BL}} = 2$. Fig. 6

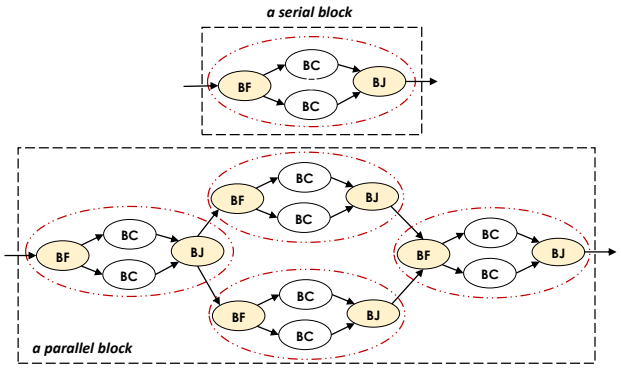

Figure 5: Serial and parallel blocks.

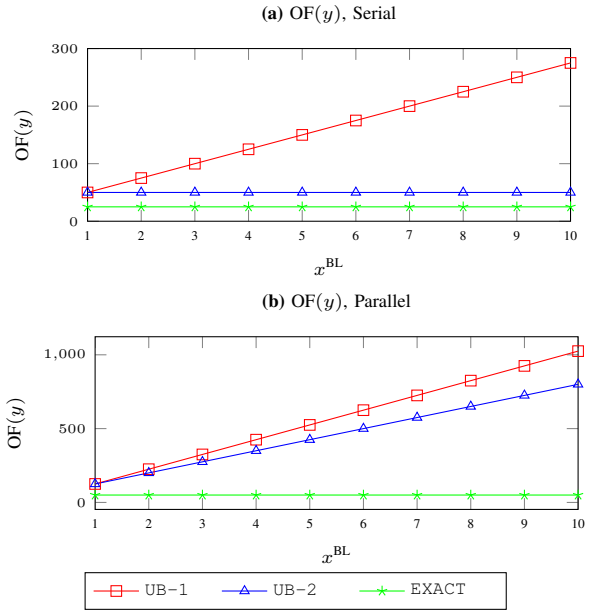

Figure 6: OF(y) for synthetic workloads.

shows that UB−1 may lead to an oversizing factor above 1000% (inset (c)), thus giving rise to the need to create many more additional threads with the negative effect of wasting an excessive amount of resources (e.g., memory) due to the pessimism in the estimation of n^B. By comparing insets (a) and (c), we observe that UB−2 gives much better results w.r.t. UB−1 in the serial block setting, while it gives much more pessimistic results w.r.t. EXACT in the case of parallel blocks, reaching OF(y) values above 800%. As expected, EXACT gives the best results, with a maximum OF(y) of 50%.

DNN Workloads. This second study aims at evaluating the performance of the proposed methods with graphs based on popular deep neural networks. In particular, we considered LeNet5 [24], AlexNet [25], GoogleNet [26], and InceptionV3 [3]. Based on the DNN structures and the interconnections of their layers, we generated the corresponding graphs by leveraging the considerations discussed in our deep inspection of the TensorFlow code reported in Section II-B. In particular, starting from the layer structure of each DNN, we substituted a subgraph using blocking synchronization to each layer that uses it (e.g., pooling or convolutional layers). The four networks have been chosen as they have quite different structures, thus allowing to test the proposed methods for different scenarios: LeNet5 and AlexNet are indeed a fully-serial sequence of layers, while GoogleNet and InceptionV3 have a much more parallel structure of layers. Fig. 7 show that all the methods are able to tightly bound OF(y) in the presence of a fully-sequential structure as those of LeNet5 and AlexNet5, while UB−1 and UB−2 again incur a notable pessimism in the presence of more parallel structures, as also observed for synthetic workloads. In particular, UB−1 and UB−2 reach OF(y) above 140% and 130%, respectively, while EXACT never exceeds 75%.

X. Related Work

To the best of our knowledge, the problem of scheduling a set of parallel real-time tasks scheduled by a pool of threads did not receive much attention from the real-time systems community. The two most closely related existing results are due to Schmid and Mottok [20] and Casini et al. [4]. However, [20] does not consider the usage of blocking synchronization mechanisms (e.g., condition variables) to implement fork-join parallelism and proposes a response-time analysis to deal with the case where the number of worker threads in a pool is less than the number of processors. Conversely, [4] considers blocking synchronization, defines the conditions to avoid deadlocks, and proposes response-time analysis techniques to account for a reduced concurrency (under both global [5] and partitioned scheduling [8]), showing significant performance degradations with respect to the case where the concurrency is not reduced (i.e., using the classical sporadic DAG model [27]). In contrast, this paper focuses on designing thread pools in such a way to fully avoid the concurrency reduction phenomenon, thus allowing reusing pre-existing results for the sporadic DAG model also when using thread pools and blocking synchronization to implement fork-join parallelism. Works from other scientific communities considered the problem of finding an optimal thread pool size, e.g., [28]–[30], but with the goal of avoiding waste of resources (e.g., memory) while still promptly responding to processing requests arriving dynamically, e.g., in the context of cloud computing [31]. None of them considered real-time constraints or the reduced concurrency phenomenon. Finally, less closely related works studied the real-time performance of specific frameworks, e.g., ROS 2 [1, 32]–[34] or OpenMP [35, 36].

