Near Optimal Task Graph Scheduling with Priced Timed Automata and Priced Timed Markov Decision Processes

Technical Report

Anne Ejsing, Martin Jensen, Marco Muñiz, Jacob Nørhave, and Lars Rechter

Aalborg University, Aalborg, Denmark
{aejsin,martje,jnarha,lrecht}16@student.aau.dk, muniz@cs.aau.dk

Abstract. Task graph scheduling is a relevant problem in computer science with application to diverse real world domains. Task graph scheduling suffers from a combinatorial explosion and thus finding optimal schedulers is a difficult task. In this paper we present a methodology for computing near-optimal preemptive and non-preemptive schedulers for task graphs. The task graph scheduling problem is reduced to location reachability via the fastest path in Priced Timed Automata (PTA) and Priced Timed Markov Decision Processes (PTMDP). Additionally, we explore the effect of using chains to reduce the computation time for finding schedules. We have implemented our models in Uppaal Cora and Uppaal Stratego. We conduct an exhaustive experimental evaluation where we compare our resulting schedules with the best-known schedules of a state of the art tool. A significant number of our resulting schedules are shown to be shorter than or equal to the best-known schedules.

Keywords: Model Checking, Scheduling, Priced Timed Automata, Priced Timed Markov Decision Processes, Uppaal, Preemption, Task Graph

1 Introduction and Motivation

The task graph scheduling problem is a well known and widely discussed problem in mathematics and computer science. Creating optimal and near-optimal schedules are relevant in many real-world applications, such as scheduling of computations in spreadsheets for parallel execution [6]. Given a task graph, computing an optimal schedule is NP-complete [10]. Applications such as spreadsheets, induce large task graphs and computation of optimal schedulers in such domains is intractable. Therefore,

we investigate how to obtain a near-optimal solution. Furthermore, we add preemption to find shorter schedules. As the task graph scheduling problem is reducible to the reachability problem in timed automata [2], we use extensions of timed automata to find near-optimal solutions, being Priced Timed Automata and Priced Timed Markov Decision Processes.

Our main contribution is a methodology for modelling task graphs, with preemptive and non-preemptive schedulers using priced timed automata and priced timed markov decision processes. The intuition of our preemptive models is that we preempt all executing tasks when any of the executing tasks have finished. In this work we assume that preemption has no cost, however an arbitrary cost can be easily implemented. We thoroughly evaluate our approach on the standard task graph set of Kasahara and Narita [8], which provides the best-known schedules. Furthermore, we compare schedules that allow preemption, to schedules that do not. This is to conclude when it is beneficial to use preemption. Lastly, we evaluate our implementations of our models based on priced timed automata and priced timed markov decision processes to investigate when either is beneficial.

Related Work In this article, we use extensions of timed automata to find task graph schedules. Timed automata originates from the work of Alur and Dill [4]. In [2] Abdeddaïm, Kerbaa, and Maler use timed automata to compute schedulers for task graphs. They do so non-preemptively and they compose these automata using a special parallel mutual exclusion composition operator. They compute a schedule via location reachability in the resulting automata. The work of Kasahara and Narita [8,9] comprises a standard task graph set and the best schedules they found for each of them. To obtain these non-preemptive schedules they use a branch and bound algorithm. In our work, we use the standard task graph set and the best-known schedules of Kasahara and Narita [8], to evaluate the quality of the schedules we obtain with our models. In [6] the authors show an attempt of analyzing spreadsheets for parallel execution via model checking. The authors focus on optimal schedulers and do not use preemption. They explore only small and few task graphs. Contrary to the work listed, we model task graphs both preemptively and non-preemptively. Besides we use priced timed automata and priced timed markov decision processes, implemented in Uppaal Cora and Uppaal Stratego, respectively. This is to explore more options for achieving

shorter schedules. Lastly, we achieve near-optimal results on larger task graphs, rather than optimal results on small task graphs as in[6].

Outline This paper is structured as follows; Section 2 introduces the theory behind task graph scheduling and preemption. Section 2.1 introduces the notion of chain decomposition of a task graph. Following this, Section 3 presents the relevant theory of priced timed automata and priced timed markov decision processes. The implementation and methodology for modelling our models in Uppaal Cora and Uppaal Stratego is documented in Section 4. In Section 5 we compare the schedules obtained by our approach with the best-known schedules.

2 Task Graph Scheduling

In this section, we define *task graphs* and *schedules* in the style of [2,1], we also introduce the notion of preemption in this context. Task graph scheduling is the activity of assigning tasks to machines. The result of task graph scheduling is a *schedule* which is a function that maps tasks to execution start times and durations.

Definition 1 (Task Graph). *A task graph is a triple:* $(\mathcal{P}, \sqsubset, D)$ *where* $\mathcal{P} = \{P_1, ..., P_i\}$ *is the set of all tasks,* $\sqsubset$ *is a strict partial-order relation on the set of tasks, and* D *is a function,* $D : P \to \mathbb{N}$ *that assigns duration to each of the tasks of* P*.*

Let M be a set of j identical machines, $M = \{m_1, ..., m_j\}$. The problem consists of assigning tasks to machines in periods of time such that:

- A task can run iff all of its predecessors are completed.
- Each machine can run at most one task at a time.

Such an assignment is known as a schedule. Schedules can be created either preemptively or non-preemptively.

