The Complexity of Stability

In Parallel Processor Scheduling

by

M. van Elsas

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday June 25, 2019 at 14:30 PM.

Student number:	4176898	
Project duration:	April 26, 2018 – June 25, 2019	
Thesis committee:	Dr. M.M. de Weerdt,	EEMCS TU Delft, supervisor
	Dr. N. Yorke-Smith,	EEMCS TU Delft
	Dr. B. Atasoy,	3ME TU Delft

TU Delft

Preface

This work is the final requirement for my master's degree. A journey that has taught me not just about computer science but about life as well.

First, I want to express my gratitude to Mathijs de Weerdt for his help and guidance. In particular the weekly discussions, mostly on this work but also on science in general.

In addition, I would like to thank my friends and family for their support. In particular, my parents for their unwavering support, that did not vary with their understanding of my work.

M. van Elsas
Delft, June 2019

Contents

1 Introduction

In the modern world, most events are planned in advance. Unfortunately things do not always go according to plan, as such altering our schedules occurs frequently. Ensuring that after the alteration we are left with an efficient plan that is similar to our original plan is often an intuitive goal.

In the remainder of this chapter we will first introduce some scenarios in which it is relevant that the schedule remains largely the same in Section 1.1. Subsequently, in Section 1.2 different environments in which uncertainty can be considered are introduced. Section 1.3 describes different ways in which the degree to which schedules differ can be measured. Finally, methods with which efficient schedules can be created in an uncertain environment are given in Section 1.4.

1.1. Motivation

When a person is performing task, it seems reasonable to assume they would rather stick to the initial schedule they have been given. This would seem particularly so if they use the same schedule multiple times and/or if the tasks are short.

When considering a medical appointment, keeping the same assignment is important. Once a schedule has been created, a patient has been scheduled to see a certain doctor and probably would rather not switch. Additionally, patients will likely want their appointment not to move too much. Being on-site earlier can only be done to a certain extent and having to wait is tiresome and can interfere with later appointments.

In the context of parcel delivery, the van that is being used is route dependent until it has been loaded. From that point, there is overhead to using a different van. For instance because computing the optimization problem including parcel transfers between vans is time consuming as is the physical act of transferring parcels. Because of this, we want parcels not to switch between the delivery vans and the vans are independent. While the delivery times are free during the initial optimization, they no longer are after they have been communicated to the destinations. From this point, we would like to have a delivery within the specified times-slot.

When scheduling combined cab rides, for instance those subsidised by the Dutch government for people who are not mobile (Valys). A large national party will subcontract these rides to smaller local cab companies. If the ride is altered – either the time or the location – this requires contacting the subcontractor, resulting in overhead for every change.

1.2. Scheduling and Uncertainty

In this section we consider scheduling in an uncertain environment. This uncertainty manifests itself in a variety of different ways. For example, a task takes longer than expected, a resource breaks down, a task is added during execution or a resource is not yet available when the task is supposed to start. An issue in many real-world environments is that insufficient reliable data is available about the distribution of these disturbances.

In the remainder of this section we consider schedules that do not change during execution in Subsection

1.2.1 and those that do in Subsection 1.2.2.

1.2.1. Static Scheduling and Uncertainty

In a static scheduling problem, we create a schedule that assigns tasks to resources in such a way that an objective is minimized. This objective is usually some function of the start times (e.g. makespan or maximum tardiness). A static scheduling environment assumes all information about the tasks and resources is known beforehand and that no adjustments are made during execution.

Unfortunately, there usually is uncertainty about execution in reality. Trying to create a single schedule that does not change to accommodate the myriad of potential real-world failures is unmanageable, we cannot take all of them into account. In practical situations we cannot even foresee all of them. Controlling for all those we can is usually impossible due to the constraints such as temporal constraints and the number of available resources. When not impossible, the resulting static schedule for all but the least error-prone environments would be very inefficient.

1.2.2. Dynamic Scheduling with Uncertainty

In dynamic scheduling, a situation is considered in which some of the information about the scheduling problem is learnt during the schedule's execution. When adjustments are made during execution there are two main approaches. The first is to fully define a schedule based on likely values and alter it when new information is learnt. The second to only partially define the schedule beforehand, and set the exact assignment and start time during execution[6, 40]. For instance by fixing the order of the tasks beforehand so that if executed in this order they meet the problem constraints and then to assign them to the first available resource during execution more sophisticated execution strategies exist[18]. This particular example only takes certain kinds of disruptions into account.

While leaving many scheduling aspects undefined until execution can work well in e.g. manufacturing environments in which the available resources are fixed. Different scenarios mentioned above benefit from having a fixed assignment to a fixed resource and time. For instance an appointment at a hospital, a time and which doctor to see would be conventional. Other examples would be a window of time in which a parcel may arrive and the time at which a subcontractor performs a certain task so that they can schedule other tasks to perform on the same day.

The above examples can be modelled as scheduling problems of varying degrees of complexity, this work limits itself to the identical parallel machine environment [26]. This choice is due to the relative simplicity of the environment while still containing the intricacy of having multiple resources. As tasks can rarely be interrupted and resumed in realistic scenarios involving people, pre-emption will not be allowed. Lastly, we will focus on fully defined schedules to enable giving fixed times and allocations as detailed above.

1.3. Types of Objectives for Scheduling with Uncertainty

When scheduling in a dynamic environment we distinguish two types of objectives. The performance objectives that are unrelated to the way in which the schedule has been changed i.e. it does not matter how different the executed schedule is from the initial one. For example, the makespan or mean flow time. Then we have the stability objectives which are related to how the schedule has been changed. A common example in literature is the completion time variance (CTV) [12, 33]. The difference in the completion time of all jobs in the initial (base-line) schedule and the realized schedule is measured. Such a metric is particularly meaningful for instance in a manufacturing environment when delivery times are agreed and altering them later brings cost due to storage or late fees.

As outlined in section 1.1, taking the allocation into account is often important. In addition, when the resources are human we also need to consider the stability from a resource perspective. That is, take into account the sequence of tasks scheduled on each resource i.e. the tasks assigned to each person.

1.4. Creating Schedules in a Dynamic Environment

The steps taken when creating a schedule in a dynamic environment are generally to create a base-line (initial) schedule and then to repair this when disruptions make the execution of this schedule impossible. As illustrated in figure 1.1, the proactive scheduler creates the first schedule from the problem instance. The objective of this step is to create a schedule that is likely to perform well with regard to the performance and stability objectives. This is highly dependent on the events that can take place in the environment and the policy employed by the reactive scheduler. The reactive scheduler repairs the schedule when a disruption occurs and can be non-existent (the base-line schedule must be possible regardless of what happens in the environment) or vary from a simple repair heuristic (e.g. right shift rescheduling[1]) to a complete reschedule.

The expectation of the performance and stability objective given the reactive policy and environment is the robustness of the schedule. The robustness should not only be taken into account in the proactive scheduler but also in the reactive scheduler.

An added degree of complexity is that we do not immediately have to communicate changes the moment a disruption happens if executing the current schedule is still possible for some time. For example if the duration of a task that is being executed increases, during that change the tasks on the other resources may not have to change. There is a trade-off between informing the resources and tasks in advance versus having inform them multiple times if more disruptions occur during the time the schedule remains the same. In addition, we can consider the communicated schedules those to which our stability objective applies. After all, these are the schedules that are considered by our tasks and resources. As the difference may be measured as the sum of the changes between consecutively communicated schedules, part of the reactive policy may be to keep the schedule identical for as long as possible to avoid having to communicate more often.

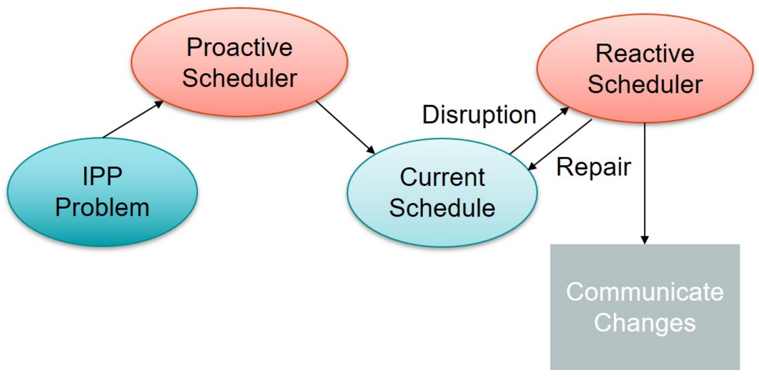

Figure 1.1: High level overview of proactive and reactive scheduling in a dynamic environment.

When the probability of another disruption approaches 0, the expected performance and stability approach the actual performance and stability. As such, when the probability distribution over all possible disruptions is part of the input, it must be at least as computationally hard as creating an optimally repaired schedule.

In practice, the repair has to be made quickly so that the repaired schedule can be executed without the need for a pause to finish the computation. To this effect, we consider weakly NP-complete problems as potentially viable because some pseudo-polynomial algorithms scale well in practice. On the other hand, due to the fact that strongly NP-complete problems can not have pseudo-polynomial algorithms or fully polynomial time approximation schemes[20] and we will consider finding a fast, exact and optimal repair strategy unlikely. The information about the disruption will be known from the moment the task starts. By applying only a single disruption, when to communicate does not have to be taken into account as there are only two schedules that must both be communicated. This document will investigate the viability of optimally repairing parallel processor schedules in an environment containing only a single disruption to the duration of a task.

2

Related Literature

In the previous chapter, we introduced the concept of performance and stability in a dynamic scheduling environment. In this chapter we investigate previous work in these fields. There are different dimensions to consider: We have proactive and reactive[41] scheduling, performance and stability objectives[24] , and different scheduling environments[5] . These environments may differ for example by the number of resources considered, the way in which tasks can be executed on specific resources, the sort of disruptions that can occur, and the likelihood of these disruptions occurring.

2.1. Partial Schedules

One option to avoid disruptions to postpone fixing scheduling details until execution[6, 40]. This enables dealing with disruptions as they occur. For more difficult scheduling problems, work done ahead of time enables quickly assigning tasks during execution. By not creating any expectations for a specific resource or time, there is no such thing as the stability of the realized schedule. On the other hand, the realized performance can be compared to the expected.

2.1.1. Temporal Networks

When considering partial schedules, they are often represented as a Simple Temporal Network (STN). A frequent property of this STN is that it only represents valid temporal assignments of the original scheduling problem. That is, all temporal assignments in the STN meet the other constraints (resource, precedence, etc.) of the scheduling problem. For example, optimizing the makespan in an identical parallel machine environment is strongly NP-complete, while it is possible to limit the solution space of the problem in order to be able to find a solution in polynomial time [40].

Such an STN will generally not contain all possible solutions to the problem. A measure that attempts to capture the intuitive property of how many solutions are contained is the flexibility[27, 50]. Brooks et al. [9] extend flexibility metrics to a robustness metric that takes the likelihood of certain events into account.

There are multiple extensions to STNs to include information about the uncertainty of dynamic scheduling. For instance in a STNU (uncertainty) we have a lower and upper bounds on time at which certain events occur. And in a PSTN (probabilistic) a probability density function over the times at which certain events occur.

When a STNU is dynamically controllable [10, 13, 37] it means that set of solutions it contains are realizable given any realization of the uncertainty. Rossi et al. [45] consider preferences for each of the time intervals between events. Of all legal schedules, they pick the one that optimizes these preferences.

Tsamardinos et al. [46] provide a method to compute lower- and upperbounds on the probability of a valid dynamic execution of an PSTN. Brooks et al. [9] call this probability the robustness of the PSTP. They experimentally determine this robustness based on simulations that sample the probability distribution.

2.2. Constraint Programming

Constraint programming enables solving a variety of scheduling problems. Such as RCPSP[17], course timetabling [7] or kernel resource feasibility [16]. Work has been done to create stable solutions to dynamic constraint programming problems (CSPs). An example is to keep the solutions similar in the sense that as few variables differ between them, known as minimum perturbation[34]. This approach has been combined with aiming for the minimization of the makespan[17]. However, in this particular example the method of backtracking from the original solution merely heuristically keeps the solutions similar. An approach that retains optimality with regard to the number of altered variables is suggested by Roos et al.[44]. They utilize a local search to systematically apply constraint propagation from the disrupted variable. First checking all options that could result in a valid solution with one variable change, then two, etc. continuing until a valid solution is found. Due to the lack of performance of this method, a heuristic approach is suggested to limit the number of solutions that are considered[42]. By pruning the search space in this manner, optimality is lost. In solving a scheduling problem of university timetables Lindahl et al.[34] offer human schedulers different options to deal with a disruption. This disruption consists of a resource (classroom, lecturer, etc.) becoming available or unavailable. The actual computation is done by solving a mixed integer program with an increasing allowance for perturbation until an acceptable solution quality is reached.

In order to guarantee a disruption in the form of a negated variable can be corrected within a perpetuation bound, the concept of supermodels [21] was developed. Such a model allows for a fixed number of changes a in the environment to be recovered by altering a fixed number of values b in polynomial time. If both a and b are fixed numbers and the original problem is in NP, creating the supermodel is in NP[21]. This is the case as there is only a fixed number of the original problem instances that need to be considered. However, problems in NP can be too computationally intensive to solve in an acceptable amount of time. In addition, scaling in a and b is not polynomial. So for larger values it is likely to be infeasible to compute the model. Even if we only allow for a single disruption, the potential number of changes required for a repair scales with the number of tasks. A further generalization of the supermodel is to find all solutions furthest way or closest to the starting solution[22].

Further advances have been made to more efficiently find a single minimum perturbation solution. For instance by combining constraint optimization with constraint satisfaction techniques[55]. Even in the most recent developments[19], no problems larger than 50 variables are considered for comparison. For a scheduling problem, this is very little as a different variable is usually used for each possible start time for every task.

2.3. Performance Objective

A well-researched area of dynamic scheduling pertains to the optimization of the performance objective. Most often, this is the same objective that was used in the creation of the baseline schedule. By creating a robust schedule[23], it is possible to guarantee that no considered disruption alters the makespan past a given bound[36].

When considering the completion times of the tasks as the objective, pseudo-polynomial algorithms exist for certain special cases[52]. In scheduling RCPSP problems, Calhoun et al. [11] make the distinction between making changes to the schedule before execution of the schedule (re-planning) and during execution of the schedule (re-scheduling). They use a heuristic to create the initial schedule and then improve on it using tabu search. Fu et al.[18] also make use of local search in RCPSP problems, of particular relevance is that they utilise a STN to find the repaired schedule. For more information see for example Herroelen et al.[24].

2.4. Stability Objective

When considering stability objectives, we make a distinction between two types. Time based stability in which only the start (or completion) times of the tasks are taken into account and resource based stability in which the assignment between tasks and resources is taken into account.

