Analyzing Parallel Real-Time Tasks Implemented with Thread Pools

Daniel Casini, Alessandro Biondi, and Giorgio Buttazzo
Scuola Superiore Sant' Anna, Pisa, Italy

ABSTRACT

Despite several works in the literature targeted predictable execution models for parallel tasks, limited attention has been devoted to study how specific implementation techniques may affect their execution. This paper highlights some issues that can arise when executing parallel tasks with *thread pools*, which may lead to deadlocks and performance degradation when adopting blocking synchronization mechanisms. A new parallel task model, inspired to a realistic design found in popular software systems, is first presented to study this problem. Then, formal conditions to ensure the absence of deadlocks and schedulability analysis techniques are proposed under both global and partitioned scheduling.

KEYWORDS

Real-Time, Parallel Tasks, Thread Pools, Tensorflow, Eigen

1 INTRODUCTION

Web services, cloud-based applications, and more recently deep neural networks (DNNs) are only a few notable examples of applications characterized by a highly-parallel workload composed of a set of sequential computations subject to precedence constraints. This kind of workload is typically modeled with directed acyclic graphs (DAGs), where nodes represent sequential computations and edges precedence constraints between them. A key observation is that real parallel workload often consists of many small sequential computations (i.e., many nodes in the DAG). Just to name an example, to infer the InceptionV3 [19] DNN on a multicore platform, the standard configuration of the popular TensorFlow machine learning framework originates a DAG with more than 34000 sequential nodes, where most of them have a very small execution time. Clearly, in such cases it is impossible to create a *dedicated* thread for each node. For this reason, a common design pattern to handle parallel workload consists in the usage of *thread pools.* That is, for each parallel task, a set (i.e., a pool) of *worker threads* are created before executing the task and used to serve the execution of the nodes. Usually, an additional main thread is in charge of dispatching the nodes to the worker threads according to a work-conserving policy. For instance, the Eigen library (used by TensorFlow to handle parallel mathematical operations) adopts randomized work-stealing scheduling [1]. Furthermore, being this dispatching implemented in user-space (i.e., by the main thread), preemptions or migrations of the nodes are typically not supported. In most implementations, nodes just correspond to functions fetched from queues of the worker threads and the operating system is not aware of their existence.

Another practical aspect in the usage of thread pools concerns the way precedence constraints are implemented. To the best of our records, the most widespread approach relies on the usage of *condition variables*, which on one hand makes the software easy to write, but on the other hand may introduce bottlenecks, unpredictability, and performance degradation.

To explain the behavior of a parallel task implemented with a thread pool and condition variables, consider the example reported in Figure 1(a), which consists of a simple fork-join pattern. A very common implementation of this parallel task is reported in Listing 1: a main function v1v5 first executes node v1, then activates the three child nodes (v2, v3, v4) to be executed concurrently, and then executes the final node v5 after *the child nodes completed.* If condition variables are used to wait for the completion of the child nodes (as a synchronization barrier), the worker thread serving v1v5 will be *suspended* (see Figure 1(b)). Therefore, *the usage of condition variables may temporary reduce the number of available threads*, which could otherwise be used to make progress in the task execution, e.g., by executing some of the child nodes.

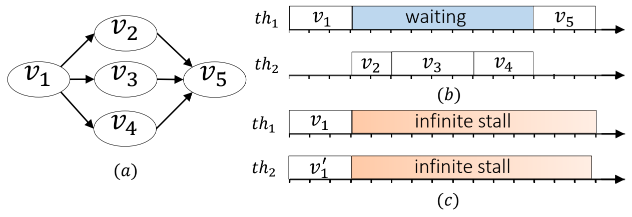

Figure 1: Inset (b) and (c) illustrate two possible execution of the graph reported in inset (a), in which the blocking synchronization worsen the performance (inset (b)) or causes a deadlock (inset (c)).

```
void v1v5() {
  <execute v1>                      void vi() { i=2,3,4
  <fork v2,v3,v4>                     <execute vi>
  <wait for v2,v3,v4>                 <signal>
  <execute v5>                      }
}
```

Listing 1: Pseudo-code for a blocking semantic.

```
                     void vi() { (i=2,3,4)
void v1() {            <execute vi>             void v5() {
  <execute v1>         <if all vi completed>      <execute v5>
  <fork v2,v3,v4>        <release v5>           }
}                    }
```

Listing 2: Pseudo-code for a non-blocking semantic.

To further complicate this issue, whenever multiple forking nodes can concurrently be executed in different threads, deadlocks can occur. For instance, if there are two replicas of the graph shown in Figure 1(a), both threads would be suspended and no child node could make progress, leading to a complete stall, as illustrated in Figure 1(c) (where v_1 and v_1' denote the root nodes of the replicas).

These issues could easily be avoided by adopting the sporadic real-time DAG task model [9], summarized in Listing 2, where each node is implemented in a dedicated function. This implementation approach, however, has the following problems: (i) it tends to be incompatible with the usage of condition variables; (ii) it contrasts the need for a coherent function context for the node that creates child nodes; and (iii) when based on dedicated per-node threads, it suffers from the shortcomings discussed at the beginning of the paper. For these reasons, many real-world software systems do not follow this approach. It is therefore of high practical relevance studying parallel tasks implemented with thread pools and condition variables; however, to the best of our knowledge, no model and real-time analysis techniques are available to handle this setting.