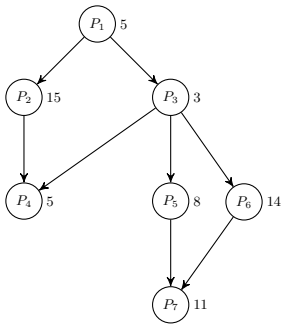
Definition 2 (Possible and Optimal Schedules). *For a task graph* $(\mathcal{P}, \sqsubset, D)$ *and* j *machines, a possible schedule is a function* $start : \mathcal{P} \to S$ *where* $S \subseteq \mathbb{R}_{\geq 0} \times \mathbb{R}_{>0}$ *is a set of pairs* (s, d)*, denoting respectively start times and intermediate durations of the task. The function must satisfy:*

- *No task can run before all of its predecessors have completed, formally: For every* $P, P' \in \mathcal{P}$ *where* $P' \sqsubset P$ *then* $inf(\{s \mid (s,d) \in start(P)\}) \geq sup(\{s' + d' \mid (s',d') \in start(P')\})$
- *At any point in time at most* j *machines can run tasks, formally: For any* $t \in \mathbb{R}^+$ *then* $|\{P \mid P \in \mathcal{P} \text{ and } (s,d) \in start(P) \text{ and } t \cap [s, s+d] \neq \emptyset\}| \leq j$
- *Running a single task in parallel is not allowed, formally: For every two pairs* $(s,d), (s',d') \in start(P)$ *where* $P \in \mathcal{P}$ *and* $s \neq s'$ *or* $d \neq d'$ *then either* $s + d \leq s'$ *or* $s' + d' \leq s$.
- *The sum of the intermediate durations of a task is equal to the full duration of the task, formally: For every* $P \in \mathcal{P}$ *then* $\sum_{(s,d) \in start(P)} d = D(P)$

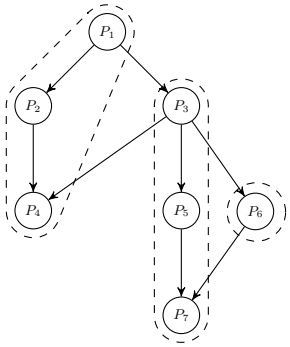
The length of a schedule is the time when the last task finishes execution, formally; $sup(\{s + d \mid (s,d) \in start(P) \text{ for every } P \text{ in } \mathcal{P}\})$. *An optimal schedule has the minimal length.*

Note that the set of non-preemptive schedules is a subset of that of preemptive schedules where the schedule contains only one pair (s,d) for each task.

The scheduler's goal is to obtain a schedule of minimal length. If the number of machines is greater than or equal to the number of tasks, this is trivial; the scheduler assigns a machine to each task. With fewer machines than tasks, the scheduler must decide which tasks to run first. To demonstrate the complexity, consider Figure 1a. The intuitive non-preemptive way to schedule is to assign tasks to machines as they become available (see Figure 2a). We achieve a shorter schedule when one machine idles while the other computes task 3 (see Figure 2b). Despite the optimal non-preemptive scheduler reducing the length of the schedule, it still contains idle time. To reduce this, we use preemption which results in a shorter schedule than Figure 2b, which is seen in Figure 2c. Now, the only idle time present is to fulfill the dependencies. The dilemma of the non-preemptive scheduler (whether optimality is reached by computing a task or idling) is now irrelevant. This is because, for the preemptive scheduler, it is *always* optimal to compute a task if possible. This is based on our assumption that context-switching is free. The task graph scheduling problem is a vast combinatorial problem and is NP-complete [10]. We, therefore, explore how to find *near optimal schedules* rather than optimal ones.



(a) Example of a task graph



(b) Chain decomposition

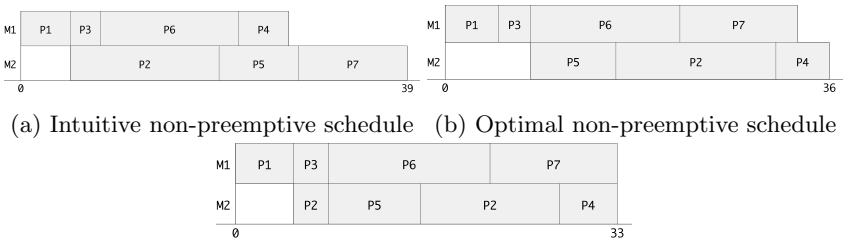
Fig. 1: Example of task graph and chain decomposition

2.1 Scheduling with Chains

To find all possible schedules, the scheduler must consider all combinations of executing all tasks while still fulfilling each task's dependencies. Thus, it must iterate through every task of the task graph each time a new task is to be scheduled. To reduce the number of tasks that the scheduler has to iterate through, we adopt the use of chains as in [1]. A chain is a directly connected route in a graph representation of a partially ordered set. We formally define chains in Definition 3. We define the decomposition of a partial order into chains in Definition 4.

Definition 3 (Chain). *A chain, C, in a strict partially ordered set, $(\mathcal{P}, \sqsubset)$, is a subset $C \subseteq \mathcal{P}$ such that for any two tasks $P, P' \in C$ where $P \neq P'$ then $P \sqsubset P'$ or $P' \sqsubset P$.*

Definition 4 (Chain Decomposition). *A chain decomposition is a partition of the elements of a partial order into chains.*



(a) Intuitive non-preemptive schedule (b) Optimal non-preemptive schedule

(c) Optimal preemptive schedule

Fig. 2: Examples of schedules obtained using preemption and non-preemption.

