Analysis of Federated Scheduling for Integer-Valued Workloads

Marion Sudvarg
Chris Gill

Marion Sudvarg and Chris Gill, “Analysis of Federated Scheduling for Integer-Valued Workloads,” in *Proc. of 30th International Conference on Real-Time Networks and Systems* (RTNS), June 2022, pp. 12-23.
DOI: 10.1145/3534879.3534892

Dept. of Computer Science and Engineering
McKelvey School of Engineering
Washington University in St. Louis

Analysis of Federated Scheduling for Integer-Valued Workloads

Marion Sudvarg
msudvarg@wustl.edu
Washington University in St. Louis
St. Louis, Missouri, USA

Chris Gill
cdgill@wustl.edu
Washington University in St. Louis
St. Louis, Missouri, USA

ABSTRACT

In federated scheduling of parallel real-time tasks on multiprocessor systems, high-utilization tasks are allocated dedicated processors on which they execute exclusively. Several methods exist for allocating a sufficient number of processors to guarantee that each task meets its deadline. In this paper, we propose two new strategies for allocating unit-speed cores to tasks with integer workload and deadline values. The first method can be performed in constant time for each high-utilization task, given the task's total workload, critical-path length, and deadline. The second method exploits the DAG structure of high-utilization tasks, providing a potentially better schedule in pseudo-polynomial time. We analyze and evaluate these new bounds in the context of existing techniques, and demonstrate that, in practice, they often allocate fewer processor cores. We also present a novel method for assigning an optimal number of dedicated cores to heavy tasks, describe how this method can be used in practice, and consider cases for which this is efficient.

KEYWORDS

parallel real-time systems, federated scheduling, integer-valued tasks, scheduling heuristics

ACM Reference Format:
Marion Sudvarg and Chris Gill. 2022. Analysis of Federated Scheduling for Integer-Valued Workloads. In *Proceedings of the 30th International Conference on Real-Time Networks and Systems (RTNS '22), June 7–8, 2022, Paris, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3534879.3534892

1 INTRODUCTION

The increasing computational demand of many real-time applications motivates scheduling strategies for task sets with *intra-task parallelism*. Systems hosting multiple tasks, where individual tasks may require the utilization of more than one processor to meet their deadlines, are becoming more common; such systems can be increasingly found in autonomous vehicles [23], computer vision systems [15], real-time hybrid testing environments [17], and even satellite telescopes [35]. Deciding how to schedule such tasks, i.e., how and when to allocate computational resources such that all execution completes before its corresponding deadline, is an important consideration on these systems.

Federated scheduling (which has been used in real-world applications such as real-time hybrid simulation [32]) proposes one approach to this problem [26]: high-utilization tasks (i.e., those with utilization $U \geq 1$) are each assigned a dedicated set of cores on which they alone execute; low-utilization tasks are then scheduled on the remaining cores using a multiprocessor scheduling algorithm for sequential tasks,[1] e.g., partitioned EDF [2], global EDF [1, 13, 25], or partitioned RM [11]. Previous approaches to federated scheduling have considered tasks for which workloads and deadlines are expressed as real numbers, forming a general and mathematically-sound basis for analysis. From a practical perspective, however, task workloads and deadlines often can be characterized by integer values; at a minimum CPU cycle counts provide a discrete base in the natural numbers, albeit at very fine granularity. This allows task execution to be decomposed into discrete unit time steps, which in some cases allows improved strategies for the assignment of dedicated processors.

In this paper, we consider the problem of efficiently assigning processors to high-utilization parallel tasks for which *deadlines* and *workloads* (including subtask workloads induced by each sequential component of the parallel program) are expressed as integer values; we refer to these as *integer-valued tasks*. This paper makes the following contributions:

- It presents a new constant-time assignment of cores to an integer-valued high-utilization task, which we prove to be (1) sufficient, (2) never worse than the original assignment presented in [26], and (3) well-defined for cases where the original assignment produced undefined values.
- It also proposes a pseudo-polynomial time heuristic algorithm for assigning cores to integer-valued high-utilization tasks, and demonstrates that this often assigns the minimum necessary number of cores to each task from a large set of randomly-generated parallel tasks.
- Finally, it formulates the problem of optimally assigning cores to high-utilization integer-valued tasks as an iterative subgraph ismorphism problem, which can be solved efficiently in many cases including those we tested.

The remainder of this paper is structured as follows: Section 2 provides background, and discusses related work; Section 3 introduces our system model for integer-valued tasks; Section 4 presents the new constant-time assignment of cores, and demonstrates its improvement over the method proposed in [26]; Section 5 discusses a pseudo-polynomial time application of list scheduling to integer-valued tasks, and proposes a heuristic algorithm; Section 6 presents the formulation of optimally assigning processors as a subgraph isomorphism problem, and discusses how to solve it; Section 7 evaluates the presented methods, considering both efficiency of assignment and efficiency of algorithmic execution time; and finally Section 8 concludes the paper, and discusses future work.

[1] A low-utilization parallel task can be scheduled as if it were a sequential task having the same total workload.

RTNS '22, June 7–8, 2022, Paris, France

ACM ISBN 978-1-4503-9650-9/22/06.
https://doi.org/10.1145/3534879.3534892

2 BACKGROUND AND RELATED WORK

Size, Weight, Power, and Cost (SWaP-C) constraints in real-time embedded systems drive the need to optimize the assignment of resources (like processor cores) to tasks, especially under federated scheduling [26] where those resources are fully partitioned among high-utilization parallel real-time tasks: efficient allocation of processors potentially allows more tasks to be admitted on systems with fewer cores. The *optimal* allocation of processors to a real-time parallel task is known to be $\mathcal{NP}$-Complete for arbitrary tasks on an arbitrary number of processors, including for tasks having only unit-workload subtasks [37]. However, a method for assigning *sufficient* processors to an implicit-deadline parallel task in constant time — with only knowledge of its total workload, deadline, and critical-path length (i.e., the makespan or *span* of its representative DAG) — was shown in [26]; in Section 4, we present an improvement of this method for integer-valued tasks. While the original federated scheduling model abstracted away overheads due to e.g., context switches or communication between subtasks, subsequent work has shown that in practice for a given platform and task set it is straightforward to measure such overheads and integrate them within system model parameters [27], even when widely varying costs (e.g., for randomization and work stealing) are involved [28].

In [3], federated scheduling is extended to constrained-deadline task systems. High-utilization tasks are assigned processors according to Graham's list scheduling algorithm [20]. This algorithm guarantees that when a processor becomes idle, available subtasks are selected for execution by each idle processor in an order defined by some priority list. Subtask priorities may be assigned, for example, according to the Critical Path rule (where a subtask's priority is assigned according to its critical-path length) or the Largest Number of Successors rule (where priority is assigned according to the total workload of the subgraph rooted at the subtask). The remaining low-utilization tasks are partitioned among remaining processors according to the method described in [7], then tasks on each processor are scheduled according to EDF. This was further extended to arbitrary-deadline task systems in [4], which again uses Graham's list scheduling to assign processors to high-utilization tasks, and partitions low-utilization tasks according to the methods described in [8]. In [5], the conditional sporadic DAG tasks model is considered for programs that contain conditional branching logic; in this case, sufficient processors are assigned according to the deadline, worst-case execution time, and critical-path length across all conditional paths.

Under Graham's list scheduling, subtasks are scheduled non-preemptively: once a subtask is selected for execution on a processor, it runs to completion on that same processor. In [14], an alternative algorithm is proposed for assigning processors to high-utilization tasks, which exploits each task's DAG structure to construct a schedule where subtasks may be preempted. Subtasks are prioritized according to the Largest Number of Successors, but a subtask is preempted if (1) after executing for some duration, the remaining workload of its rooted subgraph is exceeded by that of another available subtask; or (2) if an idle subtask becomes urgent, i.e., its critical-path length equals the remaining time until deadline, and therefore it must be scheduled immediately. The presented algorithm has pseudo-polynomial time complexity and, when evaluated in comparison to previous federated scheduling approaches, was shown often to produce less resource waste.

Both the Critical Path and Largest Number of Successors rules are known to be optimal when a task's DAG forms an in-tree or an out-tree [21, 34]. Polynomial time algorithms are known for constructing optimal schedules on two-processor systems [12, 18, 19, 31]. Further, interval-ordered parallel tasks[2] for which each subtask has unit execution time are known to be optimally schedulable in linear time [33]. Nonetheless, since optimally assigning dedicated cores to high-utilization parallel tasks is $\mathcal{NP}$-Complete in general, the heuristics above define the state of the art. In Section 5, we present a heuristic algorithm for assigning cores to integer-valued tasks that, despite being polynomial in the task's workload, nonetheless often finds an optimal core assignment for the domain of task sets evaluated in Section 7.

Other approaches relax the federated scheduling requirement of assigning completely dedicated processors to each high-utilization task. In [22], it is observed that the original assignment rule for federated scheduling in [26] rounds up any fractional part of the value; the authors propose semi-federated scheduling, which provides a framework for algorithms that instead schedule the fractional parts from high-utilization tasks on shared processors with low-utilization tasks. Under reservation-based federated scheduling [36], tasks are assigned dedicated reservation servers (rather than dedicated processors), and rules are provided for scheduling these servers on shared processors. This technique is especially useful for constrained-deadline tasks that leave processors idle between a task instance's deadline and its next release, and the paper extends analysis of the technique to arbitrary-deadline tasks. We defer as future work the application of our proposed assignment techniques to these approaches.