Contributions. This paper makes the following three contributions. First, it proposes a new model for parallel real-time tasks implemented with thread pools and with blocking synchronization mechanisms (such as condition variables) to realize precedence constraints. Second, it presents methods to detect deadlocks and build schedulability tests under global and partitioned scheduling. Finally, the proposed analysis approaches are compared with prior work to asses how the reduction of concurrency affects schedulability.

2 SYSTEM MODEL

This paper considers a set Γ of n DAG tasks $\tau_1, \ldots, \tau_n$ to be executed upon a multicore platform composed of m identical processors. Each task τ_i is scheduled by a thread pool Φ_i, which consists of a set of m threads $\{\phi_{i,j} \,:\, j = 1, \ldots, m\}$, all having the same priority π_i.

Each parallel task $\tau_i = \{G_i, D_i, T_i, \Phi_i, \pi_i\}$ is defined by a DAG $G_i = \{V_i, E_i\}$, where V_i represents the set of nodes and E_i the set of directed edges that connect the nodes. Edges denote precedence constraints between nodes. Each node $v_{i,j} \in V_i$ represents a sequential computation and is characterized by a worst-case execution time $C_{i,j}$. Each task releases a potentially infinite sequence of jobs (i.e., task instances), each separated by a minimum inter-arrival time T_i. Each job is required to complete within a relative deadline $D_i \leq T_i$ from its release time. The sets of predecessors $\texttt{pred}(v_{i,s})$ and successors $\texttt{succ}(v_{i,s})$ are defined to denote precedence constraints that are either direct (i.e., by means of an edge) or transitive (i.e., involving intermediate nodes). A node without incoming edges is referred to as a *source node*, whereas a node without outgoing edges is denoted as a *sink node*. For the sake of simplicity, this paper assumes a single sink and source node for each DAG. This is not a limitation, since any DAG with multiple source/sink nodes can always be transformed in a DAG with single source and sink node by introducing additional *dummy* source/sink nodes.

This work considers both the cases in which threads are scheduled with global and partitioned scheduling fixed-priority preemptive scheduling. When partitioned scheduling is adopted, for each pool Φ_i, each thread $\phi_{i,j} \in \Phi_i$ is statically allocated to the j-th core, and we assume the existence of a function $\mathcal{T}(v_{i,j})$ that returns the thread $\phi_{i,j} \in \Phi_i$ to which node $v_{i,j}$ is allocated.

Intra-pool scheduling. As it occurs for the threads scheduling, the workload within each pool Φ_i can be scheduled in a *global* or *partitioned* fashion. We assume that whenever global or partitioned scheduling is adopted for scheduling threads, the same policy is also adopted for intra-pool scheduling. Under global scheduling, the workload is enqueued in a single *logical* work-queue, accessible from all the computing elements (i.e., from each thread $\phi_{i,j} \in \Phi_i$)[1]. Conversely, partitioned scheduling mandates a separate work-queue for each computing element, hence requiring a partitioning phase. As reported in other works, both approaches have advantages and disadvantages [4–6]. Work-queues are managed in first-in-first-out order, i.e., no different fixed priorities are assigned to the nodes of the same task, and nodes are dispatched in a work-conserving manner.

Node types. This paper considers precedence constraints that can be implemented either in a non-blocking or blocking manner. To model the latter feature, a type $x_i \in \mathcal{X} = \{\texttt{BF, BJ, BC, NB}\}$ is associated with each node $v_{i,j}$. A node of type `BF` (*blocking fork*) originates precedence constraints with a blocking semantic, i.e., it performs some computations, spawns child nodes, and then waits for their completion on a synchronization barrier (e.g., see v_1 in function `v1v5()` of Listing 1). While waiting in the barrier, the thread that is serving the execution of the node is *suspended*, e.g., as it happens when using condition variables. A node of type `BJ` (*blocking join*) is always associated with another node of type `BF`. It executes when the corresponding `BF` node is resumed after the barrier (e.g., see v_5 in function `v1v5()` of Listing 1). All nodes $v_{i,x} \in V_i : x_{i,x} \notin \{\texttt{BF}, \texttt{BJ}\}$ included in a sub-graph delimited by a pair of nodes of type `BF` and `BJ`, respectively, are of type `BC` (*child of blocking nodes*). Formally, let $v_{i,f}$ be a node of type `BF` and $v_{i,j}$ be a node of type `BJ`, then $\forall v_{i,x} \in \texttt{succ}(v_{i,f}) \cap \texttt{pred}(v_{i,j}), x_{i,k} = \texttt{BC}$. Any other node is of type `NB` (*non-blocking*). All nodes are assumed not to suspend or block due to other mechanisms other than synchronization barriers. The node types introduced above allow modeling the example presented in Listing 1, which can represent the parallelization of an operation in TensorFlow when performed by the Eigen library.

Concurrency. The total concurrency of a parallel task τ_i is defined as the number of threads in Φ_i. Similarly, at any point in time t, the available concurrency $l(t, \tau_i)$ of a parallel task τ_i is defined as the number of threads in Φ_i that are actually *ready* to execute workload at time t, i.e., those that are not suspended on a blocking synchronization barrier. Please note that, whenever a node of type `BF` completes its execution, the available concurrency is decremented by one. Indeed, a suspended thread is not able to serve workload until it is awaken. In a dual manner, the execution of a `BJ` node increments the available concurrency by one.