.[38]

2.4.1. Time Based Stability

Considering only stability with regard to the completion times, completion time variation (CTV) is a frequent objective [12, 33, 47, 48, 51]. On a single machine, Wu et al. [51] consider the experimental difference between different heuristics and genetic algorithms on CTV as well as the order of the tasks. An interesting conclusion by Van de Vonder et al. [47] is that creating a robust schedule by adding buffers of idle time between tasks to protect the makespan against disruptions does not create stability in the sense of CTV whereas placing the buffers with the aim of the stability objective (CTV) does protect the makespan.

Another time based objective is to minimize the (weighted) tardiness of all tasks Du et al.[15] show this problem is NP-hard. However, a pseudo-polynomial algorithm does exist[30]. A robust schedule based on stochastic machine failures as disruption can be created heuristically [38].

Other stability objectives are generally considered together with a performance objective. Abumaizar et al. [1] measure the change in makespan, start times and sequence comparing right shift rescheduling to a complete reschedule of the remaining tasks. The reschedule has the makespan as the objective. In a single machine environment, Zhao and Tang [54] give a polynomial algorithm to minimize the sum of completion times and keep the sequence as similar as possible. Considering only whether the schedule has changed during execution [46] is also a stability objective. In effect, this considers only if there is a need to communicate a change.

2.4.2. Resource Based Stability

When considering which resource each task is assigned to, Alagoz et al. [2] consider the sum of completion times together with the assignment. This is under the assumption that each disruption has been recovered before the next occurs and that full information on the length of the disruption is available. Additionally, they consider a limited subset of machines on which each job can be executed. Liu et al. [35] consider the same objectives when additional work is found during execution of the schedule. They show that it is possible to perform this multi-objective optimization in polynomial time. Ozlen et al.[39] consider the same problem but in an environment where a machine is disrupted being unable to perform tasks for some time. By considering reactive scheduling policies ranging from right shift rescheduling to a full reschedule (makespan objective) Arnaout and Rabadi [3] measure the effect on the makespan, assignment and the time it takes for the schedule to be the same as the base-line schedule.

2.5. Complexity Theory

This section contains the most relevant aspects of complexity theory to this thesis. First we give a (very) brief refresher on NP-hardness followed by a short introduction on the concept of strong NP-hardness. We will consider two classes of problems P and NP [28]. The class P contains all problems we can decide in polynomial time. The class NP contains all problems for which we can verify the solution in polynomial time (given a certificate). There are certain problems for which we know that if we can solve them in polynomial time, we can solve any problem in NP in polynomial time. These problems are known as NP-hard. Not all NP-hard problems are in NP, those that are are referred to as NP complete. Whether $P = NP$ is an open problem in computer science. It is generally accepted that it is unlikely that $P = NP$, this is the assumption we will make in this work.

The concept of strong-NP hardness[20], follows from the fact that no fully polynomial time approximation scheme can exist for certain NP-hard problems. A pseudo-polynomial algorithm can be used to create such an approximation scheme[20]. An algorithm is pseudo-polynomial if it's runtime is a function of not just the length of the input, but also the size of the input. This implies no pseudo-polynomial algorithm can exist for strongly NP-hard problems. The strong-NP hardness property can be shown through a (pseudo)polynomial reduction to a known strongly-NP hard problem. Problems that are known to not be strongly NP-hard, are known as weakly NP-hard. One way of showing a problem is weakly NP-hard is through a pseudo-polynomial algorithm for this problem.

2.6. Conclusion

In this chapter we have given an overview of the literature related to stability in scheduling. For the single machine scheduling model, a substantial amount of work has been done on efficiently computing different stability metrics and performance metrics. However, this is the easiest scheduling problem we consider which hides much of the complexity of having multiple resources. In practice, most relevant scheduling problems include more than one resource.

For the much more general constraint programming problem, work has also been done on computing in particular solutions at minimum perturbation as well as for other stability metrics in combination with optimality objectives. However, we cannot compute optimal solutions efficiently. In fact, this problem is strongly NPC in a static context and approaches to creating repairs are often heuristical.

There is still a variety of scheduling problems that exists in between the two mentioned above. Work has been done on dynamic scheduling problems varying from identical parallel processors to RCPSP. However, work on stability when regarding these problems is sparse. Particularly when combined with an optimality objective. It is of great interest whether approaches exist for these problems that are both efficient and optimal so that we need not fall back on the more general constraint programming problem as this either offers no guarantees as to the solution quality or takes too long to make a fast repair. To this end, we consider the identical parallel processors with only an increase of duration to a single task. This model includes multiple resources and as such captures much of the complexity involved in practical schedule repair. In this context both stability and performance will be considered. In this document we study the feasibility of creating fast repairs in this context. Offering algorithms to show the feasibility of creating such a repair and proofs of complexity to show infeasibility.

3
Parallel Processor Scheduling

In this chapter we present metrics that capture stability in scheduling; the difference between the initial and realized schedule from the perspective of humans as resources or tasks. This occurs within the context of Parallel Processor Scheduling as this is the simplest model that captures the intricacies in using multiple resources. First, in Section 3.1 we formalize our scheduling environment and our disruption model. Finally, we introduce our metrics and describe how they capture stability in Section 3.2.

3.1. The Parallel Processor Scheduling Problem

In this section, we formally define our problem consisting of identical parallel processors. Furthermore, we state the way in which the single disruption to the duration is modelled. Only one disruption is considered to avoid the complications of communicating changes. The disruption is to the duration as this is a particularly common occurrence. Lastly, we give a more concise problem notation for easier reference in Chapter 4 where the complexity of these problems is analysed.

In Subsection 3.1.1 the scheduling environment and the requirements for a schedule to be valid are defined. Following this we introduce our disruption model in Subsection 3.1.2. Lastly, the notation for scheduling problems including stability is given in Subsection 3.1.3.

3.1.1. Scheduling Environment

Our environment is the identical parallel processor scheduling problem [20, 26]. The output is a schedule S so that all constraints are met. In this section we do not apply an optimization objective to S, only the requirements that must hold for any valid schedule.

The scheduling problem consists of a set $P = \{p_1, p_2, \ldots, p_m\}$ of m identical processors, a set of n tasks $T = \{t_1, t_2, \ldots, t_n\}$, and $d(t_i)$ ($\mathbb{Z}^+ \to \mathbb{Z}^+$) denotes the duration of task $t_i \in T$. Finally, we define a directed acyclic graph G determining the precedence constraint on the tasks in T, an edge from t_i to t_j indicating t_i must finish before t_j can start.

For a schedule S to be valid we require that each task $t_i \in T$ is assigned to one processor $p_j \in P$ for its duration $d(t_i)$. The function $a(S, t_i)$ ($\to \{1, 2, \ldots, m\}$) denotes that the task $t_i \in T$ is assigned to p_j in schedule S. No pre-emption is allowed, the tasks must continue until they are finished.

Additionally, we require that only one task is assigned to a processor at a time. Let $s(S, t_i)$ ($\to \mathbb{Z}^+$) denote the start time of task $t_i \in T$ in schedule S and $f(S, t_i)$ ($\to \mathbb{Z}^+$) denote the finish time. To ensure that no two tasks overlap we require that for each processor $p_j \in P$, given a set of assigned tasks $\beta(S, p_j) = \{t_i \in T : a(S, t_i) = j\}$, the following holds:

$$\forall_{t_i, t_k \in \beta(S, p_j) : t_i \neq t_k} s(S, t_i) \geq f(S, t_k) \vee s(S, t_k) \geq f(S, t_i)$$

Further, to meet the precedence constraints, it must hold that if an edge exists from t_i to t_j, $f(S, t_i) \leq s(S, t_j)$ for any valid schedule S. If in G each vertex has at most one incoming edge and at most one outgoing edge, we say that the precedence constraints are in the form of $chains$.

Lastly, we introduce notation to be used bellow. Let $L(S, p_j)$ denote the elements $t_i \in \beta(S, p_j)$ sorted by increasing $s(S, t_i)$. In other words, this is the sequence of tasks on processor p_j in the order that they are executed in S. $I(t_i, L(S, p_j))$ denotes the index (starting at 1) of t_i in $L(S, p_j)$, or -1 if $t_i \not\in \beta(S, p_j)$.

3.1.2. Disruption Model

In this subsection, our model of disruptions is introduced. While many different such models exist, we opt to investigate the simplest such model that is still realistic. A single disruption to the duration of a task. An increase in the duration as it is common and a single disruption to avoid the complication of when to communicate changes to the schedule.

When such a disruption occurs, both the baseline schedule S and the repaired schedule S' need to abide by the constraints outlined above in Subsection 3.1.1. Furthermore, there is a relation between S and S' due to part of the schedule already having been executed at the time of the disruption. Lastly, it is of particular relevance what sort of disruptions are taken into account.

In our disruption model, one task can be delayed. This delay is a finite increase to the duration of that task and is known from the moment the task starts. i.e. the delay of a single task $q \in T$, is known from time $s(S, q)$. Before the execution starts (and as such before the disruption occurs) S is fixed; both the start times and assignment can no longer be changed.

The function $d'(t_i)$ ($\mathbb{Z}^+ \rightarrow \mathbb{Z}^+$) denotes the duration of task $t_i \in T$ after the disruption.

The following applies to d':

$$d'(q) > d(q)$$

and

$$\forall_{t_i \in T \mid t_i \neq q}\ d'(t_i) = d(t_i)$$

S' denotes a valid repaired schedule, that abides by the original constraints for a valid schedule with an increased duration for one task. In addition the tasks that have already started can no longer be moved. So for each $t_i \in T | s(S, q) \geq s(S, t_i)$, it must hold that $s(S', t_i) = s(S, t_i)$ and $a(S', t_i) = a(S, t_i)$. The subset of T that contains all tasks t_i for which it does hold that $t_i \in T | s(S, q) < s(S, t_i)$ is denoted by T'. T' contains all tasks that need to be scheduled during the schedule repair. As we assume tasks that start at the time of the disruption can no longer be moved because it is too late for this, it seems logical to conclude that the tasks in T' must start after the disruption. Let a_j denote the time from which p_j is available to schedule the tasks in T', a_j is the maximum between $s(S, q) + 1$ and the time at which the last task on p_j outside T' completes.

3.1.3. Problem Notation

In order to more concisely convey which problem we are addressing, we extend the notation of Lawler et al. [31] to include disruptions. The extension consists of relating the objective to the baseline schedule S and using a_j to denote the time from which the machine is available as in Alagöz and Azizoğlu [2].

In this notation $\alpha|\beta|\gamma$ defines a class of scheduling problems where α specifies the machine environment, β the job characteristics and γ the optimality criterion.

Let a_j be a job characteristic which defines that we have release times for our processors, this occurs if we have a disruption as defined in section 3.1.2. In addition, a_j implies access to the baseline-schedule in creating the repairs. Note that this in and of itself does not relate the repaired schedule to the baseline schedule. In fact, when we consider (only) a performance objective (e.g. makespan) as optimality criterion, the criterion is not directly related to the baseline schedule.

However, if we consider a stability objective as part of the optimality criterion this does relate to the previous schedule. It is measured between S which is part of the input and S' which is part of the output. To capture this, a stability objective in the scheduling problem is passed the baseline schedule as an argument. For example when considering (only) the assignment objective as defined in section 3.2.1, $\gamma = A(S)$. The optimization problem is to keep the value as low as possible. The corresponding decision problem is whether or not a schedule S' exists such that the metric between S and S' remains below some fixed value. When considering performance objectives, we limit ourselves to instances in which the performance objective is not required to be improved in S'.

Lastly we abuse the notation by denoting the multi-objective optimization problem of our metric and the makespan by taking $\gamma = met(S), C_{max}$. Where met refers to any of the metrics in section 3.2. This corresponds to the decision problem of whether or not a solution exists that has both has a value of C_{max} below a fixed value D and the respective metric below some other fixed value. Alternative problem definitions exist for multi-objective optimization such as finding Pareto optimal solutions or first optimizing one objective and while maintaining the optimal value for it, optimizing the other. We opt for this formulation as the generality of being able to weigh the importance of stability and performance is relevant in practice and we forgo investigating Pareto optimality.

As an example, $P|a_j|A(S)$ denotes the problem of selecting a schedule without precedence constraints on identical parallel machines that is as similar as possible to schedule S with regard to the assignment. Our re-scheduling problem consists of the duration of the tasks d', the set of tasks T', and the baseline schedule S. Note that these can be inferred from the original duration of the tasks d, the original set of tasks T, S and the new duration of the delayed task $d'(q)$. The output is the repaired schedule S' for which the similarity with regard to the assignment is maximized.

3.2. Metrics

When disruptions occur, the aim is to pick a repaired schedule that is the most similar to the baseline schedule. We measure how similar two schedules S and S' are through a function that maps assignment schedules S and S' to a non-negative real number. We will refer to such a function as a metric. Each of this section's four subsections defines a metric, these all increase as the schedules are more similar.

3.2.1. Assignment

The number of tasks that are executed on the same processor in S and S' is the assignment metric. This metric has been used previously in literature for varying reasons[2, 3, 35, 39] such as the cost associated with changing resource allocation during the schedule's execution. We elect to use it as it captures the stability of the relation between an assigned task/resource pair. In our example, this corresponds to patients keeping the same doctor as detailed in section 1.1. This metric is the number of identical assignments.

We formally define the assignment metric as follows:

Definition 3.2.1.

$$A(S, S') = |\{i \in \{1, 2, \ldots, n\} : a(S, t_i) = a(S', t_i)\}|$$

Theorem 3.2.1. *$A(S, S')$ can be computed in polynomial time.*

proof. We loop over all tasks, aggregating the number for which the assignment remains the same. This results in computing $A(S, S')$ in linear time. □

3.2.2. Order

The amount the schedule of a resource changes is relevant as explained in section 1.1. The amount of change is captured by the degree to which the order of the tasks they must perform remains the same. We approach the order by taking the number of positions the task has moved in the sequence of tasks the resource must perform. This is similar to the number of steps between two rankings taken in Spearman's rank correlation [53].

When the second task of different processors switch assignment, their position in the sequence remains the same. However, this does introduce a substantial chance in the schedule. Because of this, we only consider the position in the sequence if the assignment of a task remains the same. If not, we consider this less similar than if the assignment had stayed the same. As such, if a task is no longer executed on the same processor, the penalty u is occurred.