The decomposition of the task graph of Figure 1a to chains is seen in Figure 1b. The task graph thus comprises three chains being $chain_1 = \{P_1, P_2, P_4\}$, $chain_2 = \{P_3, P_5, P_7\}$, and $chain_3 = \{P_6\}$. Note that $\sqsubset$ totally orders the elements in each of the chains, as the definition of chains adds comparability to these subsets. As $\sqsubset$ from Definition 1 describes a dependency, then there is exactly one task in each chain that does not depend on another task in the chain; the least element of the chain. Once the task that is the least element is computed, we remove it from its chain such that the chain has a new least element. This way, the scheduler only needs to iterate through the least element of each chain, rather than iterating through every task when finding the tasks that can be scheduled. We state that we can do this without limiting the scheduler from creating any possible schedule.

Lemma 1 (Chain Optimality). *Given a task graph, any possible schedule, and any chain decomposition, then the schedule can be computed by considering only the least elements of the chains at any point in time.*

Computation of Chains Note that due to Lemma 1 it is not required to compute an optimal chain decomposition to preserve the best possible schedule. We have developed an algorithm which computes a chain cover of a given task graph. The intuition of the algorithm is to use the number of predecessors for each node as a heuristic when making chains, by choosing to add the vertices with the fewest number of predecessors to a chain. The algorithm has quadratic time complexity on the number of tasks.

3 Priced Timed Automata and Priced Timed Markov Decision Processes

Finding the optimal schedule in a task graph can be reduced to location reachability in the theory of Timed Automata [4,1]. The optimal schedule corresponds to the path with minimal time in the corresponding automaton. We extend the work in [1] by using the extensions of timed automata, we use Priced Timed Automata (PTA) and Priced Timed Markov Decision Processes (PTMDP). Both PTA and PTMPD, are useful for expressing task graph scheduling as the duration of tasks can be expressed using clocks. Additionally by using prices, we can optimize for the lowest total price, where the price could represent the difficulty of execution. Finding the cheapest path, based on the price variable, to a given goal location is also known as the *optimal reachability problem*, and when we add notions of price and time it is denoted *cost and time bounded optimal reachability problem*

In order to define how to model and solve task graph scheduling, we define the minimal required syntax and semantics of PTA and PTMDP. Our definitions are in the style of [7,3]. We refer the reader to [7,3,11] for detailed definitions.

3.1 Priced Timed Automata

In this extended version of Timed Automata a price can be specified for staying in specific locations. Prices are accumulated in a single continuous variable and specifies a price per time unit for any given location of the automaton.

Definition 5 (PTA). *A PTA* $\mathcal{A} = (L, l_0, X, \Sigma, E, P, Inv)$ *is a tuple where* L *is a finite set of locations,* $l_0 \in L$ *is the initial location,* X *is a finite set of non-negative real-valued clocks,* Σ *is a finite set of actions,* $E \subseteq L \times \mathcal{B}(X) \times \Sigma \times 2^X \times L$ *is a finite set of edges,* $P : L \to N$ *assigns a price-rate to each location, and* $Inv : L \to \mathcal{B}(X)$ *sets an invariant for each location.*

Semantically, a PTA $\mathcal{A}$ is a priced transition system, consisting of states, an initial state, a finite set of actions, and a transition relation. The states comprise pairs (l, v) with l being some location and v being a corresponding clock valuation, such that it fulfills the invariant in that

location. The finite set of actions correspond to Σ. Lastly, the transition relation is (similarly to Timed Automata) defined as action transitions and delay transitions, where in action transitions using an action we reach a new state where the location is new but the clock has not advanced. Naturally, the guard of the edge and the invariant of the new location must be satisfied. For delay transitions we reach a new state where the location remains the same but time delays. Again, the invariant of the location must remain fulfilled. Hence, the price of an action transition is 0, whereas the price of a delay transition is proportional to the delay according to the price rate of the given location.

A run through a PTA is an alternating sequence of action and delay transitions. The length of a run can be described as the number of action transitions, which is the *logical* length, or by the total time that has advanced through the delay transitions, which is denoted *metric* length.

3.2 Priced Timed Markov Decision Processes

PTMDP extends PTA with Markov Decision Processes. To define PTMDP formally we must first understand the notion of a Priced Timed Game, PTG; A PTG $\mathcal{G}$ is a PTA whose actions Σ are divided into controllable (Σ_c) and uncontrollable (Σ_u) actions. We now define PTMDP formally, as we assume the choices of delay and uncontrollable actions are stochastic and given according to a (delay,action)-probability density function for a given state.

Definition 6 (Priced Timed Markov Decision Processes). *A PTMDP is a pair $\mathcal{M} = (\mathcal{G}, \mu^u)$, where $\mathcal{G} = (L, l_0, X, \Sigma_c, \Sigma_u, E, P, Inv)$ is a PTG, and μ^u is a family of density-functions, $\{\mu^u_q : \exists l \exists v . q = (d, v)\}$, with $\mu^u_q(d, u) \in \mathbb{R}_{\geq 0}$ assigning the density of the environment aiming at taking the uncontrollable action $u \in \Sigma_u$ after a delay of d from state q.*

Using the notion of PTMDPs the cost and timed bounded reachability problem consists of finding a *strategy* that will reach the given goal location(s) within a given amount of time or cost. Informally, for PTMDPs a strategy is a family of probability density functions that assigns the density of the controller aiming at taking the controllable action after a given delay from a given state. Intuitively, the strategy is defined similarly to μ^u for Definition 6, however assigning the density of the controllable actions. For the formal definition of a strategy we refer the reader to [7].