Restrictions. The model considered in this work poses a restriction on the structure of the DAG describing a task, motivated by the DAG structures found in realistic software systems (such as Eigen). Specifically, it is assumed that each pair of nodes of type `BF` and `BJ` delimits a sub-graph whose internal nodes are not directly connected to the rest of the graph. Formally, let $V_i' \subseteq V_i$ be the set of vertices of one of such sub-graphs, and let $(v_{i,f}, v_{i,j})$ be the pair of vertices of type `BF` and `BJ` that delimits V_i'. Then, it is required that: **(i)** each inner node of V_i' is not connected to nodes outside the sub-graph, i.e., $\forall v_{i,x} \in V_i' \setminus \{v_{i,f}, v_{i,j}\}, \forall v_{i,a} \in V_i \setminus V_i', \nexists (v_{i,x}, v_{i,a}) \in E_i \wedge \nexists (v_{i,a}, v_{i,x}) \in E_i$; **(ii)** each edge outgoing from $v_{i,f}$ arrives in a

[1]Note that many practical implementations tend to replicate the behavior of global scheduling with work stealing [17].

node of the same sub-graph, i.e., $\forall v_{i,x} \in V_i \setminus V_i', \nexists (v_{i,f}, v_{i,x}) \in E_i$; and **(iii)** each incoming edge to $v_{i,j}$ starts from a node of the same sub-graph, i.e., $\forall v_{i,x} \in V_i \setminus V_i', \nexists (v_{i,x}, v_{i,j}) \in E_i$. Finally, it is assumed that such subgraphs cannot be nested.

3 PROVING THE ABSENCE OF DEADLOCKS

As mentioned in Section 1, the concurrent execution of multiple nodes of type BF can lead to a deadlock. Intuitively, if for a given task τ_i the available concurrency drops to zero, no thread $\phi_{i,j} \in \Phi_i$ can allow making progress in the task execution, hence causing a stall. Lemma 1 formalizes this intuition.

LEMMA 1. *If*

$$\exists\, t \geq 0 \;:\; l(t, \tau_i) = 0, \tag{1}$$

then the execution of τ_i suffers from a deadlock.

PROOF. The lemma directly follows by recalling the definition of $l(t, \tau_i)$, i.e., the number of threads $\phi_{i,j} \in \Phi_i$ available for executing workload at time t. If there exists a point in time in which no thread is available for executing nodes of τ_i, the overall graph execution cannot make progress and a deadlock occurs. □

Lemma 1 reports a simple sufficient condition for deadlocks, which is independent from the scheduling strategy (global or partitioned). Before discussing more specific sufficient conditions for deadlocks, the definition of *work-conserving* scheduler is recalled, adapting it to the case of thread pools in which the computing resource is a thread (rather than a processor).

DEFINITION 1. *An intra-pool scheduler is said to be work-conserving if it never idles one of the available threads when there exists pending workload to execute, where a thread $\phi_{i,j} \in \Phi_i$ is said to be available at time t if it is not suspended due to the execution of a node $v_{i,j} \in V_i : x_{i,j} = \mathsf{BF}$.*

The notion of *available thread* is also introduced. Building upon this definition, Lemma 2 presents a necessary condition for deadlocks under global scheduling.

LEMMA 2. *Suppose that a thread pool Φ_i adopts global work-conserving scheduling and that the execution of τ_i stalls at a certain time. Then, Equation (1) holds.*

PROOF. The proof is done by contradiction. Assume that there does not exist a time $t \geq 0 \;:\; l(t, \tau_i) = 0$ but the execution of τ_i stalls at a certain time t^*. It follows that at time t^* there must be $l(t^*, \tau_i) > 0$ available threads in Φ_i, i.e., ready to execute workload, but the task is not making progress. Since work-conserving scheduling is assumed, this is impossible and the lemma follows. □

Combining Lemmas 1 and 2, we conclude that Equation (1) is necessary and sufficient under global scheduling.

When partitioned scheduling is adopted, the execution of a task τ_i can also stall under less restrictive conditions with respect to those related to global scheduling. Indeed, the execution of a task can also stall as a consequence of the node-to-thread partitioning. For instance, consider a node $v_{i,x}$ enqueued in the work-queue of a thread $\phi_{i,w}$ *after* another node $v_{i,y}$ of type BF. Furthermore, suppose that $\phi_{i,w}$ is suspended due to $v_{i,y}$.

If $v_{i,y}$ waits for the completion of $v_{i,x}$ to be awaken (i.e., $v_{i,x}$ is of type BC), then a deadlock occurs, as $v_{i,x}$ will never be executed, because $v_{i,y}$ is before $v_{i,x}$ in the work-queue. Note that such scenarios can be avoided with an accurate node-to-thread partitioning, and the following lemma provides a necessary condition for obtaining a deadlock-free partitioning. To help the presentation of the lemma, it is necessary to introduce some accessory notation. Consider a node $v_{i,a}$ of type BC, and let $C(v_{i,a})$ be the set of nodes of type BF that may concurrently execute with $v_{i,a}$, i.e., they are not subject to precedence constraints with respect to $v_{i,a}$: formally,

$$C(v_{i,a}) = \{v_{i,x} \in V_i : v_{i,x} \notin \{pred(v_{i,a}) \cup succ(v_{i,a})\} \wedge x_{i,x} = \mathsf{BF}\}. \tag{2}$$

Also, let $\mathcal{F}(v_{i,a})$ be the node of type BF that waits for the completion of $v_{i,a}$, i.e., the one delimiting the corresponding sub-graph.

