On the complexity of task graph scheduling with transient and fail-stop failures

Anne Benoit[1], Louis-Claude Canon[2,3], Emmanuel Jeannot[3] and Yves Robert[1]

1. LIP (jointly operated by CNRS, ENS Lyon, INRIA and UCB Lyon), ENS Lyon,
46 Allée d'Italie, 69364 Lyon Cedex 07, France.
{Anne.Benoit|Yves.Robert}@ens-lyon.fr
2. Nancy University
3. LaBRI and INRIA Bordeaux
{Louis-Claude.Canon|Emmanuel.Jeannot}@inria.fr

Research Report LIP-2010-01

January 2010

Abstract

This paper deals with the complexity of task graph scheduling with transient and fail-stop failures. While computing the reliability of a given schedule is easy in the absence of task replication, the problem becomes much more difficult when task replication is used. Our main result is that this problem is #P'-Complete (hence at least as hard as NP-Complete problems), with both transient and fails-stop processor failures. We also study the complexity of a restricted class of schedules, where a task cannot be scheduled before all replicas of all its predecessors have completed their execution.

1 Introduction

Since the landmark papers of Bannister and Trivedi [1], Shatz et al. [12, 13], and Kartik and Ram Murthy [10], numerous papers have dealt with reliability issues in task graph scheduling. With processors subject to faults, the natural idea is to duplicate the execution of some tasks. This will increase the probability that the execution is successful in presence of one or several failures during the execution. There are two major failure types, *transient* and *fail-stop*. In a nutshell, *transient* failures invalidate only the execution of the current task, and the processor subject to that failure will be able to recover and execute the subsequent tasks assigned to it (if any). On the contrary, *fail-stop* failures are unrecoverable: once the fault occurs, the corresponding processor is down until the end of the whole execution.

The problem of computing the reliability of a schedule, (*i.e.*, the probability that its execution is successful), has been partially addressed in the literature. Although a recent paper recognizes that computing the reliability of a schedule is a difficult task and proposes exponential cost algorithms [6], to the best of our knowledge, the complexity of the problem has never been established. We show that the problem is indeed #P'-Complete[1], hence as least as hard as NP-Complete problems. This important result holds true for both transient and fail-stop failures, and constitutes the major contribution of the paper.

Another contribution of the paper is to provide a new approach to approximate the reliability of a particular class of schedules, which we call *strict* schedules. Strict schedules obey a simple rule, called *replication for reliability* in [7]: if there is a dependence edge from task t to task t' in the task graph, then all replicas of t must complete their executions before any replica of t' can start its execution. This rule restricts the graph of replicas to a serial parallel graph, which greatly simplifies the analysis of the reliability in the case of transient failures. However, we are not aware of polynomial cost techniques to compute the reliability of strict schedules in the presence of fail-stop failures.

The paper is organized as follows. We first present the models in Section 2, together with a little example to help understand the difficulty to compute the reliability of a schedule. The core contribution, namely the #P'-Completeness of reliability evaluation, is presented in Section 3. Section 4 is devoted to our results for strict schedules. We discuss related work in Section 5. Finally, we give some conclusions and perspectives in Section 6.

2 Framework

This section first details the application, platform and failure models. Next we introduce schedules with replication, and detail formulas to express their reliability. We conclude with an illustrative example showing the combinatorial nature of reliability computations. Note that Table 1 summarizes all notations used in this section.

2.1 Application and platform

The application and platform model is quite simple and borrowed from the scheduling literature [4]. The application is represented by a directed acyclic graph (or DAG) $G = (T, E)$, where T is the set of tasks to be executed, and E is the set of dependence edges between the tasks. We let $n = |T|$ be the number of tasks, and we number the tasks $t_i \in T$, $1 \leq i \leq n$, according to some topological order (which means that if $(t_i, t_j) \in E$ then $i < j$). For convenience, we assume that there is a source task t_1 and a sink task t_n.