XI. Conclusions

This paper proposed solutions to determine the optimal thread pool size to avoid the performance degradation phenomenon by modeling the problem both as a flow-cut problem and as a maximum-weight independent set problem. With the first modeling technique, we showed how to solve the problem

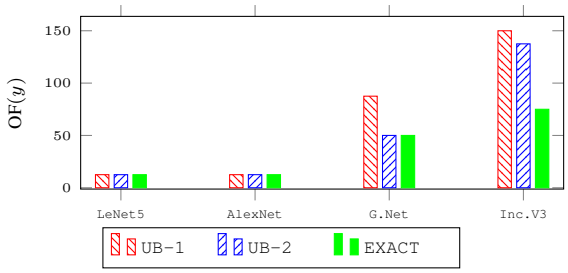

Figure 7: OF(y) for DNN workloads.

as an integer linear programming formulation, with the advantage of requiring no graph transformation and being suitable for being easily extended to consider further constraints. This comes at the expense of potentially significant running times of ILPs for large DAGs. Instead, the second formulation requires graph transformations, thus being potentially harder to implement, but it guarantees a polynomial-time complexity. We also discussed the results of a code inspection that has been conducted to understand how parallel tasks implement fork-join parallelism in Eigen. We highlighted the importance of supporting, in terms of real-time analysis theory, programming paradigms such as condition variables that are widely used beyond Eigen's use case, even for time-sensitive applications. Finally, we discussed how the results of this paper could be used under the federated and partitioned scheduling paradigms.

For future work, interesting directions comprise the study of other methods to compute n_i^B, e.g., with techniques based on the max-plus algebra [37], the design of dynamic methods to create and kill threads at runtime when other threads block on a condition variables to keep the available concurrency constant, the consideration of threads blocking on other system calls on the pool's concurrency [19], and the study of how different frameworks manage workloads run by a thread pool, e.g., the multi-threaded executor of ROS 2.

References

[1] D. Casini, T. Blaß, I. Lütkebohle, and B. B. Brandenburg, "Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling," ser. 31st Euromicro Conference on Real-Time Systems (ECRTS 2019).

[2] D. Casini, A. Biondi, and G. Buttazzo, "Timing isolation and improved scheduling of deep neural networks for real-time systems," *Software: Practice and Experience*, vol. 50, no. 9, pp. 1760–1777, 2020.

[3] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," ser. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, Nevada, United States.

[4] D. Casini, A. Biondi, and G. Buttazzo, "Analyzing parallel real-time tasks implemented with thread pools," ser. Proceedings of the 56th Annual Design Automation Conference (DAC '19), Las Vegas, NV, USA, June 2-6, 2019.

[5] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. Buttazzo, "Schedulability analysis of conditional parallel task graphs in multicore systems," *IEEE Transactions on Computers*, vol. 66, no. 2, pp. 339–353, Feb 2017.

[6] J. Fonseca, G. Nelissen, and V. Nélis, "Improved response time analysis of sporadic DAG tasks for global FP scheduling," in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, ser. RTNS '17, 2017.

[7] J. Li, J. J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah, "Analysis of federated and global scheduling for parallel real-time tasks," in *2014 26th Euromicro Conference on Real-Time Systems*, July 2014, pp. 85–96.

[8] J. Fonseca, G. Nelissen, V. Nelis, and L. M. Pinho, "Response time analysis of sporadic dag tasks under partitioned scheduling," in *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, May 2016.

[9] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo, "Partitioned fixed-priority scheduling of parallel tasks without preemptions," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, Dec 2018.

[10] "parallelFor function of Eigen." [Online]. Available: https://eigen.tuxfamily.org/dox/unsupported/TensorDeviceThreadPool_8h_source.html

[11] D. A. Matthews, "High-performance tensor contraction without transposition," *SIAM Journal on Scientific Computing*.