3 SYSTEM MODEL

In this paper, we consider a system Γ of n independent sporadic integer-valued parallel real-time tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task τ_i represents a sequence of jobs and is characterized by a *workload* $C_i \in \mathbb{N}$, representing the worst-case execution time of each of its jobs; and by a *period* $T_i \in \mathbb{N}$, representing the minimum inter-arrival time between consecutive jobs. Tasks are also characterized by relative *deadlines* $D_i \in \mathbb{N}$: a job of task τ_i arriving at time $t \in \mathbb{N}_0$ must complete execution no later than the time $t + D_i$; this describes the absolute deadline of that job. For this paper, we consider *constrained-deadline* tasks: every job must complete by the activation time of the next job of the same task, i.e. $D_i \leq T_i$. In several places throughout this paper, where we limit our consideration to implicit-deadline tasks, for which $D_i = T_i$, we explicitly state that we are considering implicit-deadline tasks; otherwise, the reader can assume that the tasks have constrained deadlines. We consider a task system to be *schedulable* on m identical cores if there exists a preemptive scheduler that can assign task execution to cores such that all jobs are guaranteed to complete before their deadlines.

[2] A parallel task is interval-ordered iff the incomparability graph of the partial-ordering of precedence constraints over its subtasks is chordal.

Each task τ_i consists of a set of subtasks $\tau_{i,j}$ with a precedence relation $\prec$ over them. Each individual subtask $\tau_{i,j}$ is characterized by a workload $c_{i,j} \in \mathbb{N}$, representing its worst-case execution time. An individual subtask must execute sequentially, i.e. it is characterized by a sequence of instructions that must be completed in order, and take up to $c_{i,j}$ time to complete. We assume that subtask execution is reentrent; execution may be preempted by another subtask, and it need not resume executing on the same core in the system. The precedence relation constrains subtask execution such that if $\tau_{i,a} \prec \tau_{i,b}$, then $\tau_{i,a}$ must fully complete its execution before $\tau_{i,b}$ is scheduled. We say that a task $\tau_{i,j}$ becomes *available* when all tasks $\tau_{i,a}$ for which $\tau_{i,a} \prec \tau_{i,j}$ have completed execution. As in [26], we characterize tasks as either *light* or *heavy*: heavy tasks are those for which $C_i \geq D_i$; tasks for which $C_i > D_i$ must have their potential parallelism exploited to be schedulable. The focus of this paper is *federated scheduling*, for which each heavy task is assigned a dedicated set of cores on which it alone executes. We consider only tasks for which the deadline D_i and the value of each subtask workload $c_{i,j}$ are positive integers; we refer to these as *integer-valued tasks*.

The partial-ordering of precedence over subtask execution that describes task execution can be characterized as a *directed acyclic graph (DAG)*. We say, for each parallel task τ_i, that there exists a DAG G_i with a collection of vertices $v_{i,j}$ corresponding to subtasks $\tau_{i,j}$. Each vertex is assigned as an attribute the workload $c_{i,j}$ of the corresponding subtask. A directed edge from vertex $v_{i,a}$ to $v_{i,b}$ exists if and only if $\tau_{i,a} \prec \tau_{i,b}$ and there is no $\tau_{i,c}$ for which $\tau_{i,a} \prec \tau_{i,c} \prec \tau_{i,b}$, i.e., $\tau_{i,b}$ directly succeeds $\tau_{i,a}$.

For each graph vertex $v_{i,j}$, we characterize its critical-path length, or *span*, $l_{i,j} \in \mathbb{N}$ as the longest path originating at that vertex, and weighted by the execution time of each vertex along the path (including $c_{i,j}$, the weight of the originating vertex). We characterize, for the corresponding task τ_i, the span $L_i \in \mathbb{N}$, which is the greatest span among all vertices, i.e. the critical-path length of the entire DAG. The span corresponds to the earliest completion time of a job in the task, relative to its activation time, if given an infinite number of cores on which to execute. It is clear that for a task to be schedulable, $L_i \leq D_i$.

The system model presented in this section abstracts away context switches, costs of communication between subtasks, and other overheads that are likely to be relevant in practice. However, as we noted in Section 2, prior work has shown that for a given platform and task set it is reasonably straightforward to measure and integrate them within the parameters of the system model.

4 FEDERATED SCHEDULING FOR INTEGER-VALUED WORKLOADS

In [26], it is shown that a heavy task τ_i characterized by workload C_i, span L_i, and deadline D_i is schedulable by any work-conserving (i.e., greedy) scheduler given sufficient cores:

$$n_i = \lceil \frac{C_i - L_i}{D_i - L_i} \rceil \tag{1}$$

We show here that a parallel task, to which integer values are assigned for its deadline and all subtask workloads, is schedulable on $n'_i = \lceil \frac{C_i - L_i + 1}{D_i - L_i + 1} \rceil$ identical cores, which provides practical and intuitive benefits over Equation 1 for bounding the number of cores assigned to heavy tasks.

4.1 A Proof of the Assignment

We begin with two definitions and reiterate two lemmas proven in [25].

Definition 4.1. Summarized from [26]: Assume that a machine's execution time is divided into discrete quanta called *steps*. A time step during which any processor is idle is an **incomplete step**.

Definition 4.2. Summarized from [25]: For a partially executed instance of a task, the **remaining critical path length** is the length of the longest path in the unexecuted portion of the DAG (including partially executed nodes). In other words, it is the critical path length (as described in Section 3) of the sub-DAG describing the remaining work.

Lemma 4.3. *From Lemma 2 in [26]: Consider a greedy scheduler running on n_i processors for t time steps. If the total number of incomplete steps during this period is t^*, the total work F^t done during these time steps is at least:*

$$F^t \geq n_i t - (n_i - 1)t^*$$

Lemma 4.4. *From Lemma 3 in [26]: If a job of task τ_i is executed by a greedy scheduler, then every incomplete step reduces the remaining critical-path length of the job by 1.*

Now, we state our theorem, with a proof that closely follows the proof of Theorem 2 in [26]:

Theorem 4.5. *If a parallel task τ_i having integer-valued subtask workloads and deadline is assigned*

$$n'_i = \lceil \frac{C_i - L_i + 1}{D_i - L_i + 1} \rceil \tag{2}$$

dedicated processors, then all its jobs can meet their deadlines, when using a greedy scheduler.

Proof. Assume some job of heavy task τ_i misses its deadline when scheduled on $n'_i = \lceil \frac{C_i - L_i + 1}{D_i - L_i + 1} \rceil$ cores by a greedy scheduler. Then, over the D_i time steps between this job's release and its deadline, there are fewer than L_i incomplete steps; otherwise, by Lemma 4.4, the job would have completed, i.e., $t^* < L_i$.

Because τ_i has integer values for all subtask workloads $c_{i,j}$ and deadline D_i, it also has an integer-valued span L_i. So, we can say that each time step is of unit time, and so t^* is an integer. This implies that if $t^* < L_i$, then $t^* \leq L_i - 1$. Then, from Lemma 4.3:

$$F^t \geq n'_i t - (n'_i - 1)t^*$$

and since $t^* \leq L_1 - 1$ and $t = D_i$,

$$\begin{aligned} F^t &\geq n'_i D_i - (n'_i - 1)(L_i - 1) \\ &= n'_i (D_i - L_i + 1) + L_i - 1 \\ &= \lceil \frac{C_i - L_i + 1}{D_i - L_i + 1} \rceil (D_i - L_i + 1) + L_i - 1 \\ &\geq \frac{C_i - L_i + 1}{D_i - L_i + 1} (D_i - L_i + 1) + L_i - 1 \\ &= C_i - L_i + 1 + L_i - 1 = C_i \end{aligned}$$

Since the job has work of at most C_i, it must have finished in D_i steps, leading to a contradiction. □

4.2 The Improvement

We now show that the proposed assignment of cores in Theorem 4.5 is an improvement over Equation 1. First, we prove that the new assignment is never more pessimistic than the original, i.e. that $n' \leq n$ for heavy tasks that are schedulable under both methods:

Lemma 4.6. *$\lceil \frac{C_i - L_i + 1}{D_i - L_i + 1} \rceil \leq \lceil \frac{C_i - L_i}{D_i - L_i} \rceil$ for $L_i < D_i$*

Proof. □

Case 1. *Consider a heavy task τ_i; because $C_i \geq D_i$ and $L_i < D_i$:*

$$D_i - L_i \leq C_i - L_i \implies 1 + \frac{1}{C_i - L_i} \leq 1 + \frac{1}{D_i - L_i} \implies$$
$$\frac{C_i - L_i + 1}{C_i - L_i} \leq \frac{D_i - L_i + 1}{D_i - L_i} \implies \frac{C_i - L_i + 1}{D_i - L_i + 1} \leq \frac{C_i - L_i}{D_i - L_i} \implies$$
$$\lceil \frac{C_i - L_i + 1}{D_i - L_i + 1} \rceil \leq \lceil \frac{C_i - L_i}{D_i - L_i} \rceil$$

Case 2. *Now, consider a task for which $C_i \leq D_i$. Because all subtask workloads are integer values, this implies that $L_i \geq 1$:*

$$C_i - L_i + 1 \leq D_i - L_i + 1 \text{ and } C_i - L_i \leq D_i - L_i$$

So:

$$\lceil \frac{C_i - L_i + 1}{D_i - L_i + 1} \rceil = \lceil \frac{C_i - L_i}{D_i - L_i} \rceil = 1$$

Since $n' \leq n$, we know that Equation 1 never assigns more cores to a process than the original in [26].