When the environment is stochastic such a strategy can be determined using machine learning, in particular reinforcement learning.

4 Near Optimal Scheduling

In this section we explain the methodology for modeling task graphs with and without preemption. We model these using both PTA and PTMDPs. We use Uppaal Cora to implement PTA models and Uppaal Stratego to implement PTMDP models. Uppaal provides a C-like language, which allows for the use of variables, loops, etc. in the models. Uppaal Cora uses PTA and solves the cost optimal reachability problem using a branch and bound algorithm. Several strategies for branching are available, for example *smallest heuristic first* and *random optimal depth first* to obtain near-optimal solutions and *best first* for an optimal solution. Uppaal Stratego uses various types of reinforcement learning to create a strategy that minimizes a time-bounded reachability function [12].

We model task graphs as chains rather than individual tasks. This reduces the number of comparisons needed to know which tasks have their dependencies fulfilled. Thus, by modeling chains, we can reduce the computation time of creating schedules. The complexity of pre-computing the chains is quadratic on the number of tasks. For our models, chains can be computed fast and the time of computation is negligible compared to the time of computing a scheduler. Finally, we allow preemption to occur a once one of the scheduled tasks has finished executing.

4.1 Non-Preemptive Models

For the non-preemptive case, our Uppaal Cora and our Uppaal Stratego models are alike and comprise two templates: `Composer` and `Chain`. The `Composer` template is simple and is therefore only described in text. The `Composer` has two locations: `Init` and `Done`. The transition to `Done` is enabled (and taken) when all tasks have finished executing. The `Chain` template is seen in Figure 3 and is created in Uppaal Stratego. The intuition of the `Chain` template is that it starts in `Idle` and then takes the transition to `Running` to execute a task. This is possible when a machine is available and a task that is the least element of a chain has all its dependencies completed. When the task has finished executing, the

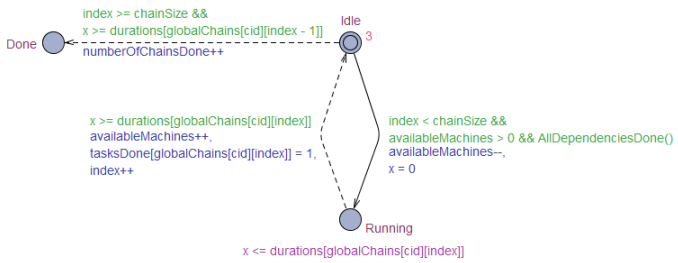


Fig. 3: Uppaal Stratego non-preemptive Chain template

transition back to idle is enabled and taken. This loop between `Idle` and `Running` continues until all tasks in that chain finish executing.

In Uppaal Stratego transitions are either controllable or uncontrollable. The controllable transitions denoted by solid arrows, are the choices on which Uppaal Stratego will use its machine learning algorithms for optimization. Thus, in the Uppaal Stratego model all transitions are uncontrollable except the one from `Idle` to `Running`. In this way, Uppaal Stratego can decide and optimize when to execute which tasks. We give the `Idle` state and the `Init` state of the composer an exponential rate to increase the probability of leaving the locations, thus ensuring advancement in the model.

With Uppaal Cora the `Chain` template is identical to Figure 3, except all edges are controllable and it has no exponential rate. The exponential rate is not available in Uppaal Cora as using the `cost'` variable ensures advancement in the model. Uppaal Cora models priced TA, so we use the `cost'` variable in the composer template. For every time unit spent in the `Init` location, the `cost'` increases by one. This value is the one Uppaal Cora minimizes in its branch and bound algorithm.

4.2 Preemptive Models

Our preemptive models share the structure of the non-preemptive, and the `Composer` templates are identical. The preemptive `Chain` template made in Uppaal Stratego is seen in Figure 4. Like in the non-preemptive case, it comprises three locations being `Done`, `Idle`, and `Running` and the general intuition remains the same. We express the preemption in the two transitions from `Running` to `Idle`. The leftmost is taken when a task has

finished executing, and it signals this to all other chains. All other chains in `Running` receive the signal and preempt the task they were running by taking the other transition from `Running` to `Idle`. This behavior is implemented using `select` statements and `broadcast` channels in Uppaal. The duration of the preempted task is decremented with the duration of the task that just finished executing. Following this, the model chooses a new set of tasks to execute. As with the non-preemptive templates, the Uppaal Cora chain template differs by edges being controllable, having no exponential rate, and adding the `cost'` variable.

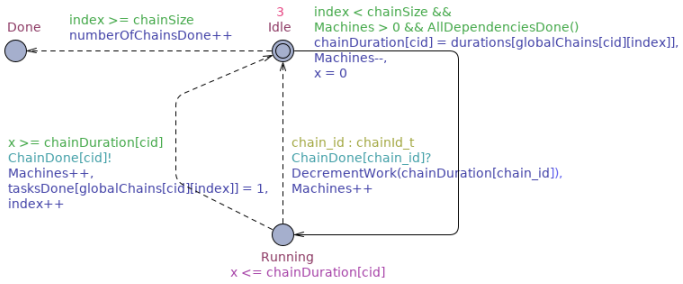


Fig. 4: Uppaal Stratego Preemptive Chain model

4.3 Correct Schedules

We argue that the schedules computed using our UPPAAL models fulfill the conditions of Definition 2. To fulfill Condition 1 from Definition 2 we store if a task is computed using an array of booleans. We uphold the condition in the `Chain` model on the guard of the transition from `Idle` to `Running`. `AllDependenciesDone` computes and-operations on the booleans in the array of computed tasks that correspond to the indices of the task dependencies, resulting in *true* if the task can be scheduled. To ensure Condition 2 from Definition 2 the `Machines` counter represents the number of available machines, initially being the total number of machines. `Machines` increments when a task is computed or preempted and decrements when a task is scheduled. The guard on the transition from `Idle` to `Running` requires that `Machines` must be positive, such that there is a machine available. Our models fulfill Condition 3 from Definition 2, as each task is executed in exactly one chain and a chain can only compute one task at a time; the first task in the chain. Lastly,