We define our order metric as follows:

Definition 3.2.2.

$$O_u(S,S') = \frac{1}{\sum_{p_j \in P} \sum_{t_i \in \beta(S,p_j)} LS(t_i, p_j, S, S', u)}$$

Where the function LS is defined as follows:

$$LS(t_i, p_j, S, S', u) = \begin{cases} |I(t_i, L(S,p_j)) - I(t_i, L(S',p_j))| & I(t_i, L(S',p_j)) > 0 \\ u & I(t_i, L(S',p_j)) \leq 0 \end{cases}$$

Theorem 3.2.2. *For a value of $u > n^2$, maximizing $O(S,S')$, maximizes $A(S,S')$.*

proof. Let $\sum_{p_j \in P} \sum_{t_i \in \beta(S,p_j)} LS(t_i, p_j, S, S', u)$ be the penalty for changes to the order. By definition, minimizing this penalty maximizes our similarity O. Now, when u is larger than n^2, the penalty for altering the assignment of one task between S and S' is more than n^2. Because of the definition of LS, each tasks can contribute no more than n to the penalty if their position is changed by n. As there are only n tasks, the contribution of the alteration of the ordering by tasks that keep their assignment is bound by n^2. As such, maximizing O requires maximizing A. □

Theorem 3.2.3. *$O(S,S')$ can be computed in polynomial time.*

proof. We loop over all processors in S determining the position of each task, we then repeat this for S'. Then, we loop over all tasks and first compare their assignment in S and S', adding u to the total if they are different. If the assignment does not change, we aggregate the number of positions the task has moved. Finally, we compute the inverse to go from a distance to a similarity. As each of the sequential steps is done in linear time, this results in computing $O_u(S,S')$ in linear time. □

3.2.3. Timed assignment

The number of changes, relevant in for example the case of the subcontractors in section 1.1 is captured in the timed assignment metric. It is defined as the number of tasks that is still executed on the same processor at the same time in S'. This metric is similar to the minimum perturbation in constraint programming[34].

More formally the timed assignment metric is:

Definition 3.2.3.

$$TA(S,S') = |\{i \in \{1,2,\ldots,n\} : a(S,t_i) = a(S',t_i) \ \wedge \ s(S,t_i) = s(S',t_i)\}|$$

Theorem 3.2.4. *$TA(S,S')$ can be computed in polynomial time.*

proof. We loop over all tasks, aggregating the number for which both the assignment and start time does not change. This results in computing $TA(S,S')$ in linear time. □

3.2.4. Stable time window

A generalization of the timed assignment metric is the number of tasks that is executed on the same processor within a time window. This metric relates to the delivery windows when considering parcels as shown in Section 1.1. This time window is symmetrical and has an allowed offset of α on either side.

We define our stable time window metric as follows:

Definition 3.2.4.

$$STW_\alpha(S,S') = |\{i \in \{1,2,\ldots,n\} : a(S,t_i) = a(S',t_i) \ \wedge \ |s(S,t_i) \ - \ s(S',t_i)| \leq \alpha\}|$$

Note that for $\alpha = 0$ this is the timed assignment metric and for $\alpha > \sum_{t_i \in T} d(t_i)$ it is the assignment metric.

Theorem 3.2.5. *$STW(S,S')$ can be computed in polynomial time.*

proof. We loop over all tasks, aggregating the number for which the assignment does not change and the start time moves by at most α. This results in computing $TA(S,S')$ in linear time. □

4 Complexity

This chapter investigates the complexity of repairing a schedule so that it maximizes the similarity to the baseline schedule. This is of particular relevance as it is a large factor in determining if fast repairs are feasible. In this chapter we assume that $P \neq NP$, whether this is true remains one of the largest open problems in computer science. However, this is a generally held belief inside the research community. We consider problems that are weakly NP-complete (wNPC) potentially feasible as pseudo polynomial algorithms for problems such as partition scale very well, and those that are strongly NP-complete (sNPC) infeasible as no pseudo polynomial algorithm can exist.

In this chapter we first investigate the complexity of optimizing each metric in an environment with an unbounded number of processors. This both provides us with a lower-bound on the complexity of harder problems (precedence constraints, multi-objective) and is a relevant result in and of itself in case the stability is the sole consideration. In addition, we consider the complexity when precedence constraints are added. These are common in environments in practice. Lastly, we consider the multi-objective problem as a combination between the stability metric and the most prevalent performance objective: The makespan. Both in the context of an unbounded number of processors as is the most general scenario and in the context of only two processors to see if significant performance increases are possible on a bounded small number of processors.

This chapter is organized as follows: First the verification of the problems is considered in Section 4.1. Followed by the complexity of the performance objective makespan in Section 4.2. The results consisting of the complexity of each problem outlined above follow, generally categorized by stability metric. Lastly, we conclude this chapter in Section 4.7 giving an overview of- and reflection on the results. This chapter contains two proof concepts in Sections 4.5.2 and 4.5.3 which are used again in Chapter 5.

4.1. Polynomial-time Verification

In this subsection we consider the complexity of the verification of any of the problems in Table 4.1. S' can be considered as a certificate that shows a solution exists for a certain value. By Theorem 3.2.1, 3.2.3, 3.2.4, and 3.2.5 we can verify this certificate in polynomial time for each of our stability metrics. Computing the makespan of a schedule S' is known to be in P. Checking that S' meets the requirements outlined in Section 3.1.1 can also be done in polynomial time as can checking the requirements of our disruption model given in Section 3.1.2. As such, all problems contained in Table 4.1 are verifiable in polynomial time and are in NP.

4.2. Makespan

The first objective we consider is the performance objective makespan. This section shows the complexity of optimizing this objective in our disruption model. The aim of this section is to use the results bellow, as a lower bound on the complexity of multi-objective optimization problems including the makespan. The proofs contained in this section are relatively simple as they reduce to the same problem without a disruption.

When considering the makespan in an optimization problem the aim is to minimize the latest finishing time of all tasks. When considering the corresponding decision problem, the question is whether a schedule exists of which the latest finishing time at most some deadline D.

We investigate the problem in the case of two processors ($P2|a_j|C_{max}$) and in the case of an unbounded number of processors ($P|a_j|C_{max}$).

4.2.1. Makespan on Two Parallel Processors: $P2|a_j|C_{max}$

Theorem 4.2.1. *$P2|a_j|C_{max}$ is NP-hard.*

We show Theorem 4.2.1 is true by showing the decision problem corresponding to $P2|a_j|C_{max}$ is NP-hard by means of a reduction from subset sum.

We use the following definition of Subset Sum:

Definition 4.2.1. Subset Sum: Given a set of elements $E = \{1,\dots,n\}$ with integer weights $w_1,\dots,w_n$ do the weights of any subset of E sum up to exactly W?

In the following proof we create a base-line schedule S so that a delay of length 1 forces swapping a task of duration $W+1$ for a set of tasks with durations equal to W. Because the durations of the tasks are equal to the weights in Subset Sum, if such a set exists we have a solution to Subset Sum. A sketch of S is given in Figure 4.1. The task that is delayed to ensure the move is q.

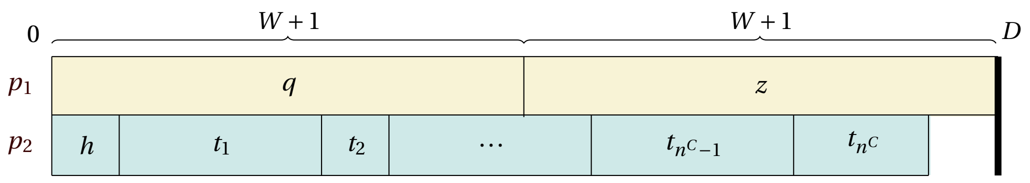

Figure 4.1: Sketch of S

Proof. Given a non-trivial instance E, W of subset sum, we create an instance of $P2|a_j|C_{max}$ as follows:

1. We create two processors p_1 and p_2.
2. We create two tasks to be scheduled on p_1 in S and $n+1$ tasks for p_2. The tasks for p_1 are q and z. For p_2 we have h and a task t_i for each element in E, the tasks corresponding to elements in E are contained in set E'.
3. The deadline D is set to $\sum_{i=1}^{n} w_i + 2$.
4. The duration for all tasks corresponding to the elements in E is equal to the weight of the element. $\forall_{e\in E} d(t_e) = w_e$
5. The duration of h is 1, $d(h) = 1$.
6. The duration of z is $W+1$, $d(z) = W+1$. This is the task that will be forced to change allocation.
7. The duration of q is set to fit with z on p_1 before D exactly, $d(q) = \sum_{i=1}^{n} w_i - W + 1$.
8. The tasks q and h start at time 0, $s(S,q) = 0$ and $s(S,h) = 0$.
9. The tasks in E' are scheduled to start after h, as soon as the previous task has finished. This means that the start times $s(S,z) = \sum_{i=1}^{n} w_i - W + 1$.
10. Our disruption to q is an increase in the duration of 1, $d'(q) = \sum_{i=1}^{n} w_i - W + 2$.

It is impossible to schedule z on p_1 in the repaired schedule S' without breaking the deadline. This is because the duration $d(q)$ is set in step 7 so that $d(q) + d(z) = D$. As such, there is no idle time on p_1 in S. Due to the delay of q in step 10, q and z cannot both be scheduled on p_1 in S', as q starts at the time of disruption it can not change allocation and z must.

There can be no idle time in a valid S'. We consider the idle time in S and the delay. Firstly, there is no idle time on p_1 in S as shown above. On p_2 the durations are chosen so that $d(h) + \sum_{t_i\in E'} d(t_i) = D-1$ in step 4 and 5. As the sum of the durations increases by 1 after the delay, there can be no idle time in S' assuming the deadline is met and all tasks are scheduled.

If a valid S' exists, we can create a subset of E of which the weights sum up to W. As there is no idle time, the durations of the tasks assigned to p_1 – excluding q – must sum up to W and the corresponding elements of E have weights that sum up to W.

If there is a subset of E of which the weights sum up to W, a valid S' exists. If there are elements in E of which the weights sum up to W, there are tasks in E' of which the durations sum up to W. As such, we can select these tasks and assign them to p_1, the remaining tasks can be assigned to p_2 as the sum of their durations $\sum_{t_i \in \beta(S',p_2)} d'(t_i)$ adds up to $\sum_{i=1}^{n} w_i + 1$.

As a valid S' exists iff there is a subset of E of which the weights sum up to W and the reduction is polynomial, $P2|a_j|C_{max}$ is NP-hard. □

4.2.2. Makespan on Parallel Processors: $P|a_j|C_{max}$

When instead of a fixed number of two processors we consider an unbounded number, the problem is strongly NP-hard.

Theorem 4.2.2. *$P|a_j|C_{max}$ is strongly NP-hard.*

We show the problem of $P|a_j|C_{max}$ is at least as hard as $P||C_{max}$ and as such strongly NP-hard[20] through a polynomial reduction from $P||C_{max}$ to $P|a_j|C_{max}$.

First we give a definition of the decision problem corresponding to $P||C_{max}$, that is minimizing the makespan in a parallel machine scheduling problem without precedence constraints:

Definition 4.2.2. Decision $P||C_{max}$: Given a set P^C of m^C identical processors and a set of n^C tasks T^C, a function d^C defining the duration of each task $t_i^C \in T^C$ and a deadline D^C ($\in \mathbb{Z}^+$). Here the superscript C is used to more easily distinguish the instance from that to which it is reduced.

Question: Does a schedule S^C exist such that the makespan of this schedule equal to or less than D^C?

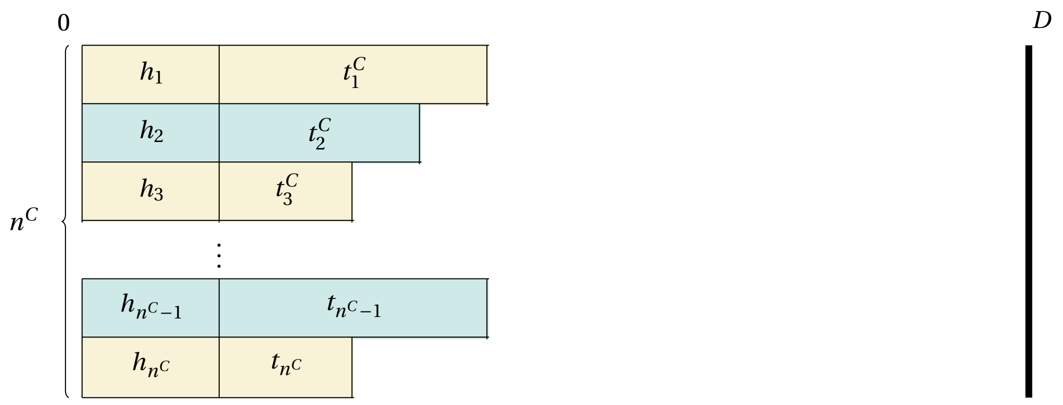

Figure 4.2: Created input schedule for $P|a_j|C_{max}$

.

The idea behind our proof below is to solve an instance of Decision $P||C_{max}$ by using a polynomial number of instances of $P|a_j|C_{max}$. The basic idea is to start by creating a schedule for the tasks in T^C between 1 and $D^C + 1$ on n^C processors, assigning 1 task to each processor. A task h_i is scheduled on each processor. Figure 4.2 shows a sketch of this configuration.

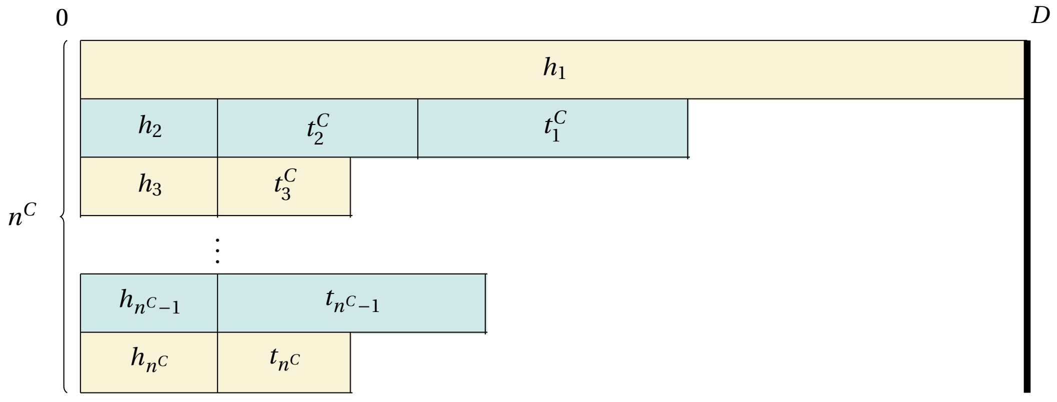

Figure 4.3: S' after one disruption, removing one processor from those available for the tasks in T^C.

We then reduce the amount of available processors by 1 through a delay to a task h_i. The tasks in T^C cannot be scheduled any later than $D^C + 1$ in the repaired schedule because this would be after D. In Figure 4.3 the delay is represented, the tasks in T^C are not.

We can then take our repaired schedule S' and use it as the baseline schedule S, restarting the execution and delaying the next task h_i in H, until we have m processors on which the original tasks T^C are scheduled.