We now present two observations, which demonstrate the practical and intuitive benefit of the new assignment.

Observation 1. *If a task τ_i has $L_i = D_i$, there should be some number of cores on which the task is schedulable.*

This can be stated, equivalently, that if a parallel task has a span equal to its deadline, the task is still schedulable given sufficiently many cores. Indeed, consider the specific case of a sequential task. A sequential task having workload C_i and utilization $U_i = 1$ is schedulable on a single, dedicated core. In this case, since $U_i = 1$, we have $D_i = C_i$, and as a sequential task, $L_i = C_i$. Then Equation 1 reduces to $n = \frac{0}{0}$ which is undefined. The new assignment of Equation 2 is $n' = \frac{1}{1}$, which is the correct result. In general, Equation 1 gives an undefined result for any task having $L_i = D_i$, since it reduces to a fraction with 0 as denominator. Under Equation 2, the fraction has 1 as the denominator, providing a well-defined result. For $L_i > D_i$, a task is not schedulable; for integer-valued deadline and subtask workloads, this reduces to $L_i \geq D_i + 1$, in which case n is undefined (having a 0 denominator) or the ceiling function of a negative fraction.

Observation 2. *An implicit-deadline task τ_i with integer-valued deadline and subtask workloads, having $L_i = 1$, can be scheduled on a number of cores n_i equal to the ceiling of its utilization, $\lceil U_i \rceil$.*

In this case, the task is composed entirely of unit-time subtasks, with no dependencies among them. It is fully parallelizable: under a work-conserving scheduler, there will be no idle cores at any time step (except the last one, if there are fewer remaining subtasks than cores). The new allocation correctly assigns cores:

$$n'_i = \lceil \frac{C_i - L_i + 1}{D_i - L_i + 1} \rceil = \lceil \frac{C_i}{D_i} \rceil = \lceil U_i \rceil$$

4.3 Capacity Augmentation Bound

The concept of a *capacity augmentation bound* was defined in [25], which, as a way to describe the performance bound of a real-time scheduler, provides an alternative to the resource augmentation bound or utilization bound. We restate the definition here:

Definition 4.7. Given a task set Γ, having a total utilization U, a scheduling algorithm $\mathcal{S}$ with **capacity augmentation bound** b can always schedule this task set on m identical unit-speed processors as long as the following conditions are satisfied:

(1) $U \leq m/b$
(2) For each task $\tau_i \in \Gamma$, $L_i \leq D_i/b$

We show, for an *implicit deadline* task set Γ, that the capacity augmentation bound of a greedy scheduler over a set of integer-valued heavy tasks is at most 2. We begin by proving the following lemma, which makes a statement similar to Lemma 4 in [26]:

Lemma 4.8. *The capacity augmentation bound of a greedy scheduler for a single implicit-deadline heavy task τ_i with integer deadline and subtask workloads assigned $n'_i = \lceil \frac{C_i - L_i + 1}{D_i - L_i + 1} \rceil$ processors is at most 2.*

Proof. Lemma 4 of [26] proves that for an implicit-deadline heavy task τ_i with integer deadline and subtask workloads, if $L_i \leq D_i/2$, then:

$$n = \lceil \frac{C_i - L_i}{D_i - L_i} \rceil \leq 2 * U_i$$

Since we know

$$n' = \lceil \frac{C_i - L_i + 1}{D_i - L_i + 1} \rceil \leq n$$

it follows that $n' \leq 2 * U_i$ □

Then, it follows that:

Corollary 4.9. *For an implicit deadline task system Γ for which there are $m \geq 2 \sum_i U_i$ processors, and for which all heavy tasks τ_i have $L_i \leq D_i/2$, the number of processors available for light tasks is at least $n_{light} \geq 2 \sum_{light} U_i$.*

This allows us to prove an upper-bound for the capacity augmentation bound of federated scheduling for integer-valued tasks:

Lemma 4.10. *The federated scheduling algorithm, which assigns*

$$n'_i = \lceil \frac{C_i - L_i + 1}{D_i - L_i + 1} \rceil$$

dedicated processors to each heavy task τ_i, has a capacity augmentation bound of at most 2.

Proof. For integer-valued task sets satisfying conditions (1) and (2) of Definition 4.7, we have shown that the total utilization of light tasks (those with $U_i < 1$) is less than $n_{light}/2$, and so these tasks are schedulable by any algorithm that provides a utilization bound of 2. Since they can be scheduled by partitioned EDF or partitioned RM [7, 11], the capacity augmentation bound holds. □

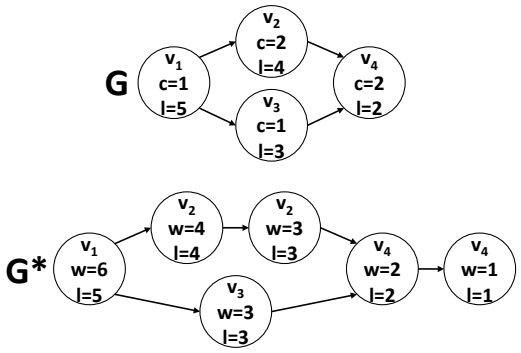

Figure 1: Decomposition of a task DAG G (top) into the DAG G^* of unit-workload vertices (bottom): c denotes subtask workload, l is span, and w is subgraph work.

5 UNIT-WORKLOAD LIST SCHEDULING

The method presented in Section 4 for federated scheduling of high-utilization integer-valued tasks assigns sufficient processors to guarantee feasibility of any work-conserving scheduler, given only each task's workload, critical-path length, and deadline. Such constant-time assignment risks over-provisioning processors to each heavy task, leading to resource waste. In [3], Graham's list scheduling [20] is applied to heavy tasks; various heuristics for prioritizing subtasks may produce feasible schedules on fewer processors. List scheduling assigns available subtasks to idle processors in a non-preemptive fashion; as noted in [14], this may lead to over-allocation of processors to heavy tasks. However, allowing the idle subtask to preempt may induce a switching problem.[3]

To address this problem, we propose list scheduling of unit-workload subtasks. An integer-valued parallel task τ_i, represented by a DAG G_i, can be decomposed into a DAG G_i^* consisting of unit-workload vertices as illustrated in Figure 1: a vertex $v_{i,j}$ in G_i with workload c is mapped to a totally-ordered sequence of c vertices in G_i^* with edges forming a path from the first to the last. Any edges into $v_{i,j}$ would, in G_i^*, connect to the first vertex in the decomposition of $v_{i,j}$. Any edges out of v_j in G_i would now come from the last vertex in its decomposition. For such a DAG, list scheduling assigns a priority to each unit of work; this enables corresponding subtasks in the original DAG G_i to be preempted at unit time step boundaries.

5.1 Common List Scheduling Heuristics

The *Critical Path rule* (CP) for list scheduling selects available subtasks for execution in order of the greatest span. The assignment of spans to subtasks can be performed in a depth-first search fashion; as each vertex's span is assigned, the total span L of the graph is updated. This can be performed in time $O(|V| + |E|)$.

The subsequent decomposition of the DAG G to the unit-workload DAG G^*, which we denote as the function **`Convert_Unit_DAG`**, proceeds by first initializing G^* as a copy of G, then establishes the set V^* as a subset of G^* consisting of the vertices that are in G^* when it is initialized. In real code, V^* may be realized as a collection of pointers to vertices in G^*. The procedure then iterates through vertices in (or pointed to by elements of) V^*. For each vertex v_i having corresponding subtask workload c_i, it creates c_i vertices with unit subtask workload. These vertices are arranged in a sequential total order, with an edge connecting each one to the next; the first one is assigned a subtask span equal to that of v_i (i.e., l_i), and each subsequent vertex is assigned a subtask span one less than the previous. For all vertices v_j in E_i^{in}, a new edge is created from v_j to the first vertex in the sequence. For all vertices v_j in E_i^{out}, a new edge is created from the last vertex in the sequence to v_j. The original vertex v_i is then deleted. Once all vertices in V^* (i.e., those that were originally in G) have been decomposed and removed, the resulting DAG G^* now contains vertices having only unit workloads. Each existing vertex in G must be decomposed, and the decomposition produces an additional $C-|V|$ vertices. Additionally, each existing edge in G must be assigned to a new vertex in G^* and $C-|V|$ additional edges are created. So, the total running time of the algorithm can be expressed as $O(|V|+C-|V|+|E|+C-|V|)$; since $C \geq |V|$ we can simplify this to $O(C+|E|)$. List scheduling on G^* can then proceed in time polynomial in C and $|E|$.

The *Largest Number of Successors rule* (LNS) for list scheduling selects available subtasks for execution in order of the greatest total workload of the subgraph rooted at the corresponding vertex (referred to as *subgraph work* in [14]). Assignment of subgraph work to a vertex $v_i \in G$ can be accomplished by summing over the subtask workloads c_j assigned to each vertex v_j that can be reached by a path from v_i. A dynamic-programming approach can be implemented over the entire graph to assign subgraph work to all vertices in time $O(|V| + |E|)$. LNS can thus also be applied in pseudo-polynomial time for list scheduling of unit-workload subtasks.

Note that the decomposition of an integer-valued task DAG G to a unit-workload DAG G^* will need to additionally assign a subgraph work to each unit-workload vertex. This is accomplished similarly to the assignment of span, and does not affect the algorithm's execution time complexity.

5.2 Combined Heuristics

We propose two new pseudo-polynomial time heuristics that combine the CP and LNS rules for unit-workload list scheduling. As we demonstrate in Section 7.2, these often produce optimal processor assignments to high-utilization, integer-valued tasks.