[12] "Eigen TensorContraction." [Online]. Available: https://eigen.tuxfamily.org/dox/unsupported/TensorContractionThreadPool_8h_source.html

[13] Z. A. H Hammadeh, T. Franz, O. Maibaum, A. Gerndt, and D. Lüdtke, "Event-driven multithreading execution platform for real-time on-board software systems," in *Proceedings of the 15th annual workshop on Operating Systems Platforms for Embedded Real-time Applications*, 2019, pp. 29–34.

[14] Z. A. H Hammadeh and O. Maibaum, "Tutorial-tasking framework: An open-source software development library for on-board software systems," in *ESWEEK 2020*.

[15] A. Lund, Z. A. Haj Hammadeh, P. Kenny, V. Vishav, A. Kovalov, H. Watolla, A. Gerndt, and D. Lüdtke, "Scosa system software: the reliable and scalable middleware for a heterogeneous and distributed on-board computer architecture," *CEAS Space Journal*, vol. 14, no. 1, pp. 161–171, 2022.

[16] "Siemens. Embedded multicore building blocks from siemens," https://github.com/ siemens/embb.

[17] D. Bonetta, A. Peternier, C. Pautasso, and W. Binder, "A multicore-aware runtime architecture for scalable service composition," in *2010 IEEE Asia-Pacific Services Computing Conference*. IEEE, 2010, pp. 83–90.

[18] D. Xu, "Performance study and dynamic optimization design for thread pool systems," Ames Lab., Ames, IA (United States), Tech. Rep., 2004.

[19] S. O. Khorasani, J. S. Rellermeyer, and D. Epema, "Self-adaptive executors for big data processing," in *Proceedings of the 20th International Middleware Conference*, 2019, pp. 176–188.

[20] M. Schmid and J. Mottok, "Response time analysis of parallel real-time dag tasks scheduled by thread pools," in *29th International Conference on Real-Time Networks and Systems*, 2021, p. 173–183.

[21] M. C. Golumbic, *Algorithmic graph theory and perfect graphs*. Elsevier, 2004.

[22] R. V. Book, "Michael r. garey and david s. johnson, computers and intractability: A guide to the theory of np-completeness," *Bulletin (New Series) of the American Mathematical Society*, vol. 3, no. 2, pp. 898–904, 1980.

[23] M. Grötschel, L. Lovász, and A. Schrijver, "The ellipsoid method and its consequences in combinatorial optimization," *Combinatorica*, vol. 1, no. 2, pp. 169–197, Jun 1981.

[24] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[25] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[26] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[27] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," ser. 2011 IEEE 32nd Real-Time Systems Symposium, 2011.

[28] Y. Ling, T. Mullen, and X. Lin, "Analysis of optimal thread pool size," *SIGOPS Oper. Syst. Rev.*, vol. 34, no. 2, p. 42–55, Apr. 2000.

[29] D. L. Freire, R. Z. Frantz, and F. Roos-Frantz, "Towards optimal thread pool configuration for run-time systems of integration platforms," *International Journal of Computer Applications in Technology*, vol. 62, no. 2, pp. 129–147, 2020.

[30] N. Costa, M. Jayasinghe, A. Atukorale, S. Abeysinghe, S. Perera, and I. Perera, "Adapt-t: An adaptive algorithm for auto-tuning worker thread

pool size in application servers," in *2019 IEEE Symposium on Computers and Communications (ISCC)*, 2019, pp. 1–6.

[31] H. Gujral, A. Sharma, and S. Mittal, "Determination of optimal thread pool for cloud based concurrent enhanced no-escape search," in *2018 Eleventh International Conference on Contemporary Computing (IC3)*, 2018, pp. 1–6.

[32] T. Blass, A. Hamann, R. Lange, D. Ziegenbein, and B. B. Brandenburg, "Automatic latency management for ros 2: Benefits, challenges, and open problems," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021.

[33] T. Blaß, D. Casini, S. Bozhko, and B. B. Brandenburg, "A ROS 2 response-time analysis exploiting starvation freedom and execution-time variance," in *2021 IEEE Real-Time Systems Symposium (RTSS)*, 2021, pp. 41–53.

[34] Y. Tang, Z. Feng, N. Guan, X. Jiang, M. Lv, Q. Deng, and W. Yi, "Response time analysis and priority assignment of processing chains on ros2 executors," in *2020 IEEE Real-Time Systems Symposium (RTSS)*, 2020, pp. 231–243.

[35] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quiñones, "Timing characterization of openmp4 tasking model," in *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2015.

[36] J. Sun, N. Guan, Y. Wang, Q. He, and W. Yi, "Real-time scheduling and analysis of openmp task systems with tied tasks," in *Proceedings of the 38th IEEE Real-Time Systems Symposium (RTSS 2017)*.

[37] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo, "Memory feasibility analysis of parallel tasks running on scratchpad-based architectures," in *IEEE Real-Time Systems Symposium (RTSS)*, 2018.