Proof. Given an instance of Decision $P||C_{max}$ create an instance of $P|a_j|C_{max}$ in the following manner:

Initialization Phase

1. We create a set P of n^C $(= |T^C|)$ processors.
2. We create a set H of n^C tasks.
3. The duration of tasks in H is 1, i.e. $\forall_{h_i \in H}, d(h_i) = 1$.
4. We retain the duration of the tasks in T^C: $\forall_{t_i \in T^C}, d(t_i) = d^C(t_i)$

Using $T = H \cup T^C$ as the set of tasks, P as the set of processors d as durations $D = D^C + 1$ as deadline we create S:

5. All tasks in H start at time 0. $\forall_{h_i \in H}, s(S, h_i) = 0$.
6. Each task in H is assigned to a different processor. $\forall_{h_i \in H}, a(S, h_i) = i$
7. Start the tasks in T^C at 1 to avoid the tasks in H. $\forall_{t_i \in T^C} s(S, t_i) = 1$
8. Assign each task in T^C to a different processor. $\forall_{t_i \in T^C} a(S, t_i) = i$

Note that the tasks in H take place in the interval $[0, 1)$ and the tasks in T^C in the interval $[1, D^C + 1)$.

Iteration Phase

1. If $n^C - m^C$ tasks in H have a duration D, accept.
2. Delay a task $h_i \in H$ with $d(h_i) = 1$ by D^C, i.e. $d'(h_i) = D$.
3. We now solve the problem $P|a_j|C_{max}$, with P, T, S, D and d' as input and get a valid repaired schedule S' or a reject.
   (a) If a reject is returned, we reject too.
   (b) If we receive a valid S', we repeat the iteration phase with $S = S'$. Note that $\forall_{h_i \in H} s(S', h_i) = 0$ as they started at the time the disruption occurred.

Given a repaired schedule S' in which the tasks in T^C are scheduled between 1 and $D^C + 1$ we can create a schedule S^C in which only the tasks in T^C are scheduled between 0 and D^C on m^C processors. At least $n^C - m^C$ processors must contain only 1 task $h_i \in H$ as their duration is D and there is no other way to schedule them before D. This leaves m^C processors, each having a h_i scheduled on the interval [0,1). This means the tasks T^C are scheduled between 1 and D (= $D^C + 1$). As such, the assignment in S^C is identical to that in S', the start times are all decreased by 1.

Given S^C in which the tasks in T^C are scheduled between 0 and D^C we can create a schedule S' after $n^C - m^C$ disruptions, by delaying all tasks in T^C by one, and assigning the tasks assigned to a processor in S^C to a processor in S' of which the task h_i has not been delayed.

As such, a schedule S' exists after $n^C - m^C$ delays of this form iff S^C exists.

The initialization phase can be performed in polynomial time as can the iteration phase. The initialization is executed once, the iteration is executed at most $n^C - m^C$ times which is linear in the input. This makes the reduction polynomial time and as such $P|a_j|C_{max}$ must be strongly NP-hard as $P||C_{max}$ is strongly NP-hard[20]. □

4.3. Assignment

4.3.1. Order on Parallel Processors with Precedence Constraints: $P|a_j, prec|O(S)$

This problem is considered before those regarding the Assignment metric. This is due to it being more general and as such Corollary 4.3.1.1 and 4.3.1.2 follow from the following theorem:

Theorem 4.3.1. *$P|a_j, prec|O(S)$ is in P*

A polynomial time algorithm for the problem $P|a_j, prec|O(S)$ exists. Namely, Right-Shift Rescheduling (RSR)[1], the idea is to simply delay all tasks that are in some way dependent on the finishing time of q by the delay incurred by q i.e. shifting them right in the schedule. This always results in an identical order and assignment and as such an optimal solution with regard to O.

4.3.2. Assignment on Parallel Processors with Precedence Constraints: $P|a_j, prec|A(S)$

Corollary 4.3.1.1. *The problem $P|a_j, prec|A(S)$ is in P*

By Theorem 3.2.2 $P|a_j, prec|A(S)$ is a special case of $P|a_j, prec|O(S)$. As $P|a_j, prec|O(S)$ is in P by Theorem 4.3.1, it follows that $P|a_j, prec|A(S)$ is in P.

4.3.3. Assignment on Parallel Processors: $P|a_j|A(S)$

Corollary 4.3.1.2. *The problem $P|a_j|A(S)$ is in P*

Any instance of $P|a_j|A(S)$ is an instance of $P|a_j, prec|A(S)$ with no precedence constraints. As $P|a_j, prec|A(S)$ is in P by Corollary 4.3.1.1, $P|a_j|A(S)$ must be in P.

4.3.4. Assignment and Makespan on Two Parallel Processors: $P2|a_j|A(S), C_{max}$

Theorem 4.3.2. *$P2|a_j|A, C_{max}$ is weakly NP-complete.*

While for an unbounded number of processors this problem is sNPC, this does not imply there can be no (pseudo-)polynomial algorithm for a fixed number of processors. It follows from Theorem 4.2.1 that this problem is NP-hard as it is a more general case. In this section we give an optimal pseudo-polynomial algorithm.

Theorem 4.3.3. *A pseudo-polynomial algorithm exists for the problem $P2|a_j|A, C_{max}$.*

We show a pseudo-polynomial Dynamic Programming (DP) algorithm exist using Lemma 4.3.4 and Lemma 4.3.5. The combination of these lemmas shows an adaptation of a knapsack algorithm is correct for this problem.

Given an instance of $P2|a_j|A, C_{max}$, let the processor on which the delay occurs be p_1 and the other be p_2 without loss of generality. Let the idle time of a processor be the time before the deadline during which it is not assigned a task. Let the idle time of a schedule be the sum of the idle time over all processors.

The idle time in the repaired schedule S' is fixed for each problem instance. This is the case as by definition, idle time I is $F - \sum_{t_i \in T'} d'(t_i)$. Where T' is the set of tasks not yet started at the moment the disruption occurred. Note that F, the time available after the disruption to schedule the tasks in T' is fixed as

$F = 2D - a_1 - a_2$. The sum of the durations of the tasks in T', $\sum_{t_i \in T'} d'(t_i)$ is obviously fixed as well. We assume a non-trivial problem instance for which $F \geq \sum_{t_i \in T'} d'(t_i)$.

Lemma 4.3.4. *In any valid repaired schedule S' the latest completion time on p_1 after q is between $D - I$ and D.*

Proof. By definition, the idle time on p_1 in S' is no more than I. As there are no precedence constraints we can move the tasks on p_1 and p_2 in T' to start the moment the previous task is completed (or at a_j for the first task in T' scheduled on p_j). This results in no idle time between the tasks. If there is no idle time on p_2, the sum of the duration of tasks assigned to p_1 is $D - a_1 - I$, by not having any idle time the last completion time on p_1 is $D - I$. If p_2 has I idle time, the sum of the duration of tasks assigned to p_1 is $D - a_1$, and as such the latest completion time is D. In conclusion, the completion time on p_1 must be between $D - I$ and D. □

From our set of available tasks T', we can compute a subset of tasks (if one exists) that sums up to each value in the range between $D - a_1$ and $D - I - a_1$. This is the Subset Sum problem and pseudo-polynomial algorithms exist [29]. Note that any such assignment of tasks from T' to p_1, results in a valid S' as the remaining tasks will fit on p_2.

A generalization of the Subset Sum problem is the Knapsack problem in which it is not the weight of the subset that is maximized but the value of the elements.

Definition 4.3.1. Knapsack: Given a set of elements $E = \{1, \ldots, n\}$ with integer weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ what is the maximum sum of values of a subset of E of which the sum of the weights is at most W?

In order to maximize $A(S, S')$ we use the function $z(t_i)$, defined as follows:

$$z(t_i) = \begin{cases} 1 & \text{if } t_i \in \beta(S, p_1) \\ -1 & \text{if } t_i \in \beta(S, p_2) \end{cases}$$

$z(t_i) = 1$ if the tasks was scheduled on p_1 in S, and $b = -1$ if it was not (and as such was scheduled on p_2). Let K be a list containing all items.

Lemma 4.3.5. *Maximizing $\sum_{t_i \in \beta(S', p_1)} z(t_i)$ maximizes $A(S, S')$.*

Proof. By definition 3.2.1:

$$A(S, S') = |\{i \in \{1, 2, \ldots, n\} : a(S, t_i) = a(S', t_i)\}|$$

The maximum value of $A(S, S')$ is $n = |\beta(S, p_1)| + |\beta(S, p_2)|$. The actual value is this minus the tasks that were moved or $|\beta(S, p_1)| - |\beta(S, p_1) \cap \beta(S', p_2)| + |\beta(S, p_2)| - |\beta(S, p_2) \cap \beta(S', p_1)|$. $|\beta(S, p_1)| - |\beta(S, p_1) \cap \beta(S', p_2)| = \sum_{t_i \in \beta(S, p_1)} z(t_i)$ and $-|\beta(S, p_2) \cap \beta(S', p_1)| = \sum_{t_i \in \beta(S, p_2)} z(t_i)$. As $|\beta(S, p_2)|$ is constant, and we maximize the sum between the remaining positive and negative aspect, the lemma holds. □

We can now create an algorithm for $P2|a_j|A(S), C_{max}$ to show Theorem 4.3.3 holds.

Proof. To solve $P2|a_j|A(S), C_{max}$ we use a Dynamic Programming (DP) algorithm for Knapsack, choosing the duration as the primary objective and our assignment as the secondary objective. This is done by encoding both into the value attribute. Due to the nature of dynamic programming, we can compute the solution to all the sub problems in the range in Lemma 4.3.4 with little overhead.

The encoding of our problem as an instance of Knapsack is performed as follows: For each task t_i we create an item i with a weight w_i and a two-dimensional value v_i. The weight of the item is equal to $d'(t_i)$. The value v_i is denoted by a tuple (a, b) where a is also the duration $d'(t_i)$. And b is $z(t_i)$ where z is defined as follows:

The objective is to maximize the sum of the value of the subset we select while staying within the weight limit W. When comparing values, the first element (a) is dominant and the second (b) is a tiebreaker. Using a DP algorithm for knapsack, we get the possible tasks assignments for p_1 (those within the bounds of Lemma 4.3.4) and their corresponding sum of $z(t_i)$. We pick the one with the maximum value of $z(t_i)$, as this maximises $A(S, S')$ by Lemma 4.3.5.

As we pick the task assignment (from those that have valid lengths) to p_1 which maximizes $z(t_i)$, we maximize $A(S, S')$. As our algorithm for an NP-hard problem is pseudo-polynomial Theorem 4.3.2 must hold.

□

Corollary 4.3.5.1. *$P2|a_j|C_{max} is wNPC$.*

By Theorem 4.3.3, a pseudo-polynomial algorithm for the more general problem $P2|a_j|A(S), C_{max}$ exists. In addition Theorem 4.2.1 states that $P2|a_j|C_{max}$ is NP-hard. From this, it follows that $P2|a_j|C_{max}$ is weakly NP-complete.

4.3.5. Assignment and Makespan on Parallel Processors: $P|a_j|A(S), C_{max}$

Corollary 4.3.5.2. *$P|a_j|A(S), C_{max}$ is sNPC*

$P|a_j|A(S), C_{max}$ is sNPC as $P|a_j|A(S), C_{max}$ is at least as hard as $P|a_j|C_{max}$. Which is strongly NP-hard by Theorem 4.2.2.

4.4. Order

Order on Parallel Processors: $P|a_j|O(S)$

Corollary 4.4.0.1. *$P|a_j|O(S)$ is in P*

The problem $P|a_j|O(S)$ is a special case of $P|a_j, prec|O(S)$. As $P|a_j, prec|O(S)$ is in P by Theorem 4.3.1, $P|a_j|O(S)$ must be in P also.

4.4.1. Order and Makespan on Two Parallel Processors: $P2|a_j|O(S), C_{max}$

Corollary 4.4.0.2. *$P2|a_j|A(S), C_{max}$ is NPC*

This problem is at least as hard as $P2|a_j|C_{max}$ which by Theorem 4.2.1 is NP-hard.

4.4.2. Order and Makespan on Parallel Processors: $P|a_j|O(S), C_{max}$

Corollary 4.4.0.3. *$P|a_j|A(S), C_{max}$ is sNPC*

This problem is at least as hard as $P|a_j|C_{max}$ which by Theorem 4.2.2 is strongly NP-hard.

4.5. Timed Assignment

4.5.1. Timed Assignment on Parallel Processors: $P|a_j|TA(S)$

Theorem 4.5.1. *$P|a_j|TA(S)$ is in P*

First, we note that the processors $p_j \in P$ are independent with regards to TA. This is the case as we can always schedule tasks in S' after all tasks in S are done and there are no precedence constraints.

For all processors p_j for which $q \notin \beta(S, p_j)$ we can keep the start times and assignments from S.

We know that for all tasks t_i on the same processor as q in S that those for which $s(S, t_i) \leq s(S, q)$ can not be moved (and are optimal with regard to TA). For the remaining tasks it holds that all those that have a start time $s(S, t_i)$ that is less than $s(S, q) + d'(q)$ can never be scheduled at the same time. The tasks for which $s(S, t_i) \geq s(S, q) + d'(q)$ do not have to be moved.

So, by only moving the tasks t_i on the same processor as q for which $s(S, q) < s(S, t_i) < s(S, q) + d'(q)$ (e.g. by adding the makespan to the start time in S), we create an optimal schedule with regards to TA. This can obviously be done in polynomial time.

4.5.2. Timed Assignment on Parallel Processors with Chain Precedence Constraints: $P|a_j, chains|TA(S)$

Theorem 4.5.2. *$P|a_j, chains|TA(S)$ is strongly NP-hard*

We prove that $P|a_j, chains|TA(S)$ is strongly NP-hard by showing a polynomial time reduction from the decision version of $P|a_j|C_{max}$. This decision version is sNPH as the maximum value of the deadline D^C is pseudo-polynomially bounded.

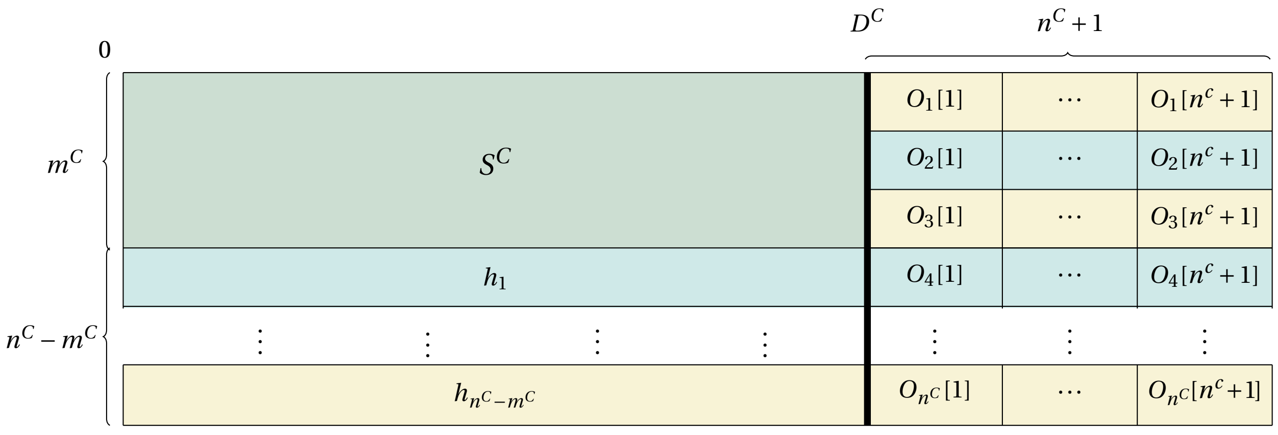

Figure 4.4: Representation of S for the problem $P|a_j,chains|TA(S)$. As a part of solving decision $P|a_j|C_{max}$.