The target platform consists of a set P of m heterogeneous processors p_j, $1 \leq j \leq m$. The execution of task t_i on processor p_j requires w_i^j time-units. It is often assumed that $w_i^j = c_i \times \tau_j$, where c_i is the cost of task t_i and τ_j is the cycle-time of processor p_j (we then speak of uniform machines). We do not

[1] The complexity class #P' is a natural extension of #P, the class of counting problems corresponding to NP decision problems [2].

enforce this restriction and deal with arbitrary execution costs. But without loss of generality, we assume that all execution times w_i^j are integers, so that time-steps are natural numbers (we can always scale rational values).

2.2 Failure models

Processors are subject to failures during the execution of the tasks that are assigned to them. There are two main categories of failures which may occur during the execution of a task t on a processor p:

Transient A transient failure will cause the execution of t to fail, but processor p will be available to execute other tasks (the next tasks assigned to it by the scheduler, if any). In other words, p will be able to contribute to the rest of the execution after the transient failure.

Fail-stop A fail-stop failure is an unrecoverable failure that causes the processor to die until the end of the execution of the whole application: all subsequent tasks assigned to it will not be executed.

Each failure category applies to well-identified scenarios. Transient failures correspond to arithmetic/software errors or recoverable hardware faults (power loss). Fail-stop failures correspond to hardware resource crashes, or to the recovery of a loaned machine by his/her user during a cycle-stealing episode.

We consider various probability failure distributions. The most general distribution is that processor p_j fails at time t, (*i.e.*, during time-interval $[t, t+1[$) with probability $d_j(t)$, where $0 \leq d_j(t) \leq 1$ are arbitrary numbers. Note that we do not always enforce the condition $\sum_{t=0}^{+\infty} d_j(t) = 1$, although it often holds for fail-stop failures, because it is assumed that every processor will fail eventually. For transient failures, it is often assumed that the failure probability is related to, or even proportional to, the duration of the task to be executed.

For instance, a widely used distribution for failed-stop failures is the Poisson process. The probability that p_j fails in the interval $[0, u[$ is $\sum_{t=0}^{u-1} d_j(t) = 1 - \exp^{-\lambda_j \times u}$. Here, λ_j is the (constant) failure rate of p_j, and we do have $\sum_{t=0}^{+\infty} d_j(t) = 1$. On the contrary, for transient failures, when executing a task t_i on p_j, it is natural to let the failure probability depend on the task duration w_i^j. If we further assume that the failure probability does not depend upon the start-up time S_i^j of the task, but only upon its duration, we can have a Poisson process again, with $\sum_{t=S_i^j}^{C_i^j-1} d_j(t) = 1 - \exp^{-\lambda_j w_i^j}$. Most of our results apply for general distributions, where the failure probabilities are arbitrary rational numbers.

Without loss of generality, we assume failures to happen only during task computations, and not during processor idle times. This assumption has an impact only with fail-stop failures, and the probability of failure during an idle time can always be added to the failure probability of the next scheduled task without modifying the reliability of the schedules.

Finally, processor failures, either transient or fail-stop, are always supposed to be independent, regardless of the distribution laws that they follow.

2.3 Schedules with task replication

The objective is to execute the application onto the platform defined above. The schedule assigns tasks to processors. Without replication, each task is assigned to a single processor, with the schedule defining the start-up and completion times of each task onto its assigned processor. However, to remedy the effect of failures, the scheduler may replicate the execution of the tasks onto different processors: if one task fails on a given processor, it is hoped that it will execute successfully on another processor, thereby enabling the rest of the application to proceed despite the failure.