LEMMA 3. *Let*

$$\mathcal{P}(v_{i,a}) = \{\phi_{i,y} \in \Phi_i : \exists v_{i,x} \in \{C(v_{i,a}) \cup \mathcal{F}(v_{i,a})\} \wedge \mathcal{T}(v_{i,x}) = \phi_{i,y}\}$$

be the set of threads to which at least a node in $C(v_{i,a}) \cup \mathcal{F}(v_{i,a})$ is allocated to. If Equation (1) does not hold and

$$\forall v_{i,a} \in V \;:\; x_{i,a} = \mathsf{BC},\; \mathcal{T}(v_{i,a}) \notin \mathcal{P}(v_{i,a}) \tag{3}$$

then no deadlock can occur during the execution of τ_i under partitioned scheduling.

PROOF. The proof is by contradiction. Suppose that Equation (1) does not hold, Equation (3) is satisfied, but a deadlock occurs. Since Equation (1) does not hold, at any point in time during the execution of τ_i there exists at least one available thread. Hence, the deadlock must be originated by a node of type BF that is waiting for the completion of at least one node $v_{i,a}$ of type BC that is in turn waiting in a work-queue of a suspended thread $\phi_{i,w} = \mathcal{T}(v_{i,a})$. Thread $\phi_{i,w}$ must be suspended due to the execution of a node $v_{i,x}$ of type BF, which must be either (i) one of those that may concurrently execute with $v_{i,a}$, or (ii) the one that is waiting for the completion of $v_{i,a}$. Note that nodes of case (i) are included in the set $C(v_{i,a})$, while the node of case (ii) is $\mathcal{F}(v_{i,a})$. The threads to which such nodes are allocated to are those in the set $\mathcal{P}(v_{i,a})$. However, by Equation (3) $\phi_{i,w}$ is not included in $\mathcal{P}(v_{i,a})$, hence reaching a contradiction. The lemma follows. □

The lemmas proposed in this section can be applied on a per-task basis: hence, the overall absence of deadlocks can be guaranteed by applying them $\forall \tau_i \in \Gamma$.

3.1 A lower bound to the available concurrency

The previous lemmas require verifying the condition $\forall t \geq 0, l(t, \tau_i) > 0$, which involves a universal quantifier, and is hence difficult to be applied in practice. In principle, one should dispose of the value $l(t, \tau_i)$ for any possible schedule of the tasks. To overcome this issue, this section proposes a method for computing a lower bound to the available concurrency that is independent of time, i.e., $\forall t \geq 0,\ \bar{l}(\tau_i) \leq l(t, \tau_i)$. In this way, the lemmas can be applied in a much simpler (but approximate) way by just checking the condition $\bar{l}(\tau_i) > 0$ for each task. Intuitively, the proposed strategy consists in identifying the maximum number of nodes of type BF, denoted as $\bar{b}(\tau_i)$, which can affect the execution of another node. Then, $\bar{l}(\tau_i)$ can be computed as $m - \bar{b}(\tau_i)$.

$\bar{b}(\tau_i)$ can be computed by recalling that, for each node $v_{i,a} \in V_i$, the nodes of type BF that may affect the execution of $v_{i,a}$ are those contained in the set $\mathcal{X}(v_{i,a})$, where $\mathcal{X}(v_{i,a}) = \mathcal{C}(v_{i,a})$ (Eq. (2)) if $x_{i,a} \neq \mathsf{BC}$, and $\mathcal{X}(v_{i,a}) = \mathcal{C}(v_{i,a}) \cup \mathcal{F}(v_{i,a})$ otherwise. Hence, $\bar{b}(\tau_i)$ equals to the maximum cardinality of set $\mathcal{X}(v_{i,a})$ over all nodes in the graph. Given a node, set $\mathcal{X}(v_{i,a})$ can be computed in $O(n^2)$ time [16], hence the overall computational complexity required for computing $\bar{l}(\tau_i)$ is cubic in the number of nodes.

4 SCHEDULABILITY ANALYSIS

This section proposes two methods for analyzing the schedulability of the task model proposed in this paper under both global and partitioned scheduling. Despite the lemmas presented in Section 3 can be used to guarantee a deadlock-free execution, the presence of blocking precedence constraints make schedulability analysis particularly challenging. Due to space limits, this paper does not propose novel fine-grained analysis techniques, but rather shows how existing results can be adapted to handle the proposed model.

4.1 Global scheduling

This section shows how the analysis for DAG tasks presented in [14] can be modified to account for limited concurrency. This analysis aims at computing the response time R_i of each task. In this way, a task set is deemed schedulable if $\forall \tau_i \in \Gamma, R_i \leq D_i$. Before proceeding, it is necessary to recall some definitions from [14]. A path $\lambda_{i,k} = (v_{i,s}, \ldots, v_{i,e})$ is an ordered sequence of nodes, starting from the source and ending in the sink, where there is a direct precedence constraint between any two adjacent nodes. For each path, the function $len(\lambda_{i,k})$ is defined to return its length, i.e., the sum of the WCETs of all the nodes in the path. The critical path λ_i^* is defined as the path with the longest length. Finally, the volume of a task τ_i is defined as the sum of the WCETs of all its nodes, i.e., $vol(\tau_i) = \sum_{v_{i,j} \in V_i} C_{i,j}$. The response time analysis of [14] computes an upper-bound on the actual response time of each DAG task τ_i as the sum of (i) the length of its critical path λ_i^* and (ii) the interference due to higher-priority tasks or nodes $v_{i,j} \notin \lambda_i^*$, i.e., $R_i \leq len(\lambda_i^*) + I_i(R_i)$, where $I_i(L)$ is defined as the cumulative time in which, in any interval of length L, there are nodes belonging to the critical path of τ_i ready, but not executing because all the available cores (i.e., threads of the pool in our case) are busy. The analysis in [14] computes the interference $I_i(L)$ by leveraging a set of terms $I_{j,i}(L)$, each denoting a bound on the amount of workload generated by τ_j that can potentially interfere with τ_i. Then, it exploits the work-conserving property for equally-dividing the sum of such terms among all the available processors. Unfortunately, in the limited-concurrency model, the number of available threads varies over time due to the execution of nodes of type BF and BJ, and hence the bounds of [14] may not work. Lemma 4 provides a bound for the interference under the limited-concurrency model.