The basic idea behind this proof is that we create an instance of $P|a_j,chains|TA(S)$ that includes all the tasks (T^C) from our instance of decision $P|a_j|C_{max}$. In fact, we keep the assignment and start times from the baseline schedule S^C of $P|a_j|C_{max}$. Such an instance is represented by Figure 4.4. In the created instance, we add a chain of n^C+1 tasks to each processor starting at the deadline. We add a precedence constraint from each task in T^C to one of these chains. Only the first task in each of these chains is available as the others already have a predecessor. As such, this requires n^C chains. To this end we add n^C-m^C additional processors to facilitate these chains. Finishing a task in T^C later than the deadline results in moving at least n^C+1 tasks, as the entire chain dependent on it moves. If the instance of $P|a_j|C_{max}$ should be accepted, we can find a solution to the created instance of $P|a_j,chains|TA(S)$ in which at most n^C tasks (T^C) are moved.

Let an instance of $P|a_j|C_{max}$ be defined by the set of processors P^C, the set of tasks T^C, the durations after the disruption d'^C, the deadline D^C, and the base-line schedule S^C.

Proof. We solve an instance of decision $P|a_j|C_{max}$ through the following reduction:

1. We create a new set $U=\{u_1,\dots,u_{n^C-m^C}\}$ of processors.
2. The set of processors contains P^C and U, $P=P^C\cup U$.
3. We create a list O_j for each processor $p_j\in P$. Each of these lists consists of n^C+1 tasks.
4. For each list O_j, the duration of all tasks is 1: $\forall_{j\in 1,\dots,m}\forall_{t_i\in O_j}, d(t_i)=1$
5. Precedence constraints are added between the tasks in each of the lists. We add an edge from $O_j[i]$ to $O_j[i+1]$ for all $j\in\{1,\dots,m\}$ and all $i\in\{1,\dots,m\}$ to the graph G.
6. Let O be the set of tasks that are contained in some list O_j.
7. In order to avoid the tasks in T^C being executed after those in O, we add an edge from each task $t_i\in T^C$ to $O_i[1]$ to the precedence graph G.
8. We add a task h_i for every processor in U. Each of these tasks has a duration D^C. The set of these tasks is denoted by H.
9. The set of tasks is defined as follows: $T=T^C\cup O\cup H$.

Using T as the set of tasks, P as the set of processors and G as the set of precedence constraints we create S:

10. Copy the start times from S^C, $\forall_{t_i\in T^C}s(S,t_i)=s(S^C,t_i)$.
11. The assignment stays the same (this will always be to a processor in P^C), $\forall_{t_i\in T^C}a(S,t_i)=a(S^C,t_i)$.
12. The list O_j is assigned to processor with index j, i.e. $\forall_{j\in\{1,\dots,m-1\}}\forall_{\{1,\dots,n\}}a(S,O_j[i])=j$.
13. Schedule the tasks in O_j in precedence order after D^C, $\forall_{j\in\{1,\dots,m-1\}}\forall_{\{1,\dots,n\}}s(S,O_j[i])=D^C-1+i$.

14. The tasks in H start at time 0, $\forall h_i \in Hs(S, h_i) = 0$.
15. Each task h_i is assigned to a different processor not in P^C, $\forall h_i \in Ha(S, h_i) = u_i$.

The tasks in T^C are in the interval $[0, D^C)$ as are the tasks in H, but they do not overlap as they are scheduled on a different set of processors (P^C and U respectively). There is no overlap between the tasks in H or T^C and those in O as the tasks in O are in the interval $[D^C, D^C + n + 1)$.

We now solve the problem $P|a_j, chains|TA(S)$, with P, T, G, and d' as input and get a schedule S'. The one task with an increased duration in d', is that which is delayed in the input problem $P|a_j|C_{max}$.

By scheduling a task in T^C to finish later than the deadline D^C in S' at least $n+1$ tasks in O are moved. Without moving tasks in O, at most n tasks can be moved. Due to the precedence constraints between the different tasks in a list O_j, delaying the first task in the list means the remainder of the list must be delayed as well. Due to the each task in T^C being a prerequisite for the first task in at least one of the lists, scheduling it to finish after D^C means moving at least $n+1$ tasks. The tasks in H start at time 0 and as such can not be moved, this leaves us with at most n tasks that can be moved if no tasks in O move.

Given S' for which $TA(S, S') \le n^C$, we can create S'^C by taking the processors in P^C and omitting the tasks in O from S'. All tasks in T^C will be finished before D^C, as otherwise at least $n+1$ tasks would have been moved.

Given a valid S'^C, we can create a valid S' for which $TA(S, S') \le n^C$ by adding the tasks in O and H with the start time and assignment in S. There will be no overlap between O and T^C as the tasks in T^C all finish before D^C and the tasks in O start after D^C. The remaining tasks in O and those in H can be scheduled on the processors in U with their assignment and start time from S.

A valid S'^C exists iff an S' exists for which $TA(S, S') \le n^C$. As the reduction is polynomial and $P|a_j|C_{max}$ is strongly NP-hard, $P|a_j, chains|TA(S)$ is strongly NP-hard.

□

4.5.3. Timed Assignment and Makespan on Two Parallel Processors: $P2|a_j|TA(S), C_{max}$

Theorem 4.5.3. *$P2|a_j|TA(S), C_{max}$ is strongly NP-hard*

We show Theorem 4.5.3 through a reduction from the decision version of $P||C_{max}$ (Definition 4.2.2).

The idea behind our proof is to create a delay on one processor so that our tasks are forced to move to the other. As there are already boundaries in place on the second processor, the tasks are forced to divide over m^C intervals between these barriers. Each of these intervals represents the solution for a processor in the $P||C_{max}$ problem. We can create these boundaries by scaling all tasks by a factor of n^C+1 so that we can fit n^C+1 tasks in what used to be our smallest time unit. An example for $m^C = 3$ is given in figure 4.5. The boundaries consist of n^C+1 tasks of length 1. When we minimize TA, disrupting one such task implies disrupting all of them as we can create an equivalent schedule with fewer disruptions otherwise.

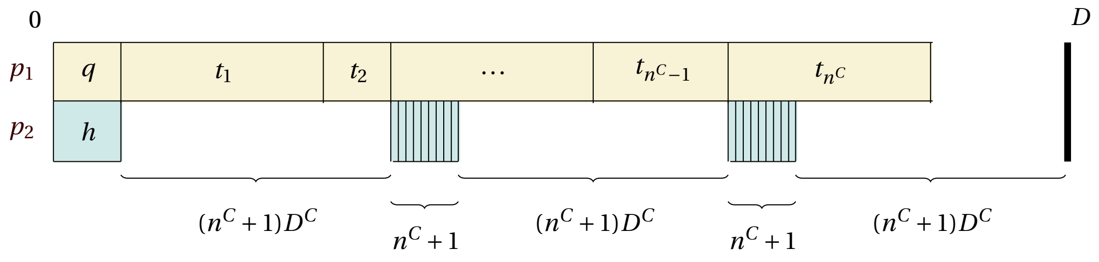

Figure 4.5: Sketch of S for a non-trivial example. In this instance $m^C = 3$.

Proof. Given an instance of Decision $P||C_{max}$ consisting of P^C, T^C, d^C, and D^C, we create an instance of $P2|a_j|TA(S), C_{max}$ as follows:

1. Create a set of two processors $P = p_1, p_2$.

2. Create the tasks in the boundaries. Here X is the set of $(m^C - 1)(n^C + 1)$ tasks that make up the $(m^C - 1)$ boundaries. For each boundary subset of X is denoted by X_j where $j \in \{1, \ldots, m^C - 1\}$, these subsets are disjoint.
3. q is the first task on p_1, h is the first task on p_2.
4. The set of tasks is a combination of the original tasks T^C, the boundary tasks, q and h, i.e. $T = T^C \cup X \cup \{q\} \cup \{h\}$.
5. Scale the duration of all tasks in T^C by $n^C + 1$, $\forall_{t_i \in T^C} d(t_i) = (n^C + 1) d^C(t_i)$.
6. The duration of the boundary tasks is 1, $\forall_{t_i \in X} d(t_i) = 1$.
7. The duration of q and h is 1, $d(q) = 1$, $d(h) = 1$.
8. The deadline is set so that there is enough available time to schedule all tasks in T^C, our barriers X and h on p_2 (if the tasks in T^C divide up), $D = (n^C + 1) m^C D^C + (m^C - 1)(n^C + 1) + 1$.
9. Create S by scheduling all tasks in T^C and q on p_1 and all tasks in X and h on p_2.
10. Both q and h start at time 0, $s(S, q) = s(S, h) = 0$.
11. If $\sum_{t_i \in T^C} d(t_i) > D - 1$ there can be no solution as this implies $\sum_{t_i \in T^C} d^C(t_i) > D^C$, otherwise they fit on p_1. Starting at time 1, the tasks in T^C are scheduled without idle time between them.
12. Start the tasks in X_j in the interval $[(n+1)(D^C + j - 1) + 1, (n+1)(D^C + j) + 1]$. This creates spaces without any scheduled tasks of length $(n+1)(D^C)$.
13. Delay q by $D - 1$, $d'(q) = D$. For all other tasks t_i the duration remains the same $d'(t_i) = d(t_i)$. The delay ensures that q is the only task performed on p_1 as it lasts until D and can not be moved.

The resulting S' will have a value for $TA \leq n$ iff a valid S^C exists. Moving one of the tasks in X implies that we have either moved all of the tasks in an X_j, or we can create a schedule S' with fewer moved tasks by starting the tasks in T^C at a start time that is an integer number times $n + 1$ plus one.

A valid S' with $TA \leq n^C$ implies a valid S^C. We can create S^C from S' by taking the tasks in T^C that are scheduled between X_j and X_{j+1} and assigning them to processor p^C_{j+1}. For processor p^C_1 we take those tasks before X_1. As the tasks in X have not been moved, the tasks in these intervals will be done before D^C. Note that the order on the processors in P^C does not matter and that the task duration is determined by the function d^C.

A valid S^C implies a valid S' with $TA \leq n$. We can create S' from S^C by taking the tasks on processor p^C_{j+1} and assigning them to the interval between X_j and X_{j+1}. For processor p^C_1 we assign them between h and X_1. As the tasks fit (even with the durations d'), the schedule S' is valid. As only n^C tasks are moved, $TA \leq n^C$.

As the bi-implication holds and this reduction is polynomial, $P2|a_j|TA(S), C_{max}$ is strongly NP-hard. □

4.5.4. Timed Assignment and Makespan on Parallel Processors: $P|a_j|TA(S), C_{max}$

Corollary 4.5.3.1. *$P|a_j|TA(S), C_{max}$ is sNPC*

This problem is at least as hard as $P|a_j|C_{max}$ which by Theorem 4.2.2 is sNPH. It is also at least as hard as $P2|a_j|TA(S), C_{max}$ which is sNPH by Theorem 4.5.3.

4.6. Stable Time Window

4.6.1. Stable Time Window on Parallel Processors: $P|a_j|STW(S)$

Theorem 4.6.1. *$P|a_j|STW(S)$ is in P.*

We show Theorem 4.6.1 by proving a greedy algorithm is optimal. In the construction of such an algorithm the first thing we note is that we only need to considered the processor on which the disruption occurs. This is because assignment is required for a task to add to the similarity and there is no bound on the finishing times so tasks that are outside their time window can simply be executed after all other tasks. Unlike in the case of TA, the start times are not fixed.

Without loss of generality, let us consider p_1 the processor on which the disruption occurs. We define an inversion as two tasks t_i and t_j for which it holds that in the baseline schedule S, t_i started before t_j and in the repaired schedule S', t_j starts before t_i. We show that:

Lemma 4.6.2. *For every valid repaired schedule S' that is optimal with regard to maximising STW, a schedule S^* exists in which no tasks that remain in their time window are inverted.*

Proof. Proof by contradiction (exchange argument). Assume Lemma 4.6.2 is false. Then there must be an optimal valid schedule S' with at least one inversion for which no schedule S^* exists without an inversion.

The earliest start time for a task t_i in the repaired schedule S' without leaving the time window is $s(S,t_i)-\alpha$, the latest is $s(S,t_i)+\alpha$. Here α is the parameter determining the size of the time window. Let task t_i start (directly) after task t_j in S' and t_i and t_j be inverted. We can assume they are adjacent because firstly, any task that is not within its time window can be moved to start outside any possible valid time window (very late). Secondly, if an inversion amongst the tasks in their time windows exist there must be an adjacent one as we can loop over all tasks (within their time window) until two form an inversion.

We can now create a schedule S'' in which these two tasks are swapped. t_i will still be in its time window as the lowest $s(S',t_j)$ can be is $s(S,t_j)-\alpha$ which is obviously greater than $s(S,t_i)-\alpha$ as t_i started before t_j in S. It is also less than $s(S,t_i)+\alpha$ as t_i starts later than t_j in S' and is within its time window. Similarly, t_j will still be within its time window, as $s(S',t_i) \le s(S,t_i)+\alpha < s(S,t_j)+\alpha$. As swapping t_i and t_j reduces the number of inverted tasks that are within their time window, S' can not be the optimal schedule with the fewest inversions. This is a contradiction and as such Lemma 4.6.2 must hold.

□

As we can create a schedule for which all tasks that are within their time window remain in order, we can greedily start at the beginning of $L(S,p_1)$ and select tasks until their duration sums up to at least $d'(q)-d(S,q)-\alpha$. These tasks can not be scheduled within their time window. All other tasks can then be delayed by α to create a valid S' in which they stay in their time window. As this can be done in polynomial time, Theorem 4.6.1 holds.

4.6.2. Stable Time Window on Parallel Processors with Chain Precedence Constraints: $P|a_j,chains|STW(S)$

Corollary 4.6.2.1. *$P|a_j,chains|STW(S)$ is sNPC*

When we consider the problem of optimizing STW under chain precedence constraints, it turns out the problem is sNPC. This is because $P|a_j,chains|STW(S)$ is a generalization of $P|a_j,chains|TA(S)$. As shown above, $P|a_j,chains|TA(S)$ is sNPH by Theorem 4.5.2.

4.6.3. Stable Time Window and Makespan on Two Parallel Processors: $P2|a_j|STW(S),C_{max}$

Corollary 4.6.2.2. *$P2|a_j|STW(S),C_{max}$ is sNPC*

When considering the multi-objective problem $P2|a_j|STW(S),C_{max}$, we can see this is a generalization of $P2|a_j|TA(S),C_{max}$. From Theorem 4.5.3 above it follows that $P2|a_j|TA(S),C_{max}$ is sNPH.