Condition 4 from Definition 2 ensures that all tasks are fully computed. As our models only compute the first task in the chain, this fulfills the condition for all except the last task in each chain. For the last task, the condition is fulfilled by the guard from `Idle` to `Done`, where `index` must be greater than or equal to `chainSize`. As the index variable is only incremented when a task is complete, all tasks of the chain have been computed when it is equal to the chain size. Additionally, no task can be scheduled for longer than its duration because of the invariant in the `Running` state.

5 Experimental Evaluation

In this section we evaluate our approach with an open data set containing a large number of task graphs. We evaluate our models with and without preemption, with and without chains. We compare our results with an state of the art tool.

5.1 Task Graph Set

We use a standard task graph set of Kasahara and Narita [8], which contain task graphs of sizes from 50 to 5000 tasks, and for each task graph size the set contains 180 different task graphs. For each task graph, we give the shortest obtained schedule length for 2, 4, 8, and 16 machines. The shortest schedules are obtained from Kasahara and Narita [8] and are without preemption. Thus, our preemptive runs *can* have a lower absolute optimum. We test our models on all 180 different task graphs for 50, 100, and 300 tasks, and extend the tests for Uppaal Cora with the sizes 500, 750, and 1000. We test each individual task graph with 2, 4, 8, and 16 machines both preemptively and non-preemptively in both Uppaal Cora and Uppaal Stratego. In Uppaal Cora, we use the branch and bound algorithm for each task graph and number of machines. In each instance, we conduct 100 runs with different pseudo-random seeds. To obtain near-optimal solutions, we use the random optimal depth-first search as the choice of branching. In Uppaal Stratego, a strategy is learned and the task graph is then simulated 2000 times under that strategy. All schedules and their respective execution times are available on GitHub[1].

[1] `https://github.com/marmux/spreadsheets`

5.2 Experimental Setup

We compare our schedules with the algorithm used by Kasahara and Narita [8], it is a parallel depth first/implicit heuristic search algorithm [9]. We run each of our experiments on a single core of an AMD Opteron 6376 Processor. Aalborg University's MCCAAU cluster [2], We are running Ubuntu 14.04.2 LTS with Uppaal Cora 32bit, and Uppaal Stratego 4.1.20-stratego-5.

To determine the quality of our resulting schedules, we compare these to the best schedules from Kasahara and Narita [8]. We present this in Table 1 and Table 2. For Table 2 the task size ranges from 50 to 300 tasks, as with more Uppaal Stratego consumes over 64 GB RAM, which we deem too much for the experiments to continue. When computing the Uppaal Stratego preemptive models with 300 tasks, 27.6% of the runs exceeded our memory limit. For the results in Table 1 the task size ranges from 50 to 1000. Uppaal Cora can at most use 4 GB of RAM (32bit implementation), and we found that task graphs with over 360 chains experience memory overflow, especially with more than four machines. With 500 tasks the preemptive Uppaal Cora model runs out of memory in 1.24% of the runs. For models with 750 tasks the preemptive Uppaal Cora runs out of memory in 5.08% of the runs and the non-preemptive model runs out of memory in 0.840% of the runs. Lastly, the task graphs with 1000 tasks, the memory overflow occurred in 11.5% of our preemptive runs and 1.37% for our non-preemptive runs.

5.3 Discussion

In this section, we explain and discuss the results obtained in Table 1 and Table 2. Because of the large number of data points for each of the task graph sizes, we have grouped all the data points by these sizes. Thus, we assume that task graphs of the same size are equally challenging to find schedules for. This is not necessarily true, as the state space of our models grows with the number of chains rather than tasks. However, to express as much of the data as possible, we use the minimum, quartiles, and the maximum. These give a good notion of the distribution of the values. Furthermore, we remove the outliers, which in most cases removes the most radical one or two percent. The only issue with removing the

[2] `https://sites.google.com/site/mccaau/`

Table 1: Aggregated results of UPPAAL CORA. ***Size*** is the number of tasks in the task graphs being compared. ***Mach.*** is the number of machines used for scheduling. Following is the percentage deviation between our schedules and the shortest-known schedules (from [8]); minimum length (`Min`), first quartile (`Q1`), median (`Q2`), third quartile (`Q3`), and maximum length (`Max`) of all the runs on the given task size and number of machines. Note, negative numbers denote instances where our schedules are shorter than those of [8]. We remove outliers greater than $Q3 + (Q3 - Q1) \cdot 2$ or less than $Q1 - (Q3 - Q1) \cdot 2$ to ensure an even distribution of data entries between the min, max, and quartiles.