A schedule is thus defined as a one-to-many function which maps each task onto a subset of processors, each of them executing one replica of the task. For each replica, we record a triple of values: the processor

Notation	Definition
$T = \{t_i : i \in [1..n]\}$	set of tasks
n	number of tasks
$G = (T, E)$	directed acyclic graph with tasks and precedence constraints
Pred(t_i)	set of direct predecessors of task t_i
$P = \{p_j : j \in [1..m]\}$	set of processors
m	number of processors
$\pi : T \longrightarrow 2^{P \times \mathbb{N} \times [0..1]}$	schedule defining the processors, start-up times and failure probabilities of each task
t_i^j	replica of task t_i assigned to processor p_j
S_i^j	start-up time of t_i^j (undefined if not scheduled)
w_i^j	execution time of t_i^j
C_i^j	completion time of t_i^j (0 if not scheduled)
$C_{\max}(\pi) = \max_j C_n^j$	makespan of schedule π
rel(π)	reliability of schedule π

Table 1: List of notations.

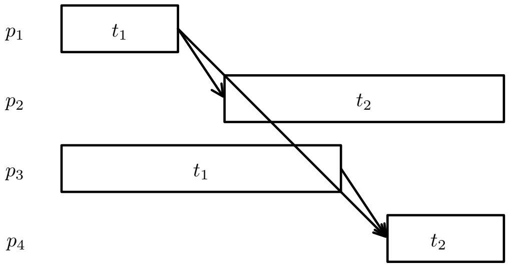


Figure 1: Possible execution with a general schedule.

number, the start-up time and the failure probability. Formally, $\pi : T \longrightarrow 2^{P \times \mathbb{N} \times [0..1]}$ maps every task on a set of such triples.

Let t_i^j be the replica of task t_i on processor p_j (if it exists). The completion time of t_i^j is $C_i^j = S_i^j + w_i^j$. By convention, if t_i is not assigned to p_j, we let $C_i^j = 0$ (and leave S_i^j undefined). Also, it is not authorized to schedule twice the same task onto a given processor.

The schedule must enforce dependence constraints. Without replication, there is no choice: if there is a dependence from task t_i to task $t_{i'}$, *i.e.*, if $(t_i, t_{i'}) \in E$, then the schedule must enforce that $t_{i'}$ cannot start before t_i completes: $C_i \leq S_{i'}$. When several copies of the same task are executed, there are two possible rules:

Strict schedule A task cannot start before all copies of each predecessor have completed.

General schedule A task can start as soon as one replica of each predecessor has completed.

Obviously, strict schedules are a particular case of general schedules. Although they are less general, they are simpler to analyze, at least for transient failures (see Section 4.1).

It is important to point out that the above definitions apply to a failure-free execution. The start-up and completion times of all tasks are deterministic and known in advance from the schedule definition, before the execution starts. Failures may happen randomly during the execution. See the example in Figure 1: with a general schedule, t_2^2, the replica of t_2 on p_2, can start as soon as t_1^1, the replica of t_1 on p_1, has completed:

there is no need to wait for the completion of the other replica t_1^3 of t_1 on p_3. However, if t_1^1 fails during execution, then t_2^2 becomes useless.

Now for each dependence edge $(t_i, t_{i'}) \in E$ and for each processor pair $(p_j, p_{j'}) \in P^2$, there are two cases:

- the completion time C_i^j of the replica t_i^j of t_i is not larger than the start-up time $S_{i'}^{j'}$ of the replica $t_{i'}^{j'}$ of $t_{i'}$. In that case, we say that the replica pair $(t_i^j, t_{i'}^{j'})$ is valid.
- the completion time C_i^j of the replica t_i^j of t_i is larger than the start-up time $S_{i'}^{j'}$ of the replica $t_{i'}^{j'}$ of $t_{i'}$. In that case, we say that the replica pair $(t_i^j, t_{i'}^{j'})$ is not valid.

For a strict schedule, all replica pairs must be valid for every precedence edge in the task graph. For a general schedule, this constraint is not enforced. However, at least one replica pair must be valid for every precedence edge in the graph: if it is not the case, the schedule will never be able to complete its execution, even without any failure. Intuitively, we expect strict schedules to be more *reliable* than general schedules:

- for a strict schedule, a task will be able to start execution if and only if at least one replica of each of its predecessors has successfully executed.
- for a general schedule, the range of admissible predecessor copies is restricted to those whose completion time is not later than the task start-up time.