4.6.4. Stable Time Window and Makespan on Parallel Processors: $P|a_j|STW(S),C_{max}$

Corollary 4.6.2.3. *$P|a_j|STW(S),C_{max}$ is sNPC*

In the case of an unbounded number of processors, the problem $P|a_j|STW(S),C_{max}$ is at least as hard as $P|a_j|TA(S),C_{max}$. Shown above is that $P|a_j|TA(S),C_{max}$ is sNPH in Corollary 4.5.3.1. As such, Corollary 4.6.2.3 holds.

4.7. Conclusion

Table 4.1: Overview of the computational complexity of the problems considered in this section. The number indicates the theorem or corollary from which this result follows. NPC indicates that it is an open whether this problem is strongly or weakly NPC.

	A	O	TA	STW
$P\|a_j\|met(S)$	$P^{4.3.1.2}$	$P^{4.4.0.1}$	$P^{4.5.1}$	$P^{4.6.1}$
$P\|a_j, chains\|met(S)$	$P^{4.3.1.1}$	$P^{4.3.1}$	$sNPC^{4.5.2}$	$sNPC^{4.6.2.1}$
$P\|a_j, prec\|met(S)$	$P^{4.3.1.1}$	$P^{4.3.1}$	$sNPC^{4.5.2}$	$sNPC^{4.6.2.1}$
$P2\|a_j\|met(S), C_{max}$	$wNPC^{4.3.2}$	$NPC^{4.4.0.2}$	$sNPC^{4.5.3}$	$sNPC^{4.6.2.2}$
$P\|a_j\|met(S), C_{max}$	$sNPC^{4.3.5.2}$	$sNPC^{4.4.0.3}$	$sNPC^{4.5.3.1}$	$sNPC^{4.6.2.3}$

In conclusion, of the problems considered in this chapter, few appear feasible based on their complexity. An overview of the shown complexities can be found in Table 4.1. The complexity of all but one of the problems considered is known. It remains an open research problem whether $P2|a_j|O(S), C_{max}$ is weakly or strongly NP-complete. Those problems that are feasible are the easier problems that may not capture the intricacies of the problem in practice, for instance when performance is also relevant. Some reservations have to be made with regard to these results when applied to a practical context. Due to the structure of the proof of Theorem 4.3.5.2 it requires a processor for each task. In practice, problems tend to have fewer processors than tasks, this proof does not tell us whether instances in which m is bounded in n (eg $m \leq \frac{n}{2}$) are still strongly NP-hard. However, as we can see from the complexity of our problem given only two processors, this algorithm would at best be pseudo-polynomial. It is also impossible for such an algorithm to scale well with in the number of processors due to the strong NP-hardness in the case of an unrestricted number of processors.

As so many of our problems are in fact strongly NP-hard, further research into solving these problems should focus on finding heuristic methods to get usable results or approximation algorithms to get bounds on the quality of the solution.

5

Partial Order Schedules

As we have seen in Chapter 4, many of the problems considered are (strongly) NP-hard making a fast repair unlikely. As such, methods that speed up these repairs will be considered. Many of the reductions that lead to these results come from problems involving the makespan of a schedule. In addition, for many problems the makespan is one of the optimization objectives, making the problem NP-hard by Lemma 4.2.1 or even strongly NP-hard in the case of an unbounded number of processors as shown in Theorem 4.2.2.

Using a Partial Order Schedule (*POS*), we can optimally compute the makespan in polynomial time[40]. This is done by limiting the search space for a scheduling problem by only containing schedules that are valid with regard to the requirements stated in Section 3.1.1. Note that not all valid solutions are generally contained in a *POS*, so optimality in the original problem is sacrificed.

In Section 5.1 *POS*s are formally defined in our scheduling environment, integrated into our disruption model and included in our notation. Finally, Section **??** investigates the complexity of our the problems including a *POS*.

5.1. The Parallel Processor Scheduling Problem with Partial Order Schedules

In this section we extend our definition of the scheduling problem to include *POS*s. First, in Subsection 5.1.1 *POS*s are formally defined in the context of Parallel Processor Scheduling. Second, in Subsection 5.1.2 the model of disruptions within a *POS* is considered. Last, the problem notation is extended to encompass *POS* in Subsection 5.1.3.

5.1.1. Scheduling Environment

In this subsection, the scheduling environment of Section 3.1.1 is extended to include *POS*s. To achieve this, we first give a definition of *POS*s and then define how our base-line and repaired schedule relates to this.

We define a partial order schedule [43] for a parallel processor scheduling problem as follows:

Definition 5.1.1. Given a set of precedence constraints in the form of a graph denoted by Z on the tasks T, it holds that for every combination of start times $s(S, t_i)$ of all tasks $t_i \in T$. Given a schedule S that meats the constraints in Z, a valid assignment $a(S, T)$ exists.

We note that for a valid assignment to exist, at most m (the number of processors) tasks can have overlapping execution times.

We introduce some notation for use bellow. At time x, let U_x denote the set of tasks that has been completed and let W_x denote the remaining tasks in T where $T = U_x \cup W_x$. If for a task $v \in W$ no path exists in the graph Z from a task $w \in \{W_x | v \neq w\}$ to v, this means we can start the task v. Let the set of tasks for which this property holds be denoted by V_x. We know that at any time $|V_x| \leq m$.

When considering a $POS\ Z$ as part of our environment, both the initial schedule S and the repaired schedule S' must meet the precedence constraints of Z.

5.1.2. Disruption Model

The manner in which using a POS effects the disruption model is considered in this subsection. By requiring that both the base-line and the repaired schedule are in (the same) POS, the repair options are limited. This is likely to result in sub-optimal solutions to the original problem.

5.1.3. Problems

In this subsection, we extend the notation defined in Section 3.1.3 to include POSs. Given a problem instance, the input is extended with a POS. Let $\alpha|a_j,POS|\gamma$ denote a problem instance containing a disruption and a POS. Note that as any precedence constraints must be met in the POS and our requirements on the POS do not distinguish between the original problem with or without precedence constraints we do not consider the problems with and without precedence constraints separately.

5.2. Complexity in Partial Order Schedules

In this section the complexity of finding an optimal repair within a POS is considered. As mentioned above in Subsection 5.1.3, POSs that are the result of problems with precedence constraints are not separately considered. Similar to the situation without POSs, we considered the stability metrics as sole objective and as multi-objective in combination with the makespan. In the multi-objective case we again consider both an unbounded number of processors and the case in which there are only two processors. The unbounded case is for scalability in realistic instances, the case with only two processors to determine if for some smaller instance with a fixed low number of processors a fast repair may be possible.

The remainder of this section consists of the complexity of these problems. These are generally divided into subsections by stability metric, some exceptions are made for theorems from which corollaries follow.

5.2.1. Order on Parallel Processors in a Partial Order Schedule: $P|a_j,POS|O(S)$

Theorem 5.2.1. *$P|a_j,POS|O(S)$ is in P.*

The first such exception is the problem of order stability in a scheduling context that includes a POS. Similarly to $P|a_j|O(S)$, $P|a_j,POS|O(S)$ is in P. Right-Shift Rescheduling (RSR)[1] does not change whether the constraints in the POS are met. Nor does it change the order of the tasks.

5.2.2. Assignment on Parallel Processors in a Partial Order Schedule: $P|a_j,POS,prec|A(S)$

Corollary 5.2.1.1. *$P|a_j,POS|A(S)$ is in P.*

By Theorem 3.2.2 $P|a_j,POS|A(S)$ is a special case of $P|a_j,POS|O(S)$. As $P|a_j,POS|A(S)$ is in P by Theorem 5.2.1, so is $P|a_j,POS|A(S)$.

5.2.3. Assignment and Makespan on Parallel Processors in a Partial Order Schedule: $P|a_j,POS|A(S),C_{max}$

In this section we show the complexity of the multi-decision problem for both makespan and assignment does not change given a POS. That is, it stays strongly NP-hard.

Theorem 5.2.2. *$P|a_j,POS|A(S),C_{max}$ is strongly NPC.*

The idea behind our proof is to create a reduction from the decision problem $P||C_{max}$. Due to the POS, tasks in $P|a_j,POS|A(S),C_{max}$ are not generally free to be reordered. While a large part of the complexity of $P||C_{max}$ is that no constraints are placed on the order. In this reduction, we add additional processors and tasks in such a way that the original tasks T^C from $P||C_{max}$ do not have precedence relations between them. We do this by creating additional tasks and processors that we occupy half of the time. By occupying them half of the time, a single chain can cover two processors. By adding enough of them, each task in T^C can be alone in it's chain. As we have the same number of chains as we do processors this is a valid POS. To avoid the extra processors from being used by the tasks in T^C, we scale the duration of these tasks so that moving one of them to these processors that are only occupied half the time results in moving at least n^C+1 tasks. At least n^C+1 tasks must be moved, as the tasks that occupy processors half of the time are part of chains that have no idle time until the deadline so they cannot shift in time.

Proof. We take an instance of the decision problem $P||C_{max}$ as in Definition 4.2.2 and create a (pseudo)polynomial reduction to $P|a_j, POS|A(S), C_{max}$. Let P^C, T^C, D^C, d^C, be our instance of $P||C_{max}$.

1. Let the number of tasks in T^C be n^C $(= |T^C|)$.
2. Create a set $Z = \{z_1, \ldots, z_{2n^C}\}$ of $2n^C$ processors, p_0 is a special processor and $P = P^C \cup Z \cup \{p_0\}$.
3. Create a set H of m^C tasks$(= |P^C|)$.
4. Create a set Y of $l = 2(n^C + 1)D^C \cdot n^C$ tasks, t_0 is a special task and $T = T^C \cup H \cup\ Y \cup \{t_0\}$. The tasks in Y are in one of n sequences Y_i of length $2(n+1)D^C$. We let $Y_i[j]$ indicate the j^{th} element of the i^{th} sequence.
5. The duration of each task t_i in T^C is multiplied by $2(n^C + 1)$, i.e. $\forall_{t_i \in T^C} d(t_i) = 2(n^C + 1)d^c(t_i)$.
6. We set the duration of t_0 and all tasks in Y to 1, i.e. $d(t_0) = 1$ and $\forall_{y_i \in Y} d(y_i) = 1$.
7. The deadline is set to allow all tasks in T^C and t_0 to fit on p_0 initially. $D = 1 + 2(n^C + 1)\sum_{t_i \in T^C} d(t_i)$.
8. For the tasks h_i in H the duration is $D - 2(n^C + 1)D^C$. So that $2(n^C + 1)D^C$ time is available to schedule other tasks on the same processor in the repaired schedule S'.
9. Our POS consists of several chains. Each task $t_i \in T^C$ is a chain on its own, as is each task in H. Similarly, t_0 is also a chain on its own.
10. The tasks in Y form n^C chains, each Y_i being one. We have an edge from $Y_i[j]$ to $Y_i[j+1]$ for all $i \in \{1, \ldots, n\}$ and all $j \in \{1, \ldots, D-1\}$.
11. As such, we have a total of $2n^C + m^C + 1$ chains and $2n^C + m^C + 1$ processors making our POS valid, as at every time at most one task from each chain can be scheduled.

We create a schedule S:

12. The tasks in Y are scheduled starting at time 0, without time between two successive tasks, so $s(S, Y_i[j]) = j$.
13. The tasks in Y are assigned to processors in Z. Those in Y_i are divided over processors z_{2i-1} and z_{2i}. i.e. $a(S, Y_i[j]) = z_{2i-1}$ if j is even and $a(S, Y_i[j]) = z_{2i}$ if j is odd. Because each Y_i is split over two processors, the processors in Z are idle half of the time.
14. t_0 starts at time 0 and is assigned to p_o, i.e. $s(S, t_0) = 0$ and $a(S, t_0) = p_0$.
15. We schedule the tasks $t_i \in T^C$ at time $s(S, t_i) = f(S, t_{i-1})$ with $a(S, t_i) = p_0$.
16. The tasks in H are scheduled on P^C, starting at time 0. These are the processors the tasks in T^C can go to in S' without moving more than n other tasks. i.e. $a(S, h_i) = p_i^C$ and $s(S, h_i) = 0$.
17. Our disruption to S is the delay of t_0 by $D - 1$, so $d'(t_0) = D$ and for all other tasks $d'(t_i) = d(t_i)$.

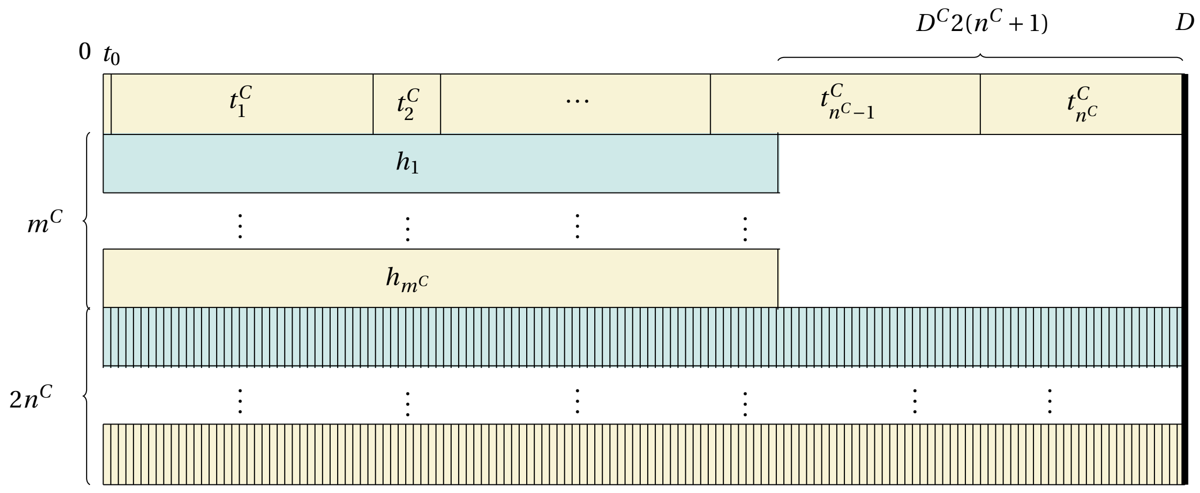

Figure 5.1: Graphical representation of S

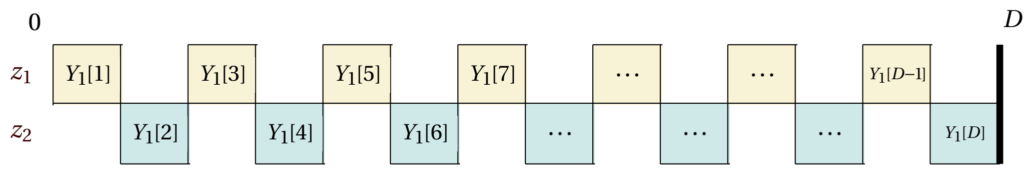

Figure 5.2: A single chain Y_1 scheduled on two processors z_1 and z_2.