5.2.1 The Heuristics. The first proposed heuristic, ***CP+LNS***, prioritizes subtasks according to greatest critical path first. Among available subtasks having equal spans, it prioritizes according to the greatest subgraph work. The algorithm for list scheduling of a unit-workload parallel task on n processors using **CP+LNS** maintains a priority queue of vertices, **`avail`**, sorted by descending span, then by descending subgraph work (this may be realized, e.g., by a max-heap). This is initialized by iterating over all vertices of G^*, and inserting those for which there are no incoming edges. It also initializes an empty candidate schedule $\mathcal{S}$.

For each unit time step before the deadline, it pops up to n vertices from the head of **`avail`** (or as many as **`avail`** contains if $\leq n$). Each of these vertices is assigned to a processor at that time step in $\mathcal{S}$. Each of these vertices then has its outgoing edges

[3]For example, consider least-laxity first scheduling: as the preempting subtask executes, its span immediately decreases, triggering a preemption by the subtask it just preempted, ad infinitum.

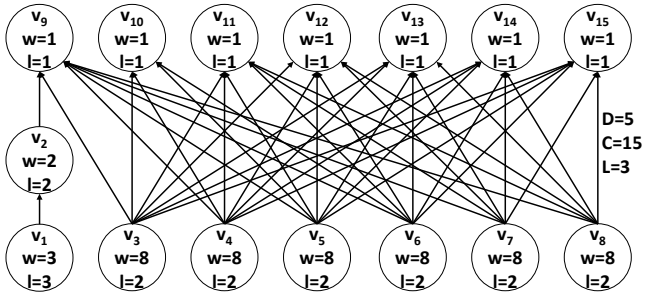

Figure 2: A parallel task where *LNS+CP* is optimal. Vertices $v_3 - v_8$ are each connected by an edge to each of $v_9 - v_{15}$.

removed, and as each edge is removed, the vertex pointed to by that edge is checked for remaining input edges. Any vertices that no longer have input edges are inserted into **avail**. If at any time step, a vertex popped from **avail** has a greater span than the remaining time before the deadline, **CP+LNS** will fail to produce a feasible schedule, so it returns **infeasible**. If, after any time step, **avail** is empty, then $\mathcal{S}$ is feasible as currently constructed (as discussed in Lemma 5.5). After D time steps, if **avail** is not empty, then there remain subtasks to be scheduled after the deadline, so the algorithm returns **infeasible**.

Because each vertex must be inserted once into **avail**, and each edge must be traversed once, the running time is $O(|V^*| \log |V^*| + |E^*|)$, where V^* and E^* are the respective sets of vertices and edges in G^*. Since for each vertex $v_i \in G$, the decomposition procedure adds $c_i - 1$ edges and vertices, $|V^*| = C$ and $|E^*| = |E| + C - |V|$. So the running time can be expressed in terms of the original graph G as $O(C \log C + |E| + C - |V|)$, or, more simply, as $O(C \log C + |E|)$.

Our second proposed heuristic, ***LNS+CP***, prioritizes subtasks according to greatest subgraph work first, and among available subtasks having equal subgraph work, it prioritizes according to the greatest critical path. It proceeds similarly to the **CP+LNS** algorithm, but **avail** is instead sorted first by subgraph work, then by span. Additionally, at each time step, it attempts to schedule any urgent subtasks, i.e., those having a span equal to the remaining time until the deadline (similarly to the heuristic in [14]). To do so, the algorithm first traverses **avail** to find any such subtasks. If there are more such subtasks than available processors, or if any subtask in **avail** has a span that exceeds the time until deadline, the schedule is infeasible. If there are processors remaining to be assigned at that time step, subtasks are scheduled from the head of **avail**. Like the **CP+LNS** heuristic, the **LNS+CP** algorithm must insert each vertex once into **avail** and traverse each edge once. However, it must additionally traverse **avail** at each time step. Since **avail** can have up to C vertices, this may incur an extra $O(C * D)$ time; this brings the running time of **LNS+CP** to $O(C \log C + C * D + |E|)$.

5.2.2 A List Scheduling Algorithm. Since algorithms that are polynomial in task workload exist to implement both ***LNS+CP*** and ***CP+LNS***, neither is optimal for integer-valued tasks in general. Further, neither can be expected to always assign fewer processors than the other to all integer-valued tasks.

Example 5.1. Consider a task with deadline $D = 5$ and the DAG structure shown in Figure 2 (note that edges connect each vertex in $\{v_3, \ldots, v_8\}$ to each vertex in $\{v_9, \ldots, v_{15}\}$). ***LNS+CP*** can provide a feasible schedule on 3 cores. ***CP+LNS***, however, requires 4 cores. On 3 cores, it will first schedule v_1, then any two of $\{v_3, \ldots, v_8\}$. At the second time step, it will schedule another three vertices from $\{v_3, \ldots, v_8\}$. At the third time step, there are only two available vertices (v_2 and the remaining vertex from $\{v_3, \ldots, v_8\}$). Since all processors must execute at each time step for the deadline to be met, a feasible schedule will not be constructed.

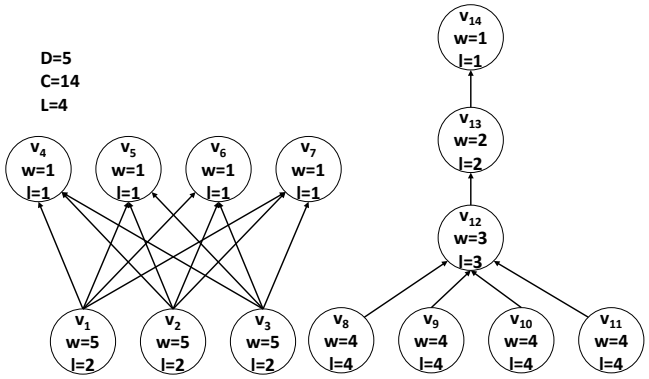

Figure 3: A parallel task where *CP+LNS* is optimal.

Algorithm 1: Integer_List_Schedule(G,D)

Input: Integer-valued task DAG G and deadline D
Output: A feasible schedule $\mathcal{S}$ for a number of processors n

Initial Setup
  $L, G \leftarrow$ `Calc_Spans`(G)
  $G \leftarrow$ `Calc_Subgraph_Work`(G)
  $G^* \leftarrow$ `Convert_Unit_DAG`(G)
  $n \leftarrow \lceil \frac{C_i}{D_i} \rceil$
  $n_{max} \leftarrow \lceil \frac{C_i - L_i + 1}{D_i - L_i + 1} \rceil$
if $n == n_{max}$ **then return greedy**, n ;
while $n < n_{max}$ **do**
  if `CP_LNS`(G^*,D,n) != **infeasible then return** $\mathcal{S}$, n;
  if `LNS_CP`(G^*,D,n) != **infeasible then return** $\mathcal{S}$, n;
  $n \leftarrow n + 1$
end
return greedy, n

Example 5.2. Consider a task with deadline $D = 5$ and the DAG structure in Figure 3. ***CP+LNS*** can provide a feasible schedule on 3 cores. ***LNS+CP***, however, requires 4 cores. On 3 cores, it will first schedule $\{v_1, v_2, v_3\}$, as no vertices have a span of 5 or greater. At the next time step, it will schedule $\{v_8, v_9, v_{10}\}$; v_{11} must also be scheduled at this time, as it has a span of 4, but there are no available processors remaining.

Because neither heuristic is guaranteed to assign fewer processors than the other, and because both execute in pseudo-polynomial time, we propose to assign processors to a heavy integer-valued task according to Algorithm 1. This first computes the span and subgraph work of each subtask, then computes the minimum number of processors n_{min} that could feasibly schedule the task (the ceiling of the task utilization) and the maximum number of processors n_{max} needed according to Theorem 4.5. If the two are equal, it

returns that value; otherwise, it decomposes the DAG to the unit-workload DAG G^*. It then attempts to construct a schedule using ***CP+LNS*** then ***LNS+CP*** for n_{min} processors,[4] returning if either is feasible. Otherwise, it continues to call them both in an iterative fashion, increasing n by one for each iteration. It stops when n reaches n_{max}; at this point, any greedy algorithm will successfully schedule the task.

5.3 Correctness of Algorithm 1

Lemma 5.3. *The function* **Convert_Unit_DAG** *is surjective over the set $\mathbb{G}$ of finite parallel task DAGs with integer-value subtask workloads onto the set $\mathbb{G}^*$ of finite parallel task DAGs with unit-value subtask workloads.*

Proof. This follows from the function **Convert_Unit_DAG**, which for any $G \in \mathbb{G}$, will produce a DAG $G^* \in \mathbb{G}^*$. Further, $\mathbb{G}^* \subset \mathbb{G}$, and for any $G^* \in \mathbb{G}^*$, **Convert_Unit_DAG**$(G^*) = G^*$. Therefore the function is surjective. □

We also present the following lemma, which states that if a parallel task DAG G with deadline D is decomposed to a DAG G^* by the application of the **Convert_Unit_DAG** function, and G^* is schedulable, then G^* is also schedulable by a preemptive scheduler (i.e., one for which subtask execution can be preempted).

Lemma 5.4. *Given a DAG G and deadline D, if there exists a feasible schedule $\mathcal{S}^*$ for the DAG G^* =* **Convert_Unit_DAG**(G)*, then there exists a function $f : \mathcal{S}^* \rightarrow \mathcal{S}$ where $\mathcal{S}$ describes a feasible schedule for G.*

Proof. Consider a modified algorithm for the **Convert_Unit_DAG** function that adds, to each unit workload vertex in the resulting graph G^*, the index of the corresponding vertex in G from which it was generated (like in Figure 1). Then, for the schedule $\mathcal{S}^*$, create a schedule $\mathcal{S}$ by replacing replace all vertices v^* with the original vertex $v \in G$.