However, the total execution time, or *makespan*, is likely to be smaller for general schedules than for strict schedules, because there are fewer dependence pairs that are accounted for, hence fewer predecessor copies to wait for. Recall that the makespan $C_{\max}(\pi)$ of a schedule π is formally defined as the completion time of the last replica of the sink task t_n, (*i.e.*, $C_{\max}(\pi) = \max_j C_n^j$.

2.4 Reliability

The reliability $\text{rel}(\pi)$ of a schedule π is defined as the probability that the schedule is *successful*, *i.e.*, succeeds to complete its execution. A strict schedule is easily checked to be successful if and only if at least one replica of each task completes its execution. Determining whether a general schedule is successful is more complicated: we traverse the DAG to check whether the execution of each replica is successful or not. More precisely, a replica $t_{i'}^{j'}$ of a task $t_{i'}$ is successful if and only if
- $p_{j'}$ does not fail during the execution of $t_{i'}^{j'}$, and
- for each predecessor $t_i \in \text{Pred}(t_{i'})$ (if any), there exists at least one valid replica pair $(t_i^j, t_{i'}^{j'})$ such that t_i^j is successful. Finally, a general schedule is successful if at least one replica of the sink task t_n is successful (because it induces that each task has been successfully computed at least once). Formally, we have the following definition:

Definition 1 (Reliability). *Let π be a schedule:*

- *E_{ij} denotes the event that processor p_j does not fail during the execution of t_i^j. With transient failures, this simply means that p_j does not fail between the start-up and completion times of t_i^j, while with fail-stop failures this means that p_j does not fail from the beginning of the schedule until the end of execution of t_i^j. Note that this event is necessary but not sufficient for replica t_i^j to be successful: a valid replica of each predecessor of t_i must have been successfully executed too. Let $\Pr[E_{ij}] = 0$ if task t_i is not assigned to processor p_j.*
- *F_{ij} denotes the event that replica t_i^j of task t_i is successful. Let $\Pr[F_{ij}] = 0$ if task t_i is not assigned to processor p_j*

- *The reliability $rel(\pi)$ of a strict schedule π is defined as*

$$rel(\pi) \quad = \quad \Pr\left[\bigcap_i \bigcup_j E_{ij}\right] \tag{1}$$

- *The reliability $rel(\pi)$ of a general schedule π is defined recursively as follows:*

$$F_{ij} \quad = \quad E_{ij} \text{ if } \mathit{Pred}(i) = \emptyset \tag{2}$$

$$F_{ij} \quad = \quad \left(\bigcap_{i' \in \mathit{Pred}(i)} \quad \bigcup_{j', C_{i'}^{j'} \leq S_i^j} F_{i'j'} \right) \cap E_{ij} \tag{3}$$

$$rel(\pi) \quad = \quad \Pr\left[\bigcup_j F_{nj}\right] \tag{4}$$

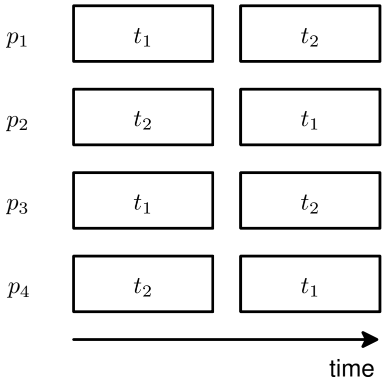
Note that Equation 2 only applies to $i = 1$, as t_1 is a unique source task. Note also that in Equation 3 the set of predecessor copies has been restrained to valid replica pairs (*i.e.*, to any predecessor copy $t_{i'}^{j'}$ such that $C_{i'}^{j'} \leq S_i^j$). This illustrates the difference with strict schedules, for which the F_{ij}, whose values are $E_{ij} = \left(\bigcap_{i' \in \text{Pred}(i)} \bigcup_{j'} E_{i'j'}\right) \cap E_{ij}$, are not needed to compute the reliability.