		Cora Preemptive					***Cora Non-preemptive***				
Size	***Mach.***	***Min***	***Q1***	***Q2***	***Q3***	***Max***	***Min***	***Q1***	***Q2***	***Q3***	***Max***
50	2	-0.823	0.741	2.098	4.599	12.308	0.0	0.784	2.193	4.598	12.222
	4	0.0	2.74	7.227	13.483	34.951	0.0	2.74	7.558	13.208	33.981
	8	-4.545	0.0	3.077	11.429	34.286	0.0	0.0	2.222	11.429	34.0
	16	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
100	2	0.0	0.432	1.228	3.372	9.227	0.0	0.541	1.345	3.303	8.824
	4	0.0	2.308	4.803	8.917	22.059	0.0	2.21	4.478	8.333	20.513
	8	0.0	0.0	4.706	11.765	35.088	0.0	0.0	3.141	10.145	30.435
	16	0.0	0.0	0.0	1.709	5.128	0.0	0.0	0.0	0.0	0.0
300	2	0.0	0.166	0.414	1.034	2.766	0.0	0.193	0.473	1.079	2.851
	4	0.0	1.928	3.5	6.601	15.944	0.0	1.311	2.597	5.413	13.615
	8	0.0	2.586	8.28	12.694	32.642	0.0	1.389	5.163	9.048	24.227
	16	0.0	0.0	1.592	6.829	20.488	0.0	0.0	0.0	0.866	2.571
500	2	0.0	0.077	0.224	0.571	1.555	0.0	0.114	0.259	0.578	1.5
	4	0.0	1.408	2.395	4.888	11.835	0.0	0.73	1.42	3.29	8.41
	8	0.0	4.502	8.644	12.352	28.008	0.0	1.791	4.159	7.692	19.336
	16	0.0	0.548	3.367	12.874	37.5	0.0	0.0	0.0	3.917	11.747
750	2	0.0	0.065	0.154	0.434	1.172	0.0	0.075	0.188	0.441	1.17
	4	0.05	1.174	2.034	4.277	10.475	0.0	0.486	0.952	2.748	7.271
	8	0.0	4.042	7.747	11.773	27.199	0.0	1.184	2.609	5.714	14.774
	16	0.0	0.748	3.674	17.401	50.685	0.0	0.0	0.559	5.93	17.742
1000	2	0.0	0.05	0.138	0.428	1.183	0.0	0.058	0.149	0.393	1.059
	4	0.05	1.162	1.956	4.983	12.605	0.0	0.374	0.765	3.152	8.705
	8	0.0	3.716	7.552	11.111	25.852	0.0	0.888	1.917	4.343	11.228
	16	0.0	0.534	2.402	16.854	49.43	0.0	0.0	1.182	5.814	17.431

outliers occurs with the models where the first and the third quartiles are equal. Here, we remove a larger number (10-15%). However, as all the remaining data values (85-90%) have the same value, we do not think this issue is significant. The results of Table 1 and Table 2 contain a noticeable number of schedules that are as short as the shortest-known [8]. This is because the optimal non-preemptive schedule is found in both cases. Ad-

Table 2: Aggregated results of Stratego. For an explanation of the contents, see Table 1

		Stratego Preemptive					***Stratego Non-preemptive***				
Size	***Mach.***	***Min***	***Q1***	***Q2***	***Q3***	***Max***	***Min***	***Q1***	***Q2***	***Q3***	***Max***
50	2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.727	2.174
	4	-2.609	0.0	1.02	4.412	13.235	-2.797	0.0	2.098	4.505	13.514
	8	-8.791	0.0	0.0	4.651	13.953	-2.326	0.0	0.0	2.083	6.25
	16	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
100	2	-0.654	0.0	0.0	0.364	1.091	-0.24	0.0	0.321	0.743	2.23
	4	-0.508	1.042	2.817	5.224	13.455	0.0	1.071	2.655	4.592	11.628
	8	-2.797	0.0	2.273	8.081	24.242	0.0	0.0	0.4	5.983	17.949
	16	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
300	2	0.0	0.12	0.321	0.691	1.833	0.0	0.083	0.234	0.476	1.26
	4	-0.127	1.099	2.228	4.674	11.808	-0.127	1.027	1.921	3.3	7.843
	8	0.0	0.14	2.367	7.368	21.649	0.0	0.287	4.091	7.212	20.915
	16	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.218	0.654

ditionally, for each task size, a minimum schedule length is obtained that is better than or equal to those obtained in Kasahara and Narita [8]. Looking at the tables, Uppaal Stratego generally obtains better schedules than Uppaal Cora. In Table 2, we see that the preemptive models perform better than the non-preemptive ones when considering the minimum schedules. Furthermore, the preemptive models achieve the highest maximum values. Scheduling preemptively leads to a much larger state space, as many additional choices are available at any point when scheduling. However, the overhead of the increased state space does evidently not compromise the power of preemption to a significant extent, especially when considering the minimum or the small task graphs. Looking at the median values, the preemptive models generally perform slightly worse than the non-preemptive, especially in the cases with many tasks.

Considering Figure 5 we see that the most populated areas of the graph are on $y = 0$, meaning that our schedules match the known best schedules. Furthermore, 93% of the entries where the deviation is not 0%, is between 0% and 10% worse. Below zero percent we see that some schedules found were better than those found by Kasahara and Narita [8]. Although there are schedules up to 60% worse, the area above 10% is scarcely populated considering the number of entries in the graphs. The large deviations only occur on the shorter schedules, especially with a large number of machines.