If $\mathcal{S}^*$ is a feasible schedule, then there will be no v^* in $\mathcal{S}^*$ that executes after the deadline; similarly, then, in $\mathcal{S}$ all execution will occur before the deadline. Since for each $v_i \in G$ having a corresponding subtask workload of c_i, there are c_i corresponding vertices in G^*, when these are converted back to vertices in G, each vertex in G will be assigned to execute in c_i slots in the schedule $\mathcal{S}$. Further, each slot will be at a different time step: two vertices v^* generated from the same v_i cannot both be in **avail** at the same time step, since vertices are added to **avail** only when they no longer have incoming edges, and all vertices generated from the same v_i form a totally-ordered sequence with connecting edges.

Additionally, if $v_i \prec v_j$, then v_i it will occupy only slots at time steps before the slots occupied by v_j: there is a directed path from the last unit vertex generated from v_i to the first unit vertex generated from v_j in G^*, which means that v_j (and its successors) will not be moved into **avail** until v_j (and its predecessors) have been scheduled. Thus, since each $v_i \in G$ will be scheduled in $\mathcal{S}$ for c_i unique time-steps before the deadline, and all precedence constraints are respected, $\mathcal{S}$ is feasible. □

[4]Trying ***CP+LNS*** before ***LNS+CP*** is motivated by results shown in Section 7.2, which suggest that ***CP+LNS*** assigns fewer processors, on average, for the task DAGs tested.

Algorithm 2: Federated_Scheduling(Γ,m)

Input: A set Γ of integer-valued tasks τ_i with DAGs G_i and deadlines D_i; m processors
Output: Feasible schedules $\mathcal{S}_i$ for each task

```
Init
    m_r ← m ; ▷ Remaining processors
forall Heavy tasks τ_i ∈ Γ do
    n_i, S_i = Integer_List_Schedule (G_i,D_i)
    m_r ← m_r − n_i
    if m_r < 0 then return infeasible ;
end
if Can schedule light tasks on m_r processors then return S ;
else return infeasible ;
```

Lemma 5.5. *The* **Integer_List_Schedule** *algorithm always produces a feasible schedule for G^*.*

Proof. For any given value of n, the **CP_LNS** and **LNS_CP** algorithms schedule vertices in a greedy fashion; therefore, by Theorem 4.5 both are guaranteed to produce a feasible schedule for the value n_{max} given by Equation 2. **Integer_List_Schedule** attempts both algorithms for values of n up to $n_{max} - 1$, after which any greedy scheduler will produce a feasible schedule.

It remains to prove that any schedule produced for $n < n_{max}$ is feasible. Both **CP_LNS** and **LNS_CP** produce a schedule iff **avail** is empty after any time step before D. Assume by contradiction that **avail** is empty, but the schedule $\mathcal{S}$ produced is not feasible. The schedule $\mathcal{S}$ will have respected all precedence constraints among vertices: if a vertex is placed in **avail**, it has no remaining input edges, so all of its predecessors have already been scheduled; therefore, a vertex can only be scheduled after its predecessors. Since a schedule has been produced for up to D time steps, no vertex in $\mathcal{S}$ will be scheduled after D. So, for $\mathcal{S}$ to not be a feasible schedule for G, it must be the case that there is a subset V of vertices in G not in $\mathcal{S}$. But, since G (and therefore V) is finite, and G is acyclic, there must be some vertex $v \in V$ that has no input edge. But, if this were the case, it would be in **avail**. Therefore, if **avail** is empty after any time step before D, the produced schedule $\mathcal{S}$ is feasible. □

The theorem follows:

Theorem 5.6. *The* **Integer_List_Schedule** *algorithm produces a feasible schedule for a parallel task τ_i having integer-value subtask workloads and an integer deadline D.*

5.4 Application to Federated Scheduling

For completeness, we describe how Algorithm 1 can be incorporated into a federated scheduling algorithm for a complete task set, on a given number of identical processors. The procedure is outlined in Algorithm 2. Heavy tasks are scheduled on dedicated processors according to **Integer_List_Schedule**; the remaining processors are allocated to light tasks, which are scheduled according to an existing multiprocessor scheduling algorithm (e.g. partitioned EDF or partitioned RM). The algorithm fails if there are insufficient processors to schedule any task.

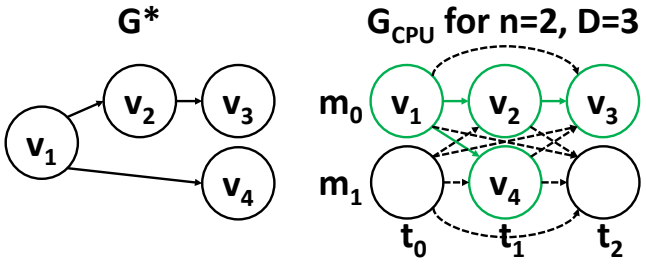

Figure 4: An isomorphism from a task with deadline 3 and unit-workload DAG $G*$ to a subgraph of G_{CPU} for 2 cores.

6 OPTIMAL PROCESSOR ASSIGNMENT

Despite being $\mathcal{NP}$-Complete in general, assigning an optimal number of processors to a constrained-deadline integer-valued parallel task often can be solved efficiently in practice. In this section, we discuss the problem of finding the minimum number of processors n on which a task τ_i with integer-valued deadline D_i and subtasks with integer-valued workloads (and therefore an integer-valued total workload C_i) can be feasibly scheduled.

Certainly, Algorithm 1 presented in the previous section might produce an optimal assignment; indeed, if it assigns $n_{min} = \lceil C_i/D_i \rceil$ cores, then the assignment is known to be optimal. However, in general, this is not guaranteed to be the case. For those cases where Algorithm 1 does not produce a feasible schedule on n_{min} cores, an optimal schedule can still be constructed. In [6], it is shown that the problem of producing a feasible schedule for a parallel task, given its DAG, can be represented as an integer linear program, then solved using a standard ILP solver.

We propose an alternative approach: the problem of constructing a feasible schedule for an integer-valued task on n processors, given its DAG, can be transformed in polynomial time to a subgraph isomorphism problem. Despite being $\mathcal{NP}$-Complete in general, algorithms exist that solve this efficiently in many cases. We outline a procedure by which this can be used to assign an optimal number of processors to a task; in Section 7.3, we demonstrate the viability of this method for a large set of randomly-generated synthetic tasks.

The procedure is illustrated in Algorithm 3. For a high-utilization integer-valued parallel task, it takes its representative DAG G, a deadline D, and a number of processors n. It decomposes G into its corresponding unit-workload DAG G^*, then generates a DAG G_{CPU} in which each vertex represents a time step t of execution on a processor $m \in \{1, \ldots, n\}$; there are $D*n$ such vertices, indexed $v_{t,m}$. Edges are constructed such that for any two vertices $v_{a,x}$, $v_{b,y}$, if $a < b$ then there exists an edge from $v_{a,x}$ to $v_{b,y}$; each vertex in G_{CPU} has an edge to every other vertex corresponding to a later time step. The algorithm then attempts to find a subgraph of G_{CPU} isomorphic to G^*; if it finds an isomorphism, it uses this to construct a schedule $\mathcal{S}$ for G^*, as illustrated in Figure 4. From Lemma 5.4, this can mapped to a schedule for the original DAG G. The time to construct G_{CPU} is $(D * n)^2$; adding this to the time to construct G^* from G, we have a time complexity of $O(D^2 * n^2 + C + |E|)$. Since $D*n \geq C$ (otherwise, the task is not schedulable on n processors) and $C^2 > |E|$, we can state this more simply as $O(D^2 * n^2)$. Constructing a schedule from the isomorphism f involves assigning each subtask $v_i \in G$ to processor m at time t given by the mapping $v_{t,m} = f(v_i)$; this simply traverses each vertex, and so can be performed in time $O(C)$.

Algorithm 3: Subgraph_Isomorphism_Schedule(G,D,n)

Input: Integer-valued task DAG G with deadline D and a number of processors n
Output: A feasible schedule $\mathcal{S}$

Initial Setup
  $G^* \leftarrow$ `Convert_Unit_DAG(`G`)`
  $\mathcal{S}$: an empty candidate schedule
Generate CPU DAG
  G_{CPU} ; ▷ `Empty DAG`
  for $t \leftarrow 0$ **to** $D-1$ **do**
    for $m \leftarrow 1$ **to** n **do**
      Create vertex $v_{t,m}$ in G_{CPU}
    end
  end
  forall $v_{t,m} \in G_{CPU}$ **do**
    for $t^* \leftarrow t+1$ **to** $D-1$ **do**
      for $m^* \leftarrow 1$ **to** n **do**
        Create edge $v_{t,m} \rightarrow v_{t^*,m^*}$ in G_{CPU}
      end
    end
  end
Find an isomorphism $f : G^* \rightarrow S_{CPU}$ for $S_{CPU} \subseteq G_{CPU}$
if f *exists* **then**
  forall $v_i \in G^*$ **do**
    $v_{t,m} \in G_{CPU} = f(v_i)$
    Assign v_i to processor m at time t in $\mathcal{S}$
  end
  return $\mathcal{S}$
else return `infeasible`;

6.1 Correctness of Algorithm 3

We begin with the following lemma:

Lemma 6.1. *For an integer-valued task τ with DAG G and deadline D, if there is a subset of G_{CPU} isomorphic to G^*, then there exists a feasible schedule on n cores.*

Proof. Assume there exists an isomorphism f between G^* and a subset of G_{CPU}. The schedule $\mathcal{S}$ produced by Algorithm 3 is feasible for the corresponding unit-workload task: since each vertex $v_i \in G^*$ is mapped to a vertex $v_{t,m}$ in G_{CPU}, and since t takes values in the range $[0, D-1]$, this means that each unit of work is scheduled before the deadline. Since $m \leq n$, no processor is assigned more than one unit of work at each time step. Further, for $v_i, v_j \in G^*$, if $v_i \prec v_j$, then $f(v_i) \prec f(v_j)$; the edges added to G_{CPU} guarantee that for $v_{a,x}, v_{b,y} \in G_{CPU}$, if $v_{a,x} \prec v_{b,y}$, then $a < b$. This implies that v_i is scheduled at an earlier time than v_j, so the schedule follows all precedence constraints specified by G^*. By Lemma 5.4, if $\mathcal{S}$ is feasible for G^* on n processors, then it can be mapped to a feasible schedule G on n processors. □

Next, we prove the inverse with two lemmas.