Finally, we point out that edge failures and communication costs can easily be taken into account when evaluating the reliability of a schedule: replace each dependence edge $t_i^j \longrightarrow t_{i'}^{j'}$ by two edges $t_i^j \longrightarrow \text{comm}_{ii'}^{jj'} \longrightarrow t_{i'}^{j'}$, where $\text{comm}_{ii'}^{jj'}$ is a new task whose execution time is the communication cost between two replicas and whose failure probability (either transient or fail-stop) can be freely chosen. In the case of fail-stop failures, each task $\text{comm}_{ii'}^{jj'}$ must be scheduled on a processor of its own. This failure probability is likely to depend upon the communication cost and/or the link failure rate. If $j = j'$ we model failures during memory transfer, while if $j \neq j'$ we represent failures across interconnection links.

2.5 Example

In this section, we deal with a toy example showing the difficulty of computing the reliability with fail-stop failures, even with independent tasks. Note that all schedules are both strict and general in the case of independent tasks, since there are no dependence constraints. Figure 2 illustrates a schedule with two tasks and four processors, which all execute both tasks (but in different orders). Each task is thus replicated four times. For each processor p_j, let:
- P_{1j} denote the probability that it fails during the execution of its first replica,
- P_{2j} denote the probability that its fails during the execution of its second replica, and
- P_{3j} denote the probability that p_j does not fail before the completion of both replicas. The direct approach to evaluate the schedule reliability consists in forming all the products $P_{a1}P_{b2}P_{c3}P_{d4}$ with $a, b, c, d \in \{1, 2, 3\}^4$. Each product is the probability that a specific execution scenario occurs, and all these scenarios are distinct. Therefore, we can add the terms corresponding to successful scenarios for computing the reliability of the schedule. For instance, $P_{11}P_{22}P_{23}P_{14}$ is the probability that p_1 and p_2 fail while computing their replicas of t_1 and p_3 and p_4 fail while computing their replicas of t_2. This scenario is actually successful as t_2 is computed by p_2 and t_1 by p_3.

The table in Figure 2 shows the formulas obtained with this approach. Each formula defines the reliability when only the subset of processors defined in the first column is used. We remark that the number of terms grows exponentially with the number of processors.



	Reliability (rel(π))
p_1	P_{31}
p_1, p_2	$P_{11}P_{32} + P_{21}(P_{22} + P_{32}) + P_{31}$
p_1, p_2, p_3	$P_{11}(P_{12}P_{33} + P_{22}(P_{23} + P_{33}) + P_{32}) + P_{21}(P_{12}P_{33} + P_{22} + P_{32}) + P_{31}$
p_1, p_2, p_3, p_4	$P_{11}(P_{12}(P_{13}P_{34} + P_{23}(P_{24} + P_{34}) + P_{33}) + P_{22}(P_{13}P_{34} + P_{23} + P_{33}) + P_{32}) + P_{21}(P_{12}((P_{13} + P_{23})(P_{24} + P_{34}) + P_{33}) + P_{22} + P_{32}) + P_{31}$

Figure 2: Schedule with 2 independent tasks on 4 processors.

3 Complexity of general schedules

In this section, we show that computing the reliability of a general schedule is a #P'-Complete problem. This holds both for transient and failure-stop failures, and for arbitrary rational failure probabilities (we cannot deal with real numbers when assessing problem complexity).