As we mention to in Section 5, all the results contain an execution time alongside the obtained schedule lengths. The execution times show that Uppaal Cora is significantly faster than Uppaal Stratego. Additionally, it is possible to run a single Uppaal Cora run, which provides a near-optimal schedule. This is not possible when utilising machine learning in Uppaal Stratego, as Uppaal Stratego must make a strategy before finding schedules. This is what most of the execution time of Uppaal Stratego is spent on. However, if the problem at hand is static enough to reuse a strategy in Uppaal Stratego, then the execution time is significantly reduced. Additionally, Uppaal Cora is limited to 4GB of RAM, whereas Uppaal Stratego requires more RAM to compute the larger task graphs. Thus, which model to use varies on the application.

Furthermore, we timed the execution of all models with and without chains. Figure 6 shows the execution time between modelling the task graphs using chains and modelling all tasks individually on the task size 300, with 16 machines for Uppaal Cora non-preemptive. On the figure, the red circles show the execution time when modelling tasks individually, while green squares are runs where chains are modelled. Note, that some of the results are not shown as they are slower than 25 seconds. This is done to highlight the difference in execution time. The only runs that were slower than 25 seconds were achieved when modelling tasks rather than chains. Generally the execution time of Uppaal Stratego is ≈ 1.5 times faster when using chains as compared to not using chains. For Uppaal Cora the is more substantial, as seen on Figure 6, the execution time is ≈ 8 times faster when using chains than without chains. This is highly generalised, where smaller task graphs have smaller execution time differences, than bigger task graphs.

In Figure 5 the minimum schedules found for 50 tasks for each model and number of machines are seen. We have made one graph for each task size, and they are available on GitHub. As only Uppaal Cora has run the task graphs of size 500 to 1000, these graphs *only* contain the 1440 data entries, while the other three contain twice that. In contrast to Table 1 and 2 where we remove outliers, we do not remove them in the plots. Thus, the results may seem to deviate. We choose not to remove outliers in the plots, as we base these solely on the minimum schedules obtained for each task graph and machine number.

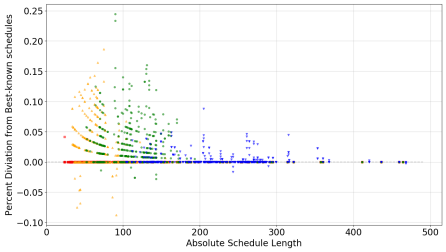


Fig. 5: This plot aggregates the minimum schedules found for 50 tasks for each model and number of machines. The colours and shapes separate results for 2, 4, 8, and 16 machines; blue triangles pointing down are 2, green circles are 4, orange triangles pointing up are 8, and red squares are 16. The x-axis is the length of the schedules we obtained, and the y-axis shows the relative deviation from the best-known schedules of Kasahara and Narita [8].

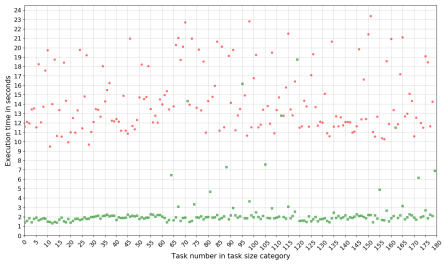


Fig. 6: Execution time for non-preemptive Uppaal Cora on the task-graphs of size 300, with 16 machines (Green squares represent runs with chains, red circles represent runs without chains)

6 Conclusions and Future Work

Task graph scheduling is a relevant problem in computer science with application in diverse domains including production lines, spreadsheets, etc. In this paper we present a methodology for finding near-optimal schedulers for task graphs. We consider preemptive and non-preemptive schedules. We model task graphs in the theories of priced timed automata and priced timed markov decision processes. Our implementation uses Uppaal Cora and Uppaal Stratego to compute the schedulers. We have compared our results with an state of the art tool [8]. Our experiments are encouraging and show that in most models we perform better

than [8]. We also explored the effect of using chains, which proved to be beneficial.

Future work include syntactic optimizations on our models, for example heuristics from [8] could be adopted to improve performance. Other future work include techniques to avoid the state explosion problem explored. Partial Order Reduction (POR) has been recently successfully applied to timed systems [5]. Application of POR for timed systems with costs could greatly improve the computation of schedulers for task graphs.