Lemma 4. *The interference experienced by a task τ_i in an arbitrary time window of length L when scheduled with a global work-conserving fixed-priority algorithm is upper-bounded by*

$$I_i(L) \leq \sum_{\tau_j \in \{hp(\tau_i) \cup \tau_i\}} \frac{I_{j,i}(L)}{\bar{l}(\tau_i)}, \tag{4}$$

where $hp(\tau_i)$ is the set of tasks with higher priority than τ_i.

Proof. Without loss of generality, assume that a job of τ_i under analysis is released at time 0. Consider an arbitrary time window $[t_1^i, t_2^i) \subseteq [0, L)$ in which τ_i is delayed (i.e., according to [14], τ_i is receiving interference because its critical path λ_i^* is delayed) and there are l_1^i available threads in the τ_i's pool. Since τ_i is delayed in $[t_1^i, t_2^i)$, it means that each of the l_1^i threads is either preempted by a higher-priority thread, or executing other nodes that are not part of λ_i^*. Hence, in $[t_1^i, t_2^i)$ the threads served $(t_2^i - t_1^i) \cdot l_1^i$ amount of interfering workload. Generalizing this rationale to all X intervals $[t_k^i, t_{k+1}^i)$ in which τ_i is delayed and there are l_k^i available threads, the total amount of interfering workload processed while τ_i is delayed amounts to $\mathcal{W} = \sum_{k=1}^{X} (t_{k+1}^i - t_k^i) \cdot l_k^i$. Clearly, the interfering workload $\mathcal{W}$ cannot be larger that the total amount of interfering workload that can insist in $[0, L)$, i.e., $\mathcal{I} = \sum_{\tau_j \in \{hp(\tau_i) \cup \tau_i\}} I_{j,i}(L)$. Hence, it holds $\mathcal{W} \leq \mathcal{I}$. Also, since $\bar{l}(\tau_i) \leq l_{i,k}, \forall k$ (by Sec. 3.1), it holds $\sum_{k=1}^{X} (t_{k+1}^i - t_k^i) \cdot \bar{l}(\tau_i) \leq \mathcal{W}$. Finally, note that the interference suffered by τ_i is given by the sum of the length of the intervals in which τ_i is interfered, i.e., $I_i(L) = \sum_{k=1}^{X} (t_{k+1}^i - t_k^i)$. Consequently, it holds $I_i(L) \cdot \bar{l}(\tau_i) \leq \mathcal{W} \leq \mathcal{I}$, which can be rewritten as Eq. (4). Hence the lemma follows. □

The interference $I_{j,i}$ can be of two kinds: (i) the intra-task interference due to nodes $v_{i,j} \in V_i \setminus \lambda_i^*$, and (ii) the inter-task interference due to higher-priority tasks. The former is bounded by $I_{i,i}(L) \leq vol(\tau_i) - len(\tau_i), \forall L \geq 0$ (see [9, 14]), and hence is not influenced by the presence of reduced concurrency. The latter is computed by quantifying the amount of interfering workload generated by the other tasks under a particular release pattern; specifically, by considering a release jitter equal to $R_j - vol(\tau_j)/m$, so obtaining [14]: $\forall j \neq i, I_{j,i}(L) \leq \left\lceil \frac{L + R_j - vol(\tau_j)/m}{T_j} \right\rceil \cdot vol(\tau_j)$.

Intuitively speaking, the term $vol(\tau_j)/m$ considers that the interfering workload can be uniformly distributed across the m threads to maximize the jitter. However, this may be not possible when the available concurrency varies over time. Nevertheless, note that upper-bounding the available concurrency with m still yields a valid upper-bound for the interference, as $\forall l \in [\bar{l}(\tau_j), m]$ it holds: $\left\lceil \frac{L + R_j - vol(\tau_j)/l}{T_j} \right\rceil \leq \left\lceil \frac{L + R_j - vol(\tau_j)/m}{T_j} \right\rceil$.

No other changes are required to the method proposed in [14] to analyze the limited-concurrency model. Finally, note that these adaptations may also work in the context of other response-time analysis approaches for parallel tasks.