Informally, Valiant [14] introduced the notions of #P and #P-Completeness to express the hardness of problems that "count the number of solutions". Because counting a number of solutions to a problem is at least as hard as determining if there is at least a solution, #P problems are at least as difficult as their corresponding NP problems. Just as for NP-Completeness, a proof of #P-Completeness consists of two parts: proof of membership in #P, and reduction from a known #P-Hard problem. There is a technical complication with schedule reliability problems, just as with network reliability problems [11, 2]: we are dealing with probability values, which are rational numbers in $[0, 1]$, instead of dealing with integers as in [14]. Thus, we follow [2] and establish the #P'-Completeness of the problem. The #P' class is a natural extension of the class #P to deal with rational numbers: it allows us to apply a polynomial function on the #P integer output, producing in our case a rational value. We refer the reader to [2] for further details on the #P' complexity class. We are ready to state the main result:

Theorem 1. *Computing the reliability of a general schedule is #P'-Complete.*

The proof of Theorem 1 follows from Propositions 1 and 2 below. First, we precisely define the evaluation problem in Definition 2. Proposition 1 states the membership of the problem in the class #P'. Proposition 2 establishes the completeness; its proof shows that Theorem 1 holds for both transient and fail-stop failures.

Definition 2. SCHEDULE-RELIABILITY *is the problem of computing the reliability rel*(π) *of a general schedule* π *as defined by Equations 2 to 4, for arbitrary rational values of* $\Pr[E_{ij}]$.

Proposition 1. SCHEDULE-RELIABILITY *is in #P'.*

Proof. In order to prove that the problem belongs to #P', we need to characterize the NP decision problem and the transformation for generating the output probability, which is a rational number.

The success probabilities of each processor p_j while computing each task t_i are all assumed to be encoded as $\frac{n_{ij}}{d_{ij}}$. A vector $x_i = (x_{ij})_{1 \leq j \leq m}$ specifies the success of each processor p_j when computing task t_i. If $1 \leq x_{ij} \leq n_{ij}$, then p_j does not fail while computing t_i. If $n_{ij} < x_{ij} \leq d_{ij}$, then p_j fails while computing t_i. Otherwise, t_i is not assigned to p_j and $x_{ij} = 0$ ($d_{ij} = 1$ and n_{ij} is left undefined in this case).

The NP decision problem is the following: given a schedule, does there exist a vector x_i such that the schedule terminates successfully? This problem belongs to NP because vector $x = (x_i)_{1 \leq i \leq n}$ of size $O(nm)$ constitutes the certificate. We check whether a vector x encodes a successful schedule execution by building a schedule containing only tasks without failures. If such a schedule is valid, namely, if all precedence constraints are respected and if all tasks are correctly computed (see Section 2.4 for more details on this schedule verification procedure), then x encodes a successful schedule execution.

The corresponding #P problem consists in computing how many distinct vectors x give successful schedules. In other words, there are $\prod_{i=1}^{n} \prod_{j=1}^{m} d_{ij}$ distinct vectors and each of them defines a possible scenario for the schedule execution. Because all scenarios are equiprobable, we obtain the reliability of a schedule by dividing the number of successful scenarios by the total number of scenarios.

□

In the proof of Proposition 1, the NP problem from which is derived the #P' problem has polynomial complexity: a solution exists if and only if the schedule execution is successful when no replica fails, *i.e.*, when letting $x_{ij} = 1$ for $1 \leq j \leq m$ and for each task t_i assigned to p_j. Yet, as shown below, the reliability problem is #P'-Complete, which is a rare occurrence in the literature. Indeed, the first counting problem corresponding to an easy P problem is the perfect matching counting problem studied in [14].

To prove #P'-Completeness, we use a reduction from CONNECTED.

Definition 3 (CONNECTED). *Given a DAG $G = (A, B)$ with edges subject to failures with rational independent probabilities, compute the probability that the source and the sink nodes are joined by a path of non-failing edges.*

Computing the probability that two arbitrary nodes from A, s and t, are joined is #P'-Complete [11, Problem 10 and Section 3]. We reduce this problem to CONNECTED by inserting two vertexes: in the DAG. The first is connected to all the vertexes in A such that it is the source. Each inserted edge never succeeds except the edge from the source to s, which always succeeds. Analogously, the second inserted vertex is the sink and only the edge from t to the sink succeeds. Moreover, rather than considering the same probability of failure for each edge, we allow arbitrary probabilities. Hence, the probability that s and t are joined is the output of the obtained CONNECTED instance. We prove therefore that CONNECTED is #P'-Complete.