Before the disruption, the situation is depicted in Figure 5.1, we can see only the processors in P^C and p_0 have room available to move to without disrupting at least n^C+1 tasks. A more detailed view on the construction of the chains is given in Figure 5.2. After the disruption to the duration of t_0, the tasks must move from p_0 to a processor in P^C (or the assignment metric is higher than n^C).

If the tasks in T^C can be scheduled on P^C to finish before D, $A(S,S') = n^C$. This is because scheduling one of the tasks in T^C on a processor in Z disrupts at least $n+1$ tasks and it is impossible to move t_0 (as this is the delayed task) and the tasks in H (as they start at the moment of disruption) so $A(S,S') \le n^C$. And as all n tasks are assigned to p_0 in S and must be moved, $A(S,S') \ge n^C$, so $A(S,S') = n^C$ is the optimum.

If a valid S' exists while $A(S,S') = n$, we can construct S^C by reducing the start times to offset the h_i, scaling the start time back, and keeping the assignment. i.e. $s(S^C,t_i) = \lfloor \frac{s(S',t_i)-d'(h_i)}{2(n+1)} \rfloor$ and $a(S^C,t_i) = a(S',t_i)$ for all $t_i \in T^C$. We floor the start time because a task may have delayed into idle time after the task.

If a valid S^C exists, it must hold that $A(S,S') = n$ for an optimal S'. This is the case as we can create such a schedule by scaling the start times and delaying them by the duration of h_i, the assignment stays the same. i.e. $s(S',t_i) = s(S^C,t_i)2(n+1)+d'(h_i)$ and $a(S^C,t_i) = a(S',t_i)$ for all $t_i \in T^C$. As these processors are empty before, and none of the other tasks are scheduled after D, we alter the assignment of only n tasks.

As S' such that $A(S,S') = n$ implies S^C and S^C implies an S' exists such that $A(S,S') = n$, this pseudo-polynomial reduction means $P|a_j,POS|A(S),C_{max}$ is as hard as the decision problem $P||C_{max}$ and therefore strongly NP-hard. □

5.2.4. Order and Makespan on Parallel Processors in a Partial Order Schedule: $P|a_j,POS|A(S),C_{max}$

Corollary 5.2.2.1. *$P|a_j,POS|O(S),C_{max}$ is strongly NPC.*

This problem is at least as hard as $P|a_j,POS|A(S),C_{max}$, as the order metric is a generalization of the assignment metric. As $P|a_j,POS|A(S),C_{max}$ is sNPC by Theorem 5.2.2, it follows that $P|a_j,POS|O(S),C_{max}$ is also sNPC.

5.2.5. Timed Assignment on Parallel Processors in a Partial Order Schedule: $P|a_j, POS|TA(S)$

Theorem 5.2.3. *If there is no idle time in the base-line schedule S, $P|a_j, POS|TA(S)$ is in P.*

Proof. When a task q is disrupted at time x, $q \in V_x$. Let us denote the set of tasks scheduled after q on the same processor by C. If there is a path in our POS from q to all elements of C these tasks will be delayed and not be scheduled at the same time. In addition, any task to which there is a path in our POS from q will be delayed.

If there is no such path, for some element $c_i \in C$. c_i must be in V_x. As c_i can be executed in parallel to q it must hold that there is another processor available and c_i can be executed at the same time but on a different processor. If this holds for all tasks in C, their assignment must change. However, all tasks for which a path exists from q in the POS but are not in C can be scheduled at the same time on the same processor.

If a path exists from q only to some elements in C, any task to which there is a path in the POS from q will still be delayed. □

If idle time in S is allowed, we show that $P|a_j, POS|TA$ is at least as hard as the decision version corresponding to $P||C_{max}$.

Theorem 5.2.4. *$P|a_j, POS|TA(S)$ is at least as hard as Decision $P||C_{max}$.*

The idea behind our proof is to take an instance of the decision problem $P||C_{max}$ and create a polynomial reduction to $P|a_j, POS|TA$. Combining the proof concepts of using O from Section 4.5.2 to ensure the deadline is met, Z from Section 5.2.3 to allow the freedom of movement for the jobs within the POS and the dividers X from Section 4.5.3 to achieve deciding the makespan outside a POS.

Proof. We take an instance of Decision $P||C_{max}$ as in Definition 4.2.2 and create a reduction to $P|a_j, POS|TA(S)$. Let P^C, T^C, D^C, d^C, be our instance of $P||C_{max}$.

We first create our tasks and processors:

1. Create two processors p_1 and p_2, and a set of $2(n^C+1)$ processors Z. Now let our set of processors $P = Z \cup \{p_1\} \cup \{p_2\}$
2. Create the tasks in the boundaries. Here X is the set of $(m^C-1)(n^C+1)$ tasks that make up the (m^C-1) boundaries. For each boundary subset of X is denoted by X_j where $j \in \{1,\dots,m^C-1\}$, these subsets are disjoint.
3. Create a set Y of $(n^C+1)D$ tasks.
4. Let D denote a deadline, we will show later that a task in T^C passing this deadline will incur a value of $TA \geq n^C+1$. The deadline D is set so that there is enough available time to schedule all tasks in T^C, h, and our barriers X on p_2, if there is a feasible solution to the instance of Decision $P||C_{max}$. As such the deadline $D = 2(n^C+1)m^C D^C + (m^C-1)(n^C+1)+1$.
5. The tasks in Y are in one of n^C+1 sequences Y_i of length D. Where $Y_i[j]$ indicates the j^{th} element of the i^{th} sequence.
6. $T = T^C \cup X \cup Y \cup\ O \cup \{q\} \cup \{h\}$.

Next we define the duration of all tasks:

7. The duration of each task t_i in T^C is multiplied by $2(n^C+1)$, i.e. $\forall_{t_i \in T^C} d(t_i) = 2(n^C+1)d^C(t_i)$.
8. We set the duration all tasks in Y to 1, i.e. $\forall_{y_i \in Y} d(y_i) = 1$.
9. The duration of the boundary tasks is 1, $\forall_{t_i \in X} d(t_i) = 1$.
10. The duration of q and h is 1, $d(q) = 1$, $d(h) = 1$.

We define the precedence constraints as part of the POS:

11. Our POS consists of several chains. q is a chain on its own, as is h.

12. Precedence constraints are added between the tasks in each of the lists in O. We add an edge from $O_j[i]$ to $O_j[i+1]$ for all $j \in \{1,\dots,m\}$ and all $i \in \{1,\dots,m\}$ to the graph G. In order to avoid the tasks in T^C being executed after those in O, we add an edge from each task $t_i \in T^C$ to $O_i[1]$ to the POS. This step forms n^C chains.
13. The tasks in Y form n^C+1 chains, each Y_i being one. We have an edge from $Y_i[j]$ to $Y_i[j+1]$ for all $i \in \{1,\dots,n\}$ and all $j \in \{1,\dots,2(n^C+1)D^C-1\}$.
14. All tasks in X form a single chain.
15. As such, we have a total of $2n^C+4$ chains and $2(n^C+1)+2$ processors making our POS valid.

Create the baseline schedule S:

16. Schedule all tasks in T^C and q on p_1 and all tasks in X and h on p_2.
17. q is the first task on p_1, h is the first task on p_2. Both q and h start at time 0, $s(S,q)=s(S,h)=0$.
18. The list O_j is assigned to processor with index j, i.e. $\forall_{j\in\{1,\dots,m-1\}}\forall_{\{1,\dots,n\}}a(S,O_j[i])=j$.
19. Schedule the tasks in O_j in precedence order after D, $\forall_{j\in\{1,\dots,m-1\}}\forall_{\{1,\dots,n\}}s(S,O_j[i])=D-1+i$.
20. If $\sum_{t_i\in T^C} d(t_i) > D-1$ there can be no solution as this implies $\sum_{t_i\in T^C} d^C(t_i) > D^C$, otherwise they fit on p_1 before the tasks in O. Starting at time 1, the tasks in T^C are scheduled without idle time between them.
21. Start the tasks in X_j in the interval $[2(n^C+1)(D^C+j-1)+1, 2(n^C+1)(D^C+j)+1]$. This creates spaces without any scheduled tasks of length $(n^C+1)(D^C)$.
22. The tasks in Y are assigned to processors in Z. Those in Y_i are divided over processors z_{2i-1} and z_{2i}. i.e. $a(S,Y_i[j])=z_{2i-1}$ if j is even and $a(S,Y_i[j])=z_{2i}$ if j is odd. Because each Y_i is split over two processors, the processors in Z are idle half of the time.
23. The tasks in Y are scheduled starting at time 0, without time between two successive tasks, so $s(S,Y_i[j])=j$.

Finally, a disruption is incurred:

24. Delay q by $D-1$, $d'(q)=D$. For all other tasks t_i the duration remains the same $d'(t_i)=d(t_i)$.

By scheduling a task in T^C to finish later than the deadline D in the repaired schedule S' at least $n+1$ tasks in O are moved. Without moving tasks in O, at most n tasks can be moved. Due to the precedence constraints between the different tasks in a list O_j, delaying the first task in the list means the remainder of the list must be delayed as well. Due to the each task in T^C being a prerequisite for the first task in at least one of the lists, scheduling it to finish after D^C means moving at least n^C+1 tasks.

If $TA \le n^C$, no task in T^C can be scheduled on a processor other than t_1. Scheduling one of the tasks in T^C on a processor in Z disrupts at least $n+1$ tasks. The only other option is p_0, which contains q which cannot be moved and lasts until D.

Lastly, we can not schedule any of the tasks in T^C on p_1 at a time during which in S a task in X was scheduled without increasing TA past n^C. Moving one of the tasks in X implies that we have either moved all of the tasks in an X_j, or we can create a schedule S' with fewer moved tasks by starting the tasks in T^C at a start time that is an integer number times $n+1$ plus one. As such, the resulting S' will have a value for $TA \le n$ iff a valid S^C exists.

A valid S' with $TA \le n^C$ implies a valid S^C. We can create S^C from S' by taking the tasks in T^C that are scheduled between X_j and X_{j+1} and assigning them to processor p^C_{j+1}. For processor p^C_1 we take those tasks before X_1. As the tasks in X have not been moved, the tasks in these intervals will be done before D^C. Note that the order on the processors in P^C does not matter and that the task duration is determined by the function d^C.

A valid S^C implies a valid S' with $TA \le n$. We can create S' from S^C by taking the tasks on processor p^C_{j+1} and assigning them to the interval between X_j and X_{j+1}. For processor p^C_1 we assign them between h and X_1. As none of the tasks in T^C are in the same chain, no precedence constraint will be violated. In addition, the tasks fit (even with the durations d'), as such the schedule S' is valid. As only n^C tasks are moved, $TA \le n^C$.

As the bi-implication holds and this reduction is polynomial, $P|a_j, POS|TA(S)$ is strongly NP-hard.

□

5.2.6. Timed Assignment and Makespan on Parallel Processors in a Partial Order Schedule: $P|a_j, POS|TA(S)$

Corollary 5.2.4.1. *$P|a_j, POS|TA(S), C_{max}$ is sNPC.*

As shown above, $P|a_j, POS|TA(S)$ is sNPC by Theorem 5.2.4. As we now additionally consider the makespan, this problem must be at least as hard.

5.2.7. Stable Time Window on Parallel Processors in a Partial Order Schedule: $P|a_j, POS|STW(S)$

Corollary 5.2.4.2. *$P|a_j, POS|STW(S)$ is sNPC.*

The Stable Time Window metric is a generalization of the Timed Assignment metric in which tasks are counted as contributing to the stability if their start time is within a fixed bound from their original start time. As such, this problem must be at least as hard as $P|a_j, POS|TA(S)$ which is strongly NPC by Theorem 5.2.4.

5.2.8. Stable Time Window and Makespan on Parallel Processors in a Partial Order Schedule: $P|a_j, POS|STW(S), C_{max}$

Corollary 5.2.4.3. *$P|a_j, POS|STW(S), C_{max}$ is strongly NPC.*

Because this problem considers the makespan as well as $STW(S)$ it is a generalization of $P|a_j, POS|STW(S)$. As $P|a_j, POS|STW(S)$ is sNPC by Corollary 5.2.4.2, it follows that $P|a_j, POS|STW(S), C_{max}$ is also sNPC.

5.3. Conclusion

Table 5.1: Overview of the computational complexity of the problems considered in this section. The number indicates the theorem or corollary from which this result follows. NPC indicates that it is an open whether this problem is strongly or weakly NPC and a ? indicates that the complexity is unknown.

	A	O	TA	STW
$P\|a_j, POS\|met(S)$	$P^{5.2.1.1}$	$P^{5.2.1}$	$sNPC^{5.2.4}$	$sNPC^{5.2.4.2}$
$P\|a_j, POS\|met(S), C_{max}$	$sNPC^{5.2.2}$	$sNPC^{5.2.2.1}$	$sNPC^{5.2.4.1}$	$sNPC^{5.2.4.3}$

The considered problems in POSs turns out to have a complexity at least as large as those without such a constraint. A full overview of the problems examined and their complexity can be found in Table 5.1. No results are known for problems involving a bound number of processors. However, due to the problems being strongly NP-hard for an unbounded number it cannot scale well with the number of processors.

While makespan as a performance objective can be computed in polynomial time, this result does not extend to any of our stability objectives being considered with the makespan. Due to the way in which the considered metrics are inherently dependent on the assignment of tasks to resources, reductions can be constructed that allow the free movement of the tasks in the original (not inside a POS) tasks. In fact, in these instances the POS can be used to include precedence constraints on these original tasks.

We can not say that in general, for metrics that depend on the assignment the problem is at least as hard in a POS as it is outside. While the results in this section are consistent with this hypothesis. They are incomplete as mentioned above. Furthermore, for metrics that do not have certain desirable properties, e.g. allowing a small group of resources to contribute all of the differences, this may well not hold.

In conclusion, for the stability problems defined in Section 1.1, it is unlikely that an improvement in complexity will be found through the use of POSs. By focussing on different problems in stability or relaxing certain requirements it may yet be possible to do so.

6 Conclusion

In this chapter we review the results regarding the complexity of obtaining stable, efficient repairs to schedules. Our results show that in the general case with multiple machines, it is computationally infeasible to create an optimally repaired schedule. For specific cases in which we do so, efficient algorithms are given. In this chapter we give an overview of our results, discuss their impact, the relation between them and finally we suggest directions for future work within the context of stable scheduling.

6.1. Discussion

The problems that allow for a (pseudo) polynomial algorithm have been identified. This includes considered stability metrics as well as the multi-objective problem of makespan and assignment on two processors4.3.3. For this subset of problems fast optimal repairs can be made.