4.2 Partitioned scheduling

Besides deadlocks, under partitioned scheduling, the presence of nodes of type BF can introduce additional delays whenever (i) a thread is suspended due to the execution of a node of type BF and (ii) other nodes are waiting in the work-queue of the same thread that cannot make progress because it is suspended. This phenomenon is denoted as *reduced-concurrency delay*. Existing analysis techniques for the classical DAG task model fail in accounting for this phenomenon and would hence produce an unsafe result. The objective of this section is to propose a partitioning algorithm that aims at avoiding—by construction—possible reduced-concurrency delays. This is accomplished by segregating each node $v_{i,f}$ of type BF in a different thread with respect to those used to serve the execution

Algorithm 1 Partitioning algorithm.

```
1:  procedure PARTITIONING
2:      for each $\tau_i \in \Gamma$ do
3:          $\forall v_{i,j} \in V_i, \mathcal{T}(v_{i,j}) \leftarrow \emptyset$
4:          for each $v_{i,j} \in V_i \ : \ x_{i,j} \neq \mathsf{BJ}$ do
5:              $\Phi_{BF} \leftarrow \{\phi_{i,k} : \exists v_{i,x} \in C(v_{i,j}) \cup \mathcal{F}'(v_{i,j}) \wedge \mathcal{T}(v_{i,x}) = \phi_{i,k}\}$
6:              if $\mathcal{T}(v_{i,j}) \neq \emptyset \wedge \mathcal{T}(v_{i,j}) \in \Phi_{BF}$ then
7:                  return FAILURE
8:              if $\mathcal{T}(v_{i,j}) = \emptyset \wedge |\Phi_{BF}| >= m$ then
9:                  return FAILURE
10:             if $\mathcal{T}(v_{i,j}) = \emptyset$ then
11:                 $\mathcal{T}(v_{i,j}) \leftarrow$ any $\phi_{i,x} \in \{\Phi_i \setminus \Phi_{BF}\}$
12:                 if $(x_{i,j} = \mathsf{BF})$ then
13:                     $\mathcal{T}(\mathcal{J}(v_{i,j})) \leftarrow \mathcal{T}(v_{i,j})$
14:             for each $v_{i,f} \in C(v_{i,j}) \cup \mathcal{F}'(v_{i,j}) : \mathcal{T}(v_{i,f}) = \emptyset$ do
15:                 $\Phi'_{BF} \leftarrow \{\phi_{i,k} \in \phi_i : \exists v_{i,x} \in C(v_{i,f}) \wedge \mathcal{T}(v_{i,x}) = \phi_{i,k}\}$
16:                 if $\Phi_i \setminus \{\Phi'_{BF} \cup \mathcal{T}(v_{i,j})\} = \emptyset$ then
17:                     return FAILURE
18:                 $\mathcal{T}(v_{i,f}) \leftarrow$ any $\Phi_i \setminus \{\Phi'_{BF} \cup \mathcal{T}(v_{i,j})\}$
19:                 $\mathcal{T}(\mathcal{J}(v_{i,f})) \leftarrow \mathcal{T}(v_{i,f})$
20:         return SUCCESS
```

of the nodes that may be waiting for the completion of $v_{i,f}$ (i.e., enqueued in a work-queue after $v_{i,f}$). Once a partitioning of this kind is obtained, then existing methods (e.g., [10]), can be used to analyze a parallel task under the limited-concurrency model.

It is worth observing that the same principle of Lemma 3 can be used to obtain such a partitioning. Indeed, by extending Equation (3) to account for nodes of types in {BC, BF, NB}, it is possible to identify the condition under which the allocation of such nodes cannot generate reduced-concurrency delays. The only difference with respect to Lemma 3 resides in a slight adaption to ensure consistency with the notation: $\mathcal{F}(v_{i,a})$ must be replaced with function $\mathcal{F}'(v_{i,a})$ defined as $\mathcal{F}'(v_{i,a}) = \mathcal{F}(v_{i,a})$, if $v_{i,a}$ is of type BC, and $\mathcal{F}'(v_{i,a}) = \emptyset$ otherwise. Also note that, since nodes of types BF and BJ are introduced to model parts of the same function (see Listing 1), their partitioning is forced to the same thread and hence there is no need to test them with Equation (3). These observations allow designing the partitioning algorithm reported in Algorithm 1.

The proposed approach incrementally assigns nodes to threads (and hence to processors). For each task τ_i, the algorithm iterates over all the nodes $v_{i,j}$ of the graph and computes the set Φ_{BF} of threads where nodes of type BF are allocated, since they may generate reduced-concurrency delay to $v_{i,j}$. As mentioned in Lemma 3, these threads either serve (i) nodes of type BF that may concurrently execute with $v_{i,j}$ (see Eq. (2)) or (ii) when $v_{i,j}$ is a node of type BC, the corresponding node of type BF.

If $v_{i,j}$ has already been allocated to a thread in Φ_{BF}, then it may suffer reduced-concurrency delay and the partitioning fails (line 7). If it has not been allocated but the nodes in Φ_{BF} are spread across all threads, then it is not possible to avoid reduced-concurrency delay and the partitioning fails (line 9). Otherwise, $v_{i,j}$ is allocated to one of the remaining threads (line 11). Whenever $v_{i,j}$ is of type BF, the algorithm also forces the allocation of its corresponding node of type BJ (denoted with $\mathcal{J}(v_{i,f})$ in the algorithm). Finally, the algorithm allocates the nodes of type BF that may generate reduced-concurrency delay to $v_{i,j}$ that have not yet been allocated. To avoid reduced-concurrency delay, each of such nodes $v_{i,f}$ is allocated to threads to which (i) $v_{i,j}$ is not allocated to, and (ii) the nodes that may in turn generate reduced-concurrency delay to $v_{i,f}$ (set Φ'_{BF} at line 15) are not allocated to. If no threads of this kind exist, the algorithm fails (line 17). The set $C(v_{i,j}) \cup \mathcal{F}'(v_{i,j})$ can be computed in $O(|V_i|^2)$ [16] hence the computational complexity required for applying the algorithm to a single task is $O(|V_i|^4)$.