Lemma 6.2. *If there exists a feasible non-preemptive schedule for the unit-workload task represented by G^* on n processors, then there is an isomorphism f between G^* and a subset of G_{CPU}.*

Proof. A non-preemptive schedule of G^* on n processors must assign each unit of work represented by a vertex $v_i \in G^*$ to be executed on a processor $0 < i \leq n$ for some time $[t, t+1]$. All vertices must be so scheduled such that $t \leq D-1$ to guarantee that the task's deadline is met.

We argue that, given a feasible non-preemptive schedule, another non-preemptive schedule may be constructed such that all vertices are scheduled at some time $t \in \mathbb{N}_0$. Indeed, if some vertex v_i is scheduled at a time $t \in \mathbb{R}_+$, it may be scheduled instead for time $t' = \lfloor t \rfloor$; its successors will still begin execution after it, and every subtask will still complete execution before the deadline. All of v_i's predecessors must have been scheduled to start at some time $t^* \leq t-1$; this implies that they now start at a time $t^{*\prime} \leq \lfloor t-1 \rfloor$, i.e. $t^{*\prime} \leq t'-1$ and so complete by t'; thus, all predecessors of v_i still complete before v_i begins execution in the new schedule.

So, if G^* can be non-preemptively scheduled on n processors, each of its vertices is scheduled at some time step $t \in \mathbb{N}_0 < D$. This means that each vertex can be mapped to exactly one node $v_{t,m}$ in G_{CPU}. This defines a mapping $f : G^* \to G_{CPU}$. For $v_i, v_j \in G^*$, if $v_i \prec v_j$, a feasible schedule must have v_i execute before v_j. This implies that for $f(v_i) = v_{a,x}$ and $f(v_j) = v_{b,y}$, $a < b$; so there is an edge connecting $v_{a,x}$ to $v_{b,y}$ in G_{CPU}; thus, f is an isomorphism. □

Lemma 6.3. *If a unit-workload task can be scheduled on n processors, then there exists a feasible non-preemptive schedule.*

Proof. Assume that there is some unit-workload task τ with deadline $D \in \mathbb{N}$ represented by a DAG G^* for which there exists a feasible preemptive schedule. Assume that the schedule is work-conserving, i.e., there is no time at which a processor is idle when there is an available subtask for it to execute.[5] Since the schedule is work-conserving, the response time R of the task must be an integer. We say that the last preemption occurs at time t^*: subtask v_i is preempted by subtask v_j, and no preemption occurs after this. Assume that if v_j does not preempt v_i at this point, the task will not meet its deadline. This implies that, if v_j does not preempt v_i, then there is a subtask v_k for which $v_j \prec v_k$ that will complete execution at some time $t' > D$. Without loss of generality, let's consider the time t_c that is the maximum completion time among all subtasks $v_k : v_j \prec v_k$. We say that $t_c = D + \delta$. By preempting, v_j begins execution some time ϵ earlier than if it must wait for v_i to complete; since v_i has a unit workload, $0 < \epsilon < 1$. So, by preempting, the maximum completion time among all subtasks v_k becomes $t_c' \geq D + \delta - \epsilon$. At this point, no more future preemptions must occur, so all subtasks complete execution before D (under the assumption that the preemptive schedule is feasible). This implies that $R = t_c$, i.e., the completion time of the subgraph rooted at v_j is the completion time of the task. Since $R \in \mathbb{N}$, we know $D + \delta \in \mathbb{N}$ and $t_c' \in \mathbb{N}$. But since $t_c' + \epsilon \geq D + \delta$ and $\epsilon < 1$, this implies that for $D + \delta$ to be an integer, $\epsilon = 0$. This forms a contradiction, and so there must be a feasible schedule even if v_j does not preempt v_i. Thus, if there exists a feasible schedule for a unit-workload task, then a feasible non-preemptive schedule for that task must exist. □

[5] If there exists a feasible preemptive schedule that is not work-conserving, then a feasible schedule must also exist that is work-conserving; indeed, consider some processor that is idle at some time t in the non-work-conserving schedule, and assume that (1) there is some time $t' > t$ for which it is scheduled to execute work, but that (2) there is available work for it to do at time t. We can have it begin execution of the available work at time t, and as long as that work can be preempted by the work that was originally scheduled for time t', the schedule remains feasible.

The theorem follows:

Theorem 6.4. *For an integer-valued task τ with DAG G and deadline D, there exists a feasible schedule on n cores iff there is a subset of G_{CPU} isomorphic to G^*.*

6.2 Solving Subgraph Isomorphism Problems

Several algorithms exist for solving subgraph isomorphism problems [24]. In general, such algorithms must find a pattern graph inside a larger target graph, which can be solved by constraint programming models. One such algorithm, proposed by McCreesh and Prosser [29], introduces a backtracking approach that allows the search to be parallelized, and introduces other heuristic optimizations. This algorithm, along with a variety of domain-specific search techniques suitable to specific classes of graphs, has been implemented in the Glasgow Subgraph Solver [30], which can run sequentially or in parallel, and produces a subgraph isomorphism if one exists. In Section 7.3, we use the Glasgow Subgraph Solver to solve the subgraph isomorphism problem in Line 18 of Algorithm 3. We find that it is efficient in its execution, and provides a suitable means by which to assign processors optimally to high-utilization integer-valued tasks.

6.3 Application to Federated Scheduling

For completeness, we describe how Algorithm 3 can be applied to federated scheduling. Similarly to Algorithm 1, we can determine the minimum sufficient and maximum necessary number of cores that must be assigned to an integer-valued task to guarantee schedulability, then attempt to construct a feasible schedule for each number of processors in the range; the minimum number of processors on which a feasible schedule is produced is an optimal assignment, per Theorem 6.4. In fact, a call to **`Subgraph_Isomorphism_Schedule`** could be placed after Line 10 of Algorithm 1; this would allow Algorithm 1 to attempt to find a schedule using its pseudo-polynomial time heuristics first, before solving the $\mathcal{NP}$-Complete problem, for each candidate number of processors.

The modified **`Integer_List_Schedule`** could then be used in Algorithm 2 to provide federated scheduling over a set of tasks, guaranteeing that each high-utilization task is allocated the minimum sufficient number of dedicated processors.

7 EVALUATION

In this section we evaluate the new scheduling techniques presented in this paper. Section 7.1 examines how many processors are assigned based on Theorem 4.5, versus the original assignment described in [26], to gauge how often the new approach shows an improvement. Section 7.2 then compares the performance of the **CP+LNS** and **LNS+CP** heuristics to each other and to the optimal assignment of cores. Finally, Section 7.3 quantifies the mean

$C \in$	NUM	% FEWER	% CPUs
[3,10]	120	35.8	81.6
[11,100]	161,580	21.7	82.0
[101,1000]	166,005,300	8.70	86.4

Table 1: Comparison of Equations 1 and 2

and maximum run-times of a representative subgraph solver as a function of the probability of pair-wise graph edges.

7.1 New Federated Scheduling Bound

To gauge whether the new assignment of processors in Theorem 4.5 allocates fewer processors than the original approach in [26], we begin by comparing the values of n and n' according to Equations 1 and 2 respectively. We generate heavy integer-valued tasks characterized only by workload C, span L, and deadline D. For C, we iterate over all integers in the range $[3, 1000]$; for each workload, we generate values of D over all integers in the range $[1, C-1]$;[6] for each deadline, we generate values of L over all integers in the range $[1, D-1]$.[7] For each generated task, we compute n and n'.

Results are summarized in Table 1. We consider tasks with workloads in the ranges $C \in [3, 10]$, $C \in [11, 100]$, and $C \in [101, 1000]$. For each subset, we report the total number of tasks generated (labeled **NUM**), the percentage of tasks for which Equation 2 allocates fewer processors than the original (labeled **% FEWER**), and the percentage of total processors assigned by Equation 2 compared to the original (labeled **% CPUs**). Notice that as C increases, the range of D increases, and so the +1 terms in the numerator and denominator of Equation 2 contribute less to the assignment; this is reflected in the fact that as our range over C increases, the new assignment provides an improvement in fewer cases; nonetheless, the improvement remains significant for large values of C. For very large values of C and D, it may be possible to express values using a courser time resolution (e.g., milliseconds instead of microseconds) to see more improvement; a study of this is deferred to future work.