This work also shows the infeasibility of computing the multi-objective optimization problem of stability objectives in combination with the makespan given an unbounded number of processors. This is also an important result as it indicates concessions will have to be made to the optimality in order to facilitate a fast repair. As a result of this, when creating the schedule ahead of time while having knowledge of the repair policy[49], either more work needs to be done ahead of time to facilitate faster repairs or the sub-optimality of the repair will need to be accepted. The stability metrics are selected to be the easiest subset that still contains the characteristics of our motivating examples in 1.1. As such, it is unlikely that different metrics capturing the important aspects of these examples that can be computed more efficiently will be found. All of the used stability metrics emphasise the assignment between resources and tasks. Metrics that do not use this coupling might be easier to compute. However, they will invariably lose an aspect of stability that is important in many situations. The makespan is both one of the most frequently used and easiest performance metrics. The complexity of other objectives such as the average completion time is not known to be easier[39]. As such, our contribution shows that it is unlikely that efficient algorithms exist for multi-objective problems with both stability and performance objectives on an unbounded number of processors.

Another attempt to achieve a lower complexity would be to alter the disruption model. Again, the model is aimed to be a simple as possible without losing that which defines the problem. A suggested option in literature is to not allow tasks after the disruption to start earlier[47]. While this might make the problem easier to solve, it is likely to create significantly less stable schedules. Whereas most real world scenarios considering stability allow for tasks that start earlier. An alternative would be to restrict the search space in a different manner. For instance through the use of *POS*s as in Chapter 5.

*POS*s were intended to be used to more efficiently create repairs in multi-objective problems as they enable minimizing the makespan in polynomial time. Much of the complexity in our multi-objective problems comes from the makespan. As it turns out, this does not decrease the complexity of our considered problems. While this is surprising, it can be explained due to the *POS* containing no information on the assignment of the resources, this is an important aspect to all considered stability metrics. An alternative to the *POS* would be to restrict tasks from switching between resources. While this would likely be good for the stability of our schedule, the impact in the makespan could in many instances be rather large. This is due to there being no way to divide the delay over different resources when the assignment is fixed. We have shown that POSs do not offer reduced complexity in these instances, it is likely that they do not offer reduced complexity in problems that consider both stability and performance objectives.

Lastly, we discuss the limits to the practical implications of our analysis using complexity theory. While this enables us to draw certain conclusions about the manner in which problems scale, it does not directly allow us to conclude that real world scenarios are infeasible. Scheduling problems can have large input sizes. As such, strongly NP-hard problems tend not to be feasible to compute exactly . Even so, there is no guarantee that for an acceptable bound on the size of a realistic problem no algorithm exists that computes an optimal solution in an acceptable amount of time. However, computing the solution to such a problem quickly would imply a (very) low constant in the runtime of the algorithm which seems unlikely.

6.2. Future Work

As mentioned above, the selected metrics capture our motivating examples, for example the timed assignment characterising the amount of taxies that are effected in the existing schedule. While they attempt to be the easiest metrics to compute that do this, no proof for this is given. We suggest further research into the properties of stability metrics, and the complexity of computing metrics with these properties. An important next step is to give a complete identification of under which conditions (metrics, environment) plan repair is polynomially solvable.

As it is computationally hard to compute optimal solutions in the general case, an area for further research would be to attempt to create sub-optimal solutions. In order to have a bound on how far from optimal the solution is, approximation algorithms or relaxations can be used. In literature there are examples of using approximation algorithms to minimize the makespan in a static context[25, 32].

An alternative would be to look into algorithms that give no bound on the quality of our solution but do tend to create good solutions. For example through using heuristics to effect a repair, we have seen this in this document in the Right Shift Rescheduling[1] approach. Heuristics are also used in the creation of static schedules with the aim of minimizing the makespan[14]. Another approach that gives no guarantees on the solution quality is machine learning. As we can evaluate the quality of our repaired schedule reasonably quickly this is an avenue worth investigating. An overview in static scheduling is given by Aytug et al.[4]. Additionally, if we do have particular information on the probability distribution of the disruption, stochastic scheduling approaches can be used[8]. Finally, in situations where a lot of time and computational resources are available before the start of the schedule, all situations can be considered in advance. In effect, creating something similar to a supermodel[21].

6.3. Closing

In closing, this work considers stable repairs to schedules. The concept of stability when the resource and the task may be human is used. This concept is based on several situations in Section 1.1 showing the relevance, such as the patient doctor assignment. We select metrics that encompass stability in these situations, aiming to make them computationally easy. Our disruption model is also aimed at being simple while still realistic. The performance metric is the most common. For the problems for which an efficient algorithm exists, it is given. For those for which none can exists, proofs of strong NP-hardness are given. As such, we can now limit ourselves to looking for non-optimal methods of solving these problems as research continues in the field of stable scheduling. So that one day when things do not go according to plan, we may find a similar and efficient repaired schedule without having to wait for it.

Bibliography

[1] Raida Jarrar Abumaizar and Joseph A Svestka. Rescheduling job shops under random disruptions. *International journal of production research*, 35(7):2065–2082, 1997.

[2] Oğuzhan Alagöz and Meral Azizoğlu. Rescheduling of identical parallel machines under machine eligibility constraints. *European Journal of Operational Research*, 149(3):523–532, 2003.

[3] Jean-Paul Arnaout and Ghaith Rabadi. Rescheduling of unrelated parallel machines under machine breakdowns. *International Journal of Applied Management Science*, 1(1):75–89, 2008.

[4] Haldun Aytug, Siddhartha Bhattacharyya, Gary J Koehler, and Jane L Snowdon. A review of machine learning in scheduling. *IEEE Transactions on Engineering Management*, 41(2):165–171, 1994.

[5] Haldun Aytug, Mark A Lawley, Kenneth McKay, Shantha Mohan, and Reha Uzsoy. Executing production schedules in the face of uncertainties: A review and some future directions. *European Journal of Operational Research*, 161(1):86–110, 2005.

[6] Debdeep Banerjee and Patrik Haslum. Partial-order support-link scheduling. In *Twenty-First International Conference on Automated Planning and Scheduling*, pages 307–310, 2011.

[7] Roman Barták, Tomáš Müller, and Hana Rudová. A new approach to modeling and solving minimal perturbation problems. In *International Workshop on Constraint Solving and Constraint Logic Programming*, pages 233–249. Springer, 2003.

[8] Anna Bonfill, Antonio Espuña, and Luis Puigjaner. Addressing robustness in scheduling batch processes with uncertain operation times. *Industrial & engineering chemistry research*, 44(5):1524–1534, 2005.

[9] Jeb Brooks, Emilia Reed, Alexander Gruver, and James C Boerkoel. Robustness in probabilistic temporal planning. In *AAAI*, pages 3239–3246, 2015.

[10] Massimo Cairo and Romeo Rizzi. Dynamic controllability made simple. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 90. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[11] Kevin M Calhoun, Richard F Deckro, James T Moore, James W Chrissis, and John C Van Hove. Planning and re-planning in project and production scheduling. *Omega*, 30(3):155–170, 2002.

[12] Yuerong Chen, Xueping Li, and Rapinder Sawhney. Restricted job completion time variance minimisation on identical parallel machines. *European Journal of Industrial Engineering*, 3(3):261–276, 2009.

[13] Carlo Combi, Luke Hunsberger, and Roberto Posenato. An algorithm for checking the dynamic controllability of a conditional simple temporal network with uncertainty. *Evaluation*, 1:1, 2013.

[14] Ali Dogramaci and Julius Surkis. Evaluation of a heuristic for scheduling independent jobs on parallel identical processors. *Management Science*, 25(12):1208–1216, 1979.

[15] Jianzhong Du and Joseph Y-T Leung. Minimizing total tardiness on one machine is np-hard. *Mathematics of operations research*, 15(3):483–495, 1990.

[16] Hani El Sakkout and Mark Wallace. Probe backtrack search for minimal perturbation in dynamic scheduling. *Constraints*, 5(4):359–388, 2000.

[17] Abdallah Elkhyari, Christelle Guéret, and Narendra Jussien. Solving dynamic resource constraint project scheduling problems using new constraint programming tools. In *International Conference on the Practice and Theory of Automated Timetabling*, pages 39–59. Springer, 2002.

[18] Na Fu, Pradeep Varakantham, and Hoong Chuin Lau. Towards finding robust execution strategies for rcpsp/max with durational uncertainty. In *Twentieth International Conference on Automated Planning and Scheduling*, pages 73–80, 2010.

[19] Alex Fukunaga. An improved search algorithm for min-perturbation. In *International Conference on Principles and Practice of Constraint Programming*, pages 331–339. Springer, 2013.

[20] Michael R Garey and David S Johnson. “strong” np-completeness results: Motivation, examples, and implications. *Journal of the ACM (JACM)*, 25(3):499–508, 1978.

[21] Matthew L Ginsberg, Andrew J Parkes, and Amitabha Roy. Supermodels and robustness. In *AAAI/IAAI*, pages 334–339, 1998.

[22] Emmanuel Hebrard, Brahim Hnich, Barry O’Sullivan, and Toby Walsh. Finding diverse and similar solutions in constraint programming. In *AAAI*, volume 5, pages 372–377, 2005.

[23] Willy Herroelen and Roel Leus. The construction of stable project baseline schedules. *European Journal of Operational Research*, 156(3):550–565, 2004.

[24] Willy Herroelen and Roel Leus. Project scheduling under uncertainty: Survey and research potentials. *European journal of operational research*, 165(2):289–306, 2005.

[25] Dorit S Hochbaum and David B Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *Journal of the ACM (JACM)*, 34(1):144–162, 1987.

[26] Te C Hu. Parallel sequencing and assembly line problems. *Operations research*, 9(6):841–848, 1961.

[27] Amy Huang, Liam Lloyd, Mohamed Omar, and James C Boerkoel Jr. New perspectives on flexibility in simple temporal planning. In *The 28th International Conference on Automated Planning and Scheduling*, pages 123–131, 2018.

[28] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[29] Jon Kleinberg and Eva Tardos. *Algorithm design*. Pearson Education Limited, pearson new international edition first edition edition, 2014.

[30] Eugene L Lawler. A “pseudopolynomial” algorithm for sequencing jobs to minimize total tardiness. In *Annals of discrete Mathematics*, volume 1, pages 331–342. Elsevier, 1977.

[31] Eugene L Lawler, Jan Karel Lenstra, Alexander HG Rinnooy Kan, and David B Shmoys. Sequencing and scheduling: Algorithms and complexity. *Handbooks in operations research and management science*, 4: 445–522, 1993.

[32] Jan Karel Lenstra, David B Shmoys, and Eva Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical programming*, 46(1-3):259–271, 1990.

[33] Roel Leus and Willy Herroelen. The complexity of machine scheduling for stability with a single disrupted job. *Operations Research Letters*, 33(2):151–156, 2005.

[34] Michael Lindahl, Thomas Stidsen, and Matias Sørensen. Quality recovering of university timetables. *European Journal of Operational Research*, 2019.

[35] Le Liu and Hong Zhou. On the identical parallel-machine rescheduling with job rework disruption. *Computers & Industrial Engineering*, 66(1):186–198, 2013.

[36] Ashish M Mehta, Jay Smith, Howard Jay Siegel, Anthony A Maciejewski, Arun Jayaseelan, and Bin Ye. Dynamic resource allocation heuristics that manage tradeoff between makespan and robustness. *The Journal of Supercomputing*, 42(1):33–58, 2007.

[37] Paul Morris. A structural characterization of temporal dynamic controllability. In *International Conference on Principles and Practice of Constraint Programming*, pages 375–389. Springer, 2006.

[38] Ronan O'Donovan, Reha Uzsoy, and Kenneth N McKay. Predictable scheduling of a single machine with breakdowns and sensitive jobs. *International Journal of Production Research*, 37(18):4217–4233, 1999.

[39] M Ozlen and M Azizoğlu. Rescheduling unrelated parallel machines with total flow time and total disruption cost criteria. *Journal of the Operational Research Society*, 62(1):152–164, 2011.

[40] Nicola Policella, Amedeo Cesta, Angelo Oddi, and Stephen F Smith. From precedence constraint posting to partial order schedules. *AI Communications*, 20(3):163–180, 2007.

[41] AS Raheja and V Subramaniam. Reactive recovery of job shop schedules–a review. *The International Journal of Advanced Manufacturing Technology*, 19(10):756–763, 2002.

[42] Yongping Ran, Nico Roos, and Jaap van den Herik. Approaches to find a near-minimal change solution for dynamic csps. In *Fourth international workshop on integration of AI and OR techniques in constraint programming for combinatorial optimisation problems*, pages 373–387. Citeseer, 2002.

[43] Riccardo Rasconi, Nicola Policella, and Amedeo Cesta. Fix the schedule or solve again? comparing constraint-based approaches to schedule execution. In *Proceedings of the ICAPS Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems*, pages 46–53, 2006.

[44] Nico Roos, Yongping Ran, and Jaap Van Den Herik. Combining local search and constraint propagation to find a minimal change solution for a dynamic csp. In *International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, pages 272–282. Springer, 2000.

[45] Francesca Rossi, Kristen Brent Venable, and Neil Yorke-Smith. Controllability of soft temporal constraint problems. In *International Conference on Principles and Practice of Constraint Programming*, pages 588–603. Springer, 2004.

[46] Ioannis Tsamardinos, Martha E Pollack, and Sailesh Ramakrishnan. Assessing the probability of legal execution of plans with temporal uncertainty. In *Proc. of ICAPS'03 Workshop on Planning Under Uncertainty and Incomplete Information*, 2003.

[47] Stijn Van de Vonder, Erik Demeulemeester, Willy Herroelen, and Roel Leus. The use of buffers in project management: The trade-off between stability and makespan. *International Journal of production economics*, 97(2):227–240, 2005.

[48] Stijn Van de Vonder, Erik Demeulemeester, Willy Herroelen, and Roel Leus. The trade-off between stability and makespan in resource-constrained project scheduling. *International Journal of Production Research*, 44(2):215–236, 2006.

[49] Roman Van Der Krogt and Mathijs De Weerdt. Plan repair as an extension of planning. In *ICAPS*, volume 5, pages 161–170, 2005.

[50] Michel Wilson, Tomas Klos, Cees Witteveen, and Bob Huisman. Flexibility and decoupling in the simple temporal problem. In *IJCAI*, pages 2422–2428, 2013.

[51] S David Wu, Robert H Storer, and Chang Pei-Chann. One-machine rescheduling heuristics with efficiency and stability as criteria. *Computers & Operations Research*, 20(1):1–14, 1993.

[52] Yunqiang Yin, TCE Cheng, and Du-Juan Wang. Rescheduling on identical parallel machines with machine disruptions to minimize total completion time. *European Journal of Operational Research*, 252(3): 737–749, 2016.

[53] Jerrold H Zar. Significance testing of the spearman rank correlation coefficient. *Journal of the American Statistical Association*, 67(339):578–580, 1972.

[54] Chuanli Zhao and Hengyong Tang. Rescheduling problems with deteriorating jobs under disruptions. *Applied Mathematical Modelling*, 34(1):238–243, 2010.

[55] Roie Zivan, Alon Grubshtein, and Amnon Meisels. Hybrid search for minimal perturbation in dynamic csps. *Constraints*, 16(3):228–249, 2011.