5 EXPERIMENTS

This section presents an experimental study that has been conducted to assess the impact of reduced concurrency under global and partitioned scheduling.

DAG tasks have been generated with the technique reported in [14], which is not described here due to lack of space; please refer to [14] for the details. This task generator has been extended to be compliant with the model proposed in Section 2. In particular, each node $v_{i,j}$ has been associated with a type: whenever a fork-join subgraph is generated, it has an associated probability $p_{BF} = \frac{d}{d+1}$ to be delimited by nodes of type BF and BJ, where d denotes the *depth* (i.e., the degree of nesting in the fork-join graph, with higher numbers representing deeper nodes) of the node in the graph. Source and sink nodes are always assigned to type NB. Similarly to [14], the WCET of each node was randomly generated in the interval $[0, 100]$ with uniform distribution, $d = 2$, and task utilizations were generated with the UUnifast algorithm [3] by specifying a fixed number of tasks n and a target utilization $U = \sum_{\tau_i \in \Gamma} C_i/T_i$, where $C_i = \sum_{v_{i,j} \in V_i} C_{i,j}$. Periods are obtained as $T_i = C_i \cdot U_i$, and $D_i = T_i$ for all tasks.

Two types of experiments have been conducted. In the first one, the analysis in [14], which targets global fixed-priority scheduling of standard DAG-tasks, is compared against the one proposed in Section 4. In the second one, Algorithm 1 is compared against the case in which tasks are partitioned using the worst-fit heuristic (with respect to the utilization of each processor). When a node can be allocated in multiple threads according to Algorithm 1, one of them is chosen with the worst-fit heuristic. Once a partitioning was obtained, the analysis of Fonseca et al. [10] (in conjuction with the SPLIT analysis for self-suspending tasks [10]) was used to test the system schedulability. Clearly, when Algorithm 1 failed or the worst-fit heuristic failed, the task set was deemed unschedulable. Figure 2 shows six representative configurations for global and partitioned scheduling, reported in insets (a)-(c)-(e) and (b)-(d)-(f), respectively. The generation parameters are reported in the captions above the graphs. For each value in the graphs, 500 task sets were tested. In the experiments where the maximum available concurrency l_{max} have been varied, the generation enforced that the number of nodes of type BF of a task that may be concurrently executed is included in the interval $[b_{min}, b_{max}]$, thus allowing to explicitly control the reduction of concurrency in the generation. Tasks that violated this requirement or which are deemed not schedulable by to the schedulability test that does not consider the reduction of concurrency ([14] for global scheduling and [10] for partitioned) were discarded and re-generated. Note that, in this case, the lower bound to the available concurrency is always included in the interval $[l_{min}, l_{max}] = [m - b_{max}, m - b_{min}]$. When the other parameters are varied (i.e., m and n), no task is discarded. In Figure 2 (a) and (b) l_{max} has been varied for $m = 8$. Figure 2 (a) targets global scheduling and shows that the schedulability ratio (i.e., the percentage of schedulable task sets) starts decreasing abruptly for $l_{max} \geq 4$. Figure 2 (b) shows that partitioned scheduling exhibits a more graceful degradation of the schedulability as l_{max} decreases.

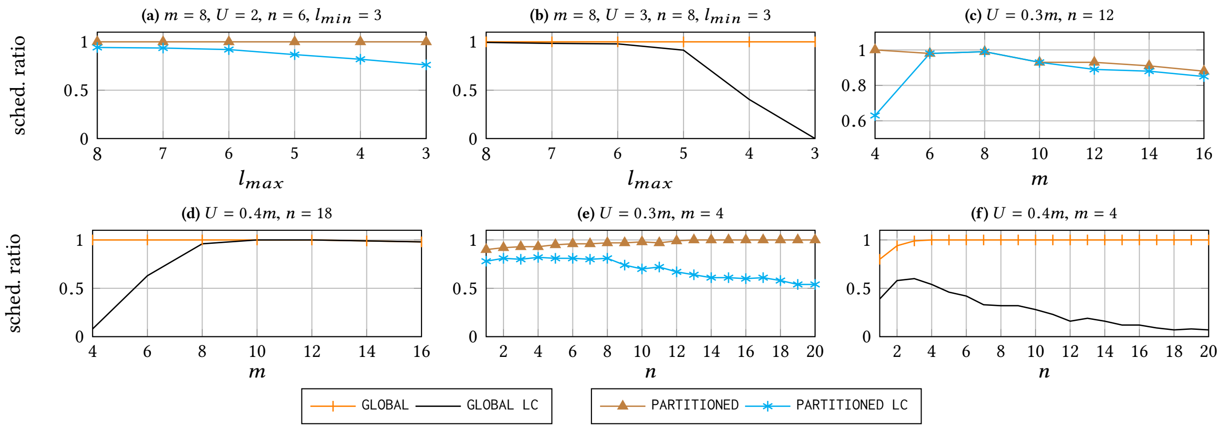

Figure 2: Schedulability ratio when l_{max}, m, and n are varied.