Next, to assess similar improvement for admission control, we consider task set admission under both naïve methods of allocating processors: for a system having m processors, we want to know the proportion of task sets having a given utilization that can be admitted, given core allocations according to Equations 1 and 2. We iterate over integer utilizations U in the range $[2, 64]$; for each value, we generate 1000 task sets with a total utilization of U. Each task set consists of a number of tasks chosen uniformly from the range $[1, U/2]$ (where $U/2$ is rounded down). This guarantees that each task can be high-utilization. We use a modified UUniSort [9] algorithm to assign a utilization to each task. Utilization values are selected from the real numbers, and we guarantee that each task has a utilization greater than 1. Each task is then assigned an integer-valued deadline in the range $[2, 1000]$, selected using the log-uniform distribution described in [16] (Eqn 4). Each task τ_i in the set has its workload assigned according to $C_i = \lfloor U_i * D_i \rfloor$ to guarantee an integer value. Finally, each task's span is selected uniformly from the integers $[1, D_i - 1]$. For each task set so generated, we compute the total number of cores n and n' required by the task set if assigned according to Equations 1 and 2 respectively. This

[6] For $D \geq C$, a single processor is sufficient.
[7] The result of [26] cannot be applied to tasks having $L = D$.

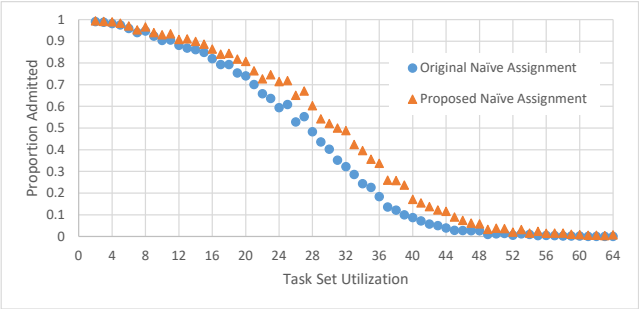

Figure 5: Comparison of admitted task sets for $m = 64$

allows us to determine, for systems having $m = \{4, 8, 16, 32, 64\}$ processors, how many task sets at each utilization will be admitted onto the system, given the two federated scheduling algorithms.

Results for 64 processors are illustrated in Figure 5. Results for systems having fewer numbers of cores are qualitatively similar: for task sets with low utilizations compared to the number of processors, both methods are able to admit almost all of the task sets. For task sets with utilizations close to the number of processors, neither method is able to admit many of the task sets. For task sets with utilizations from about $1/4 - 3/4$ of the number of processors, using the new allocation of Equation 2 allows admission of significantly more task sets compared to the original.

7.2 Comparing Heuristics

We next compare the heuristics discussed in Section 5; as these exploit a parallel task's DAG structure, we evaluate these methods for synthetic tasks over a domain of randomly-generated representative DAGs. For each generated task, we create a DAG with a number of nodes sampled uniformly from $5 - 250$; each node is then assigned an integer workload in the range $5 - 10$, and the total workload C is then calculated. The structure of the DAG is then defined by edges assigned according to the Erdős-Rényi method [10]: for each pair of vertices, an edge connecting them is added with some probability p; the edge is always directed from smaller to larger vertex index to guarantee that the graph remains acyclic. Once this procedure completes, we continue to add edges at random between disconnected vertices until the graph is weakly connected, similarly to the method used in [14]. Once the DAG is generated, its span L is calculated, and the task is assigned a deadline sampled uniformly from the range $[L, C-1]$. We generate 1000 tasks each for values of p ranging over $0.05 - 0.95$ in steps of 0.05, for a total of 19,000 task DAGs. For each task, we determine the number of cores assigned to it by the **CP+LNS** and **LNS+CP** heuristics and use the method presented in Algorithm 3 to find the optimal processor assignment.

We summarize several findings in Table 2. First, observe that over the space of 19,000 generated task DAGs, Equations 1 and 2 both assign $\lceil C/D \rceil$ cores about half the time (though Equation 2 does so for about 0.9% more of the tasks). Despite the similarity of this statistic, Equation 2 outperforms Equation 1 significantly: Equation 1 is undefined (due to a 0 in the denominator) for 1.2% of the tasks, and it allocates more cores than Equation 2 for an additional 11.6% of the generated tasks.

We also find that the **CP+LNS** and **LNS+CP** heuristics presented in Section 5 perform very well. They both produce optimal core

Criteria	Occurrences	% of Total
Tasks	19,000	100
Eqn. 1 is undefined	222	1.2
$\lceil C/D \rceil$ = Eqn. 1	9,404	49.5
$\lceil C/D \rceil$ = Eqn. 2	9,567	50.4
Eqn. 2 assigns fewer than Eqn. 1	2202	11.6
CP+LNS is optimal	18,641	98.1
LNS+CP is optimal	18,546	97.6
CP+LNS assigns fewer than **LNS+CP**	98	0.5

Table 2: Heuristic Performance Over Random DAGs

assignments for a large proportion (> 97%) of the generated tasks. This suggests that, for the class of DAGs considered, either heuristic can be used to produce an optimal core assignment with reasonable certainty. Further, we find that **CP+LNS** never assigns more cores than **LNS+CP** for the tested DAGs; this motivates the choice, in Algorithm 1, to attempt **CP+LNS** first for each value n, before attempting **LNS+CP**.

7.3 Execution Time of Subgraph Solver

For each of the parallel tasks generated in the previous subsection, we measured the time it took to find the optimal processor assignment according to Algorithm 3. The algorithm runs in constant (and brief) time for tasks where where $\lceil C/D \rceil$ is equal to Equation 2; we exclude these cases from our results.

Measurements were taken on a system with two Intel Xeon Gold 6130 Skylake processors running at 2.1 GHz, and with 32GB of memory. HyperThreading, SpeedStep, and TurboBoost were all enabled. We used the Glasgow Subgraph Solver [30] to perform Line 18 of the algorithm.[8] While the solver supports parallel solution search, we invoked it in a single-threaded sequential mode; this allowed us to write a multithreaded wrapper to run multiple instances of the solver, each for a different DAG from the set of 19,000. Results are plotted in Figure 6, where we show the mean and maximum execution time for each pairwise edge probability p used to generate the DAGs. In general, DAGs with higher values of p (for which we would expect more edges) require more time to solve. The longest measured execution time was just over 21 minutes; this suggests that the method proposed in Section 6 for constructing an optimal schedule on a minimum number of cores for an integer-valued task is viably handled by the Glasgow Subgraph Solver, even when running it in a single thread.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we have considered federated scheduling for integer-valued tasks. We presented Equation 2, a new, constant-time method for assigning sufficient cores to these tasks, and demonstrated that in practice, it may allow more task sets to be admitted on systems with constrained numbers of processors compared to the original assignment in [26]. We also presented two pseudo-polynomial time heuristics for list scheduling of unit workload tasks, and demonstrated that these can be applied to arbitrary integer-valued tasks. Over a set of synthetic tasks generated according to the Erdős-Rényi

[8]The Glasgow Subgraph Solver is available at https://github.com/ciaranm/glasgow-subgraph-solver; we used the master branch at commit 5f60e1f (the latest commit on February 9, 2022)

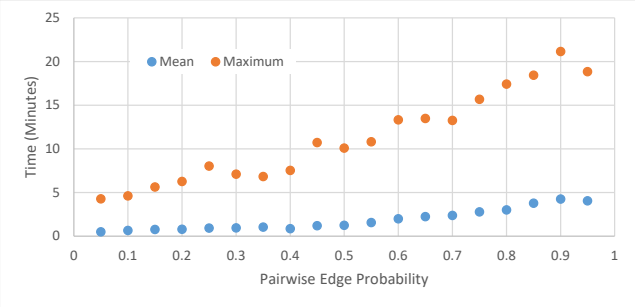

Figure 6: Solver Execution Times

method, both heuristics produce optimal processor assignments more than 97% of the time; the optimal values were found by our proposed representation of the scheduling problem as a subgraph isomorphism search. Using the Glasgow Subgraph Solver, it is viable to solve the problem for the generated tasks, even with a single-threaded implementation.

For future work, we would like to apply these same heuristics to sets of tasks generated according to other random DAG generation methods, like those detailed in [10]. We intend to study the costs and benefits of expressing workload and deadline values using a courser time resolution (e.g., for values expressed in microseconds, by rounding up the workload and rounding down the deadline to the nearest millisecond) or of converting floating-point to integer values. We will also consider the application of our new constant-time core allocation method, as well as these heuristics, to semi-federated scheduling [22], reservation-based federated scheduling [36], and federated scheduling of elastic tasks [32].

ACKNOWLEDGMENTS

Many thanks to Jing Li, Abusayeed Saifullah, Chenyang Lu, and Son Dinh for insights into their work, which inspired this study. Thanks also to Sanjoy Baruah and Abhishek Singh for sharing their wealth of knowledge in the state-of-the-art.

The research presented in this paper was supported in part by NSF grants CSR-1814739 and CNS-17653503 and NASA grant 80NSSC21K1741.

REFERENCES

[1] Björn Andersson and Dionisio de Niz. 2012. Analyzing Global-EDF for Multiprocessor Scheduling of Parallel Tasks. In *Principles of Distributed Systems*, Roberto Baldoni, Paola Flocchini, and Ravindran Binoy (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 16–30.