References

1. Abdeddaïm, Y., Asarin, E., Maler, O.: Scheduling with timed automata. Theoretical Computer Science **354**(2), 272–300 (Mar 2006). https://doi.org/10.1016/j.tcs.2005.11.018, http://dx.doi.org/10.1016/j.tcs.2005.11.018
2. Abdeddaïm, Y., Kerbaa, A., Maler, O.: Task graph scheduling using timed automata. Proceedings International Parallel and Distributed Processing Symposium p. 8 pp. (2003). https://doi.org/10.1109/IPDPS.2003.1213431
3. Aceto, L., Ingólfsdóttir, A., Larsen, K.G., Srba, J.: Reactive Systems: Modelling, Specification and Verification. Cambridge University Press (aug 2007), https://www.xarg.org/ref/a/0521875463/, iSBN: 1107410681
4. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126**(2), 183–235 (1994)
5. Boenneland, F., Jensen, P., Larsen, K., Muniz, M., Srba, J.: Start pruning when time gets urgent: Partial order reduction for timed systems. In: Proceedings of the 30th International Conference on Computer Aided Verification (CAV'18). LNCS, vol. 10981, pp. 527–546. Springer-Verlag (2018)
6. Bøgholm, T., Larsen, K.G., Muñiz, M., Thomsen, B., Thomsen, L.L.: Analyzing spreadsheets for parallel execution via model checking. In: Essays on the Occasion of Bernhard Steffen's 60th Birthday. LNCS, Springer, Heidelberg (2018)
7. David, A., Jensen, P.G., Larsen, K.G., Legay, A., Lime, D., Sørensen, M.G., Taankvist, J.H.: On time with minimal expected cost! In: Cassez, F., Raskin, J.F. (eds.) Automated Technology for Verification and Analysis. pp. 129–145. Springer International Publishing, Cham (2014)
8. Kasahara, H.: Standard task graph set, http://www.kasahara.cs.waseda.ac.jp/schedule/, accessed: 2019-05-01
9. Kasahara, H., Itoh, A., Tanaka, H., Itoh, K.: Parallel optimization algorithm for minimum execution-time multiprocessor scheduling problem. Systems and Computers in Japan **23**(13), 54–65 (1992)
10. Kwok, Y.K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Comput. Surv. **31**(4), 406–471 (Dec 1999). https://doi.org/10.1145/344588.344618, http://doi.acm.org/10.1145/344588.344618
11. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley & Sons, Inc., New York, NY, USA, 1st edn. (1994)
12. Terndrup, K.K., Sillesen, S.V.: Learn Smarter, Not Harder: Improving Uppaal Stratego through Preprocessing. Master's thesis, Aalborg University (5 2018)

A Proofs

Proof (Lemma 1). To prove Lemma 1, we show that a task is a least element of a chain at any point in time where it is scheduled. This proves Lemma 1 as all tasks that can be scheduled at any point in time - meaning that their dependencies are fulfilled - will always be a least element of a chain at that point in time. Thus, at any point in time the set of all tasks that can be scheduled is a subset of the set of least elements in all chains. We show this using the conditions of Definition 2.

Assume we have a task graph ($\mathcal{P}$, $\sqsubset$, D), a possible schedule *start*, and a chain decomposition: $C_1, C_2, ..., C_k$ where $1 \leq k \leq |\mathcal{P}|$. We denote the set of least elements in the chain decomposition at time t as $F = \{f \mid f \in C_i \setminus A \; for \; 1 \leq i \leq k, \; where \; \forall x \in C_i \setminus A \; we \; have \; f \sqsubset x \; or \; f = x\}$. Here A is defined as the completed tasks at time t;

$$A = \{P \; \in \mathcal{P} | \sum_{\{(s',d') | (s',d') \in start(P) \; and \; s'+d' \leq t\}} d' = D(P)\} \; . \tag{1}$$

Thus, to prove Lemma 1 we need to show that; for any $P \in \mathcal{P}$ and for any $(s, d) \in start(P)$ then $P \in F$ at time s. We define the dependencies of P as

$$Dep(P) = \{P' \in \mathcal{P} | P' \sqsubset P\} \; . \tag{2}$$

From Condition 1 of Definition 2 we know that

$$\forall P' \in Dep(P) \; s \geq sup(\{s' + d' \mid (s', d') \in start(P')\}) \; . \tag{3}$$

Recall, Condition 4 of Definition 2, which states that the sum of all intermediate durations must be the sum of the full duration in a possible schedule. As the supremum of $s' + d' \in start(P')$ is less than or equal to s, as stated by equation 3, then

$$\forall P' \in Dep(P) \sum_{\{(s',d') | (s',d') \in start(P') \; and \; s'+d' \leq s\}} d' = D(P') \; . \tag{4}$$

Thus, $Dep(P) \subseteq A$. As every dependency of P is in A, then P cannot be dependent on any task in C_i. Furthermore, $P \in A$ can only happen in the case where P has finished computing and $d = 0$. This is not possible as $d \in \mathbb{R}_{>0}$, according to Definition 2. Thus, $P \in F$.

B Algorithm for Chain Decomposition

The algorithm consists of two functions. The function `ChainCover` first sorts the task graph into a topologically sorted list. Then the function creates chains by calling `Visit` until the sorted list is empty. `Visit` removes the input node from the sorted list and calls itself recursively on the successor with the smallest number of predecessors to follow the heuristic. When a node is added to a chain all of its successors will have their predecessor count decremented. This is done until the input node has no successors and the entire chain is returned to the `ChainCover` function. Once the sorted list is empty, then `ChainCover` returns the list of chains. As `Visit` is called exactly $|V|$ times and each call conciders at most $|V|$ successors, then the complexity of `ChainCover` is $O(|V|^2)$.

Algorithm 1 Custom algorithm for computing the chain decomposition of a task graph

Require: A graph G, given by a list of edges E, and a list of vertices V
Ensure: A chain decomposition of the graph, given as a list of lists of vertices

```
function ChainCover(G(E, V))
    SortedList ← Topologically sorted G
    ChainsList ← NIL
    for all Edges in E do
        Increment predecessorValue of destination vertex
        Add destination vertex to the successors list of the source vertex
    end for
    while SortedList is not empty do
        Add chain from Visit function to ChainsList, using the first node from SortedList and the list itself as input.
    end while
return ChainsList
end function

function Visit(node,SortedList)
    Remove node from SortedList
    Decrement predecessorValue of all successors of node
    nextNode ← The successor with the smallest predecessorValue
    if nextNode is NIL then
        return node
    else
        return (node,visit(nextNode,SortedList)
    end if
end function
```