This is attributed to the fact that, while the analysis for global scheduling considers a constant reduction of concurrency for each task, Algorithm 1 considers the reduction of concurrency locally to each node. Figure 2 (c) and (d) show that the schedulability achieved in the presence of reduced concurrency is lower when the number of threads is low, whereas it is almost comparable with the original schedulability test for $m \geq 8$. Finally, Figure 2 (e) and (f) illustrate how schedulability decreases with the number of tasks: this is because with more tasks it is more likely to have some of them with a largely-reduced available concurrency. Overall, this experimental study highlights that new analysis methods (as those presented in this paper) are needed to analyze parallel tasks implemented with thread pools, as the optimism introduced by state-of-the-art analysis techniques is quite consistent.

6 RELATED WORK

The literature related to real-time analysis techniques for parallel tasks is quite vast: hence, only a selection of the related papers is discussed in the following. Most works adopted the DAG task model and targeted global scheduling [9, 14], federated scheduling [11, 13, 20], and partitioned scheduling [7, 10]. Other works targeted different task models: some examples are the fork-join [12], dataflow [15], and gang [8] task models. However, none of them focused on modeling implementations with thread pools and condition variables, nor addressed issues related to reduced concurrency.

The analysis of specific implementations of parallel tasks has been studied in some works, mainly targeting the OpenMP [2] framework. For instance, Serrano et al. [17] studied *tied* and *untied* sub-tasks in OpenMP, proposing a schedulability analysis for the case of a single DAG task composed of only untied sub-tasks. A tied sub-task represents a sub-graph whose nodes must all execute on a single thread, whereas untied sub-tasks have no additional constraints. Finally, Sun et al. [18] proposed an improved scheduling policy for OpenMP that improves schedulability for a single DAG task with tied sub-tasks.

7 CONCLUSIONS

This paper presented a novel scheduling model for parallel real-time tasks that allows modeling the case in which they are implemented with thread pools and condition variables. Techniques to verify the absence of deadlocks and analyze the task set schedulability have been proposed for both global and partitioned scheduling. An experimental study showed how the reduction of concurrency affects schedulability. Future work will be aimed at improving the proposed analysis techniques, e.g., by explicitly considering the variability of the available concurrency during tasks execution and by designing improved partitioning algorithms.

REFERENCES

[1] [n. d.]. Eigen Library. http://eigen.tuxfamily.org/index.php?title=Main_Page
[2] 2013. OpenMP Application Program Interface, Version 4.0.
[3] E. Bini and G.C. Buttazzo. 2005. Measuring the performance of schedulability tests. In *Real-Time Systems.*
[4] A. Biondi and Y. Sun. 2005. On the ineffectiveness of 1/m-based interference bounds in the analysis of global EDF and FIFO scheduling. In *Real-Time Systems.*
[5] B. Brandenburg and M. Gül. 2016. Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations. In *37th IEEE Real-Time Systems Symposium.*
[6] D. Casini, A. Biondi, and G. Buttazzo. 2017. Semi-Partitioned Scheduling of Dynamic Real-Time Workload: A Practical Approach Based On Analysis-driven Load Balancing. In *29th Euromicro Conference on Real-Time Systems.*
[7] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo. 2018. Partitioned Fixed-Priority Scheduling of Parallel Tasks Without Preemptions. In *2018 IEEE Real-Time Systems Symposium (RTSS).*
[8] Z. Dong and C. Liu. 2017. Analysis Techniques for Supporting Hard Real-Time Sporadic Gang Task Systems. In *IEEE Real-Time Systems Symposium.*
[9] J. Fonseca, G. Nelissen, and V. Nélis. 2017. Improved Response Time Analysis of Sporadic DAG Tasks for Global FP Scheduling. In *25th International Conference on Real-Time Networks and Systems (RTNS '17).*
[10] J. Fonseca, G. Nelissen, V. Nelis, and L. M. Pinho. 2016. Response time analysis of sporadic DAG tasks under partitioned scheduling. In *11th IEEE Symposium on Industrial Embedded Systems (SIES).*
[11] X. Jiang, N. Guan, X. Long, and W. Yi. 2017. Semi-Federated Scheduling of Parallel Real-Time Tasks on Multiprocessors. In *IEEE Real-Time Systems Symposium.*
[12] K. Lakshmanan, S. Kato, and R. Rajkumar. 2010. Scheduling Parallel Real-Time Tasks on Multi-core Processors. In *2010 31st IEEE Real-Time Systems Symposium.*
[13] J. Li, J. J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah. 2014. Analysis of Federated and Global Scheduling for Parallel Real-Time Tasks. In *26th Euromicro Conference on Real-Time Systems.*
[14] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. Buttazzo. 2017. Schedulability Analysis of Conditional Parallel Task Graphs in Multicore Systems. *IEEE Trans. Comput.* 66, 2 (Feb 2017), 339–353.
[15] H. Rihani, M. Moy, C. Maiza, R. I. Davis, and S. Altmeyer. 2016. Response Time Analysis of Synchronous Data Flow Programs on a Many-Core Processor. In *24th International Conference on Real-Time Networks and Systems.*
[16] R. Sedgewick. 1997. *Algorithms in C.*
[17] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quiñones. 2015. Timing characterization of OpenMP4 tasking model. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES).*
[18] J. Sun, N. Guan, Y. Wang, Q. He, and W. Yi. 2017. Real-Time Scheduling and Analysis of OpenMP Task Systems with Tied Tasks. In *38th IEEE Real-Time Systems Symposium.*
[19] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*
[20] N. Ueter, G. von der Brüggen, J. Chen, J. Li, and K. Agrawal. 2018. Reservation-Based Federated Scheduling for Parallel Real-Time Tasks. In *2018 IEEE Real-Time Systems Symposium (RTSS).*