[2] Sanjoy Baruah. 2013. Partitioned EDF Scheduling: A Closer Look. *Real-Time Syst.* 49, 6 (nov 2013), 715–729. https://doi.org/10.1007/s11241-013-9186-0

[3] Sanjoy Baruah. 2015. The federated scheduling of constrained-deadline sporadic DAG task systems. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1323–1328. https://doi.org/10.7873/DATE.2015.0200

[4] Sanjoy Baruah. 2015. Federated Scheduling of Sporadic DAG Task Systems. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 179–186. https://doi.org/10.1109/IPDPS.2015.33

[5] Sanjoy Baruah. 2015. The federated scheduling of systems of conditional sporadic DAG tasks. In *2015 International Conference on Embedded Software (EMSOFT)*. 1–10. https://doi.org/10.1109/EMSOFT.2015.7318254

[6] Sanjoy Baruah. 2022. An ILP representation of a DAG scheduling problem. *Real-Time Systems* 58, 1 (01 Mar 2022), 85–102. https://doi.org/10.1007/s11241-021-09370-7

[7] Sanjoy Baruah and N. Fisher. 2006. The partitioned multiprocessor scheduling of deadline-constrained sporadic task systems. *IEEE Trans. Comput.* 55, 7 (2006), 918–923. https://doi.org/10.1109/TC.2006.113

[8] Sanjoy K. Baruah and Nathan Wayne Fisher. 2007. The Partitioned Dynamic-Priority Scheduling of Sporadic Task Systems. *Real-Time Syst.* 36, 3 (aug 2007), 199–226. https://doi.org/10.1007/s11241-007-9022-5

[9] Enrico Bini and Giorgio C. Buttazzo. 2005. Measuring the Performance of Schedulability Tests. *Real-Time Syst.* 30, 1–2 (May 2005), 129–154. https://doi.org/10.1007/s11241-005-0507-9

[10] Louis-Claude Canon, Mohamad El Sayah, and Pierre-Cyrille Héam. 2019. A Comparison of Random Task Graph Generation Methods for Scheduling Problems. In *Euro-Par 2019: Parallel Processing*, Ramin Yahyapour (Ed.). Springer International Publishing, Cham, 61–73.

[11] J. Chen. 2016. Partitioned Multiprocessor Fixed-Priority Scheduling of Sporadic Real-Time Tasks. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE Computer Society, Los Alamitos, CA, USA, 251–261. https://doi.org/10.1109/ECRTS.2016.26

[12] E. G. Coffman and R. L. Graham. 1972. Optimal scheduling for two-processor systems. *Acta Informatica* 1, 3 (01 Sep 1972), 200–213. https://doi.org/10.1007/BF00288685

[13] U.M.C. Devi and J.H. Anderson. 2005. Tardiness bounds under global EDF scheduling on a multiprocessor. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*. 12 pp.–341. https://doi.org/10.1109/RTSS.2005.39

[14] Son Dinh, Christopher Gill, and Kunal Agrawal. 2020. Efficient Deterministic Federated Scheduling for Parallel Real-Time Tasks. In *2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 1–10. https://doi.org/10.1109/RTCSA50079.2020.9203660

[15] Glenn A. Elliott, Kecheng Yang, and James H. Anderson. 2015. Supporting Real-Time Computer Vision Workloads Using OpenVX on Multicore+GPU Platforms. In *2015 IEEE Real-Time Systems Symposium*. 273–284. https://doi.org/10.1109/RTSS.2015.33

[16] P. Emberson, R. Stafford, and R.I. Davis. 2010. Techniques For The Synthesis Of Multiprocessor Tasksets. In *WATERS workshop at the Euromicro Conference on Real-Time Systems*. 6–11. 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems ; Conference date: 06-07-2010.

[17] David Ferry, Gregory Bunting, Amin Maqhareh, Arun Prakash, Shirley Dyke, Kunal Agrawal, Chris Gill, and Chenyang Lu. 2014. Real-time system support for hybrid structural simulation. In *2014 International Conference on Embedded Software (EMSOFT)*. 1–10. https://doi.org/10.1145/2656045.2656067

[18] M. Fujii, T. Kasami, and K. Ninomiya. 1969. Optimal Sequencing of Two Equivalent Processors. *SIAM J. Appl. Math.* 17, 4 (1969), 784–789. http://www.jstor.org/stable/2099319

[19] M. R. Garey and D. S. Johnson. 1977. Two-Processor Scheduling with Start-Times and Deadlines. *SIAM J. Comput.* 6, 3 (1977), 416–426. https://doi.org/10.1137/0206029 arXiv:https://doi.org/10.1137/0206029

[20] R. L. Graham. 1969. Bounds on Multiprocessing Timing Anomalies. *SIAM J. Appl. Math.* 17, 2 (1969), 416–429. http://www.jstor.org/stable/2099572

[21] T. C. Hu. 1961. Parallel Sequencing and Assembly Line Problems. *Oper. Res.* 9, 6 (dec 1961), 841–848. https://doi.org/10.1287/opre.9.6.841

[22] Xu Jiang, Nan Guan, Xiang Long, and Wang Yi. 2017. Semi-Federated Scheduling of Parallel Real-Time Tasks on Multiprocessors. In *2017 IEEE Real-Time Systems Symposium (RTSS)*. 80–91. https://doi.org/10.1109/RTSS.2017.00015

[23] Junsung Kim, Hyoseung Kim, Karthik Lakshmanan, and Ragunathan Rajkumar. 2013. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *2013 ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*. 31–40.

[24] Lars Kotthoff, Ciaran McCreesh, and Christine Solnon. 2016. Portfolios of Subgraph Isomorphism Algorithms. In *Learning and Intelligent Optimization - 10th International Conference, LION 10, Ischia, Italy, May 29 - June 1, 2016, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10079)*, Paola Festa, Meinolf Sellmann, and Joaquin Vanschoren (Eds.). Springer, 107–122. https://doi.org/10.1007/978-3-319-50349-3_8

[25] J. Li, K. Agrawal, C. Lu, and C. Gill. 2013. Analysis of Global EDF for Parallel Tasks. In *2013 25th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE Computer Society, Los Alamitos, CA, USA, 3–13. https://doi.org/10.1109/ECRTS.2013.12

[26] Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah. 2014. Analysis of federated and global scheduling for parallel real-time tasks. In *2014 26th Euromicro Conference on Real-Time Systems*. IEEE, 85–96.

[27] Jing Li, David Ferry, Shaurya Ahuja, Kunal Agrawal, Christopher Gill, and Chenyang Lu. 2017. Mixed-Criticality Federated Scheduling for Parallel Real-Time Tasks. *Journal of Real-Time Systems* 53, 5 (2017), 760–811.

[28] Jing Li, Kevin Kieselbach, Kunal Agrawal, Christopher Gill, and Chenyang Lu. 2016. Randomized Work Stealing for Large Scale Soft Real-time Systems. In *IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 203–214.

[29] Ciaran McCreesh and Patrick Prosser. 2015. A Parallel, Backjumping Subgraph Isomorphism Algorithm Using Supplemental Graphs. In *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9255)*, Gilles Pesant (Ed.). Springer, 295–312. https://doi.org/10.1007/978-3-319-23219-5_21

[30] Ciaran McCreesh, Patrick Prosser, and James Trimble. 2020. The Glasgow Subgraph Solver: Using Constraint Programming to Tackle Hard Subgraph Isomorphism Problem Variants. In *Graph Transformation - 13th International Conference, ICGT 2020, Held as Part of STAF 2020, Bergen, Norway, June 25-26, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12150)*, Fabio Gadducci and Timo Kehrer (Eds.). Springer, 316–324. https://doi.org/10.1007/978-3-030-51372-6_19

[31] R.R. Muntz and E.G. Coffman. 1969. Optimal Preemptive Scheduling on Two-Processor Systems. *IEEE Trans. Comput.* C-18, 11 (1969), 1014–1020. https://doi.org/10.1109/T-C.1969.222573

[32] James Orr, Johnny Condori Uribe, Chris Gill, Sanjoy Baruah, Kunal Agrawal, Shirley Dyke, Arun Prakash, Iain Bate, Christopher Wong, and Sabina Adhikari. 2020. Elastic Scheduling of Parallel Real-Time Tasks with Discrete Utilizations. In *Proceedings of the 28th International Conference on Real-Time Networks and Systems* (Paris, France) *(RTNS 2020)*. Association for Computing Machinery, New York, NY, USA, 117–127. https://doi.org/10.1145/3394810.3394824

[33] C. H. Papadimitriou and M. Yannakakis. 1979. Scheduling Interval-Ordered Tasks. *SIAM J. Comput.* 8, 3 (1979), 405–409. https://doi.org/10.1137/0208031 arXiv:https://doi.org/10.1137/0208031

[34] Michael L Pinedo. 2016. *5.1* (fifth ed.). Springer, 114–124.

[35] Marion Sudvarg, Jeremy Buhler, James H. Buckley, Wenlei Chen, et al. 2021. A Fast GRB Source Localization Pipeline for the Advanced Particle-astrophysics Telescope. *PoS* ICRC2021 (2021), 588. https://doi.org/10.22323/1.395.0588

[36] Niklas Ueter, Georg von der Brüggen, Jian-Jia Chen, Jing Li, and Kunal Agrawal. 2018. Reservation-Based Federated Scheduling for Parallel Real-Time Tasks. In *2018 IEEE Real-Time Systems Symposium (RTSS)*. 482–494. https://doi.org/10.1109/RTSS.2018.00061

[37] J.D. Ullman. 1975. NP-complete scheduling problems. *J. Comput. System Sci.* 10, 3 (1975), 384–393. https://doi.org/10.1016/S0022-0000(75)80008-0