In order to ease the reduction, we slightly transform this problem and provide some formal notations. Without loss of generality, assume that there is a source node and a sink node in the DAG. First, we move from an edge-failing problem to a vertex-failing problem. These vertices will correspond to scheduled tasks in the reduction. Each edge (i, j) from vertex i to vertex j is replaced by a new vertex k, and by two edges, (i, k) from i to k, and (k, j) from k to j. The failure probability of (i, j) is transferred to the new vertex k. All original vertices never fail, hence, the probability that the source and the sink are joined is identical to that in the original DAG.

There are $n = |A| + |B|$ vertices in the new DAG, which we number according to a topological ordering (hence, 1 is the source node and n is the sink node). Moreover, any generated graph with this procedure has a special structure, *e.g.*, any vertex has either one successor or its successors have only one predecessor. Let H_i be the event that the vertex i is valid (does not fail). As already mentioned, the success probability of each $|A|$ vertex already present in the original DAG is equal to 1. Let I_i be the event that there is a path between the source node and node i. Evaluating the reliability in the CONNECTED problem requires to compute $\Pr[I_n]$, where $I_1 = H_1$ and I_i is defined recursively for $i > 1$ as

$$I_i \quad = \bigcup_{i' \in \text{Pred}(i)} I_{i'} \cap H_{i'} \tag{5}$$

The relation between this formulation of I_i and the definition of F_{ij} (Equation 3) is obtained using Morgan's law $\overline{X \cup Y} = \overline{X} \cap \overline{Y}$. Algorithm 1 describes how to reduce any instance of Connected into an instance of Schedule-Reliability. A task is created for each vertex of Connected. The execution time of each task replica on each processor is equal to 1. Each task is scheduled on a processor with success probability equal to the probability that the corresponding vertex in the Connected instance fails. The success probability of the Connected instance will be shown to be equal to the failure probability of the schedule created by Algorithm 1. In fact, the schedule succeeds (no successful path in Connected) if some subset of tasks succeed on their specific processors (some subset of edges fail in Connected). If the schedule is globally successful, then no path is successful in the Connected instance.

Algorithm 1: Reduction of a Connected instance into a Schedule-Reliability instance

```
partition the vertices into levels with a breadth-first search $L = (L_0, L_1, L_2, \ldots)$
time = 0
forall $l \in L$ do
    forall $i \in l$ do
        if $Pred(i) \neq \emptyset$ then
            $\pi(i) = \{(p_{\text{prop}}, \text{time}, 0)\}$
            time = time + 1
    forall $i \in l$ do
        forall $i' \in Pred(i)$ do
            if $i'$ is not scheduled on $p_{sat}$ then
                $\pi(i') = \pi(i') \cup \{(p_{\text{sat}}, \text{time}, 0)\}$
                time = time + 1
    forall $i \in l$ do
        $\pi(i) = \pi(i) \cup \{(p_i, \text{time}, 1 - \Pr[H_i])\}$
    time = time + 1
Simplify_schedule();
```

Algorithm 1 starts by grouping vertices into several levels through a breadth-first search.All the vertices at depth i are put in the i-th level. We remind that the structure of the graph is particular because it results from a transformation (from an edge-failing problem in a vertex-failing problem). Hence, all successors of the predecessors of the vertices in a level l are in l. This property is used in Lemma 1. Then, a task is created for each vertex of Connected and is scheduled three times, except for the sink and source vertices which have only two replicas. After building the schedule, we proceed to an optional simplification of the schedule, where replica with zero probability are removed as well as replica where at least one predecessor is not schedule before this task (see Fig. 3b for an example of such simplification).

The *propagate* processor, p_{prop}, and the *satisfy* processor, p_{sat}, play a special role and execute all tasks except the source and sink. These processors never fail. Each task is also scheduled on a specific processor whose index is the same as the task index (*i.e* task t_i is mapped on processor p_i).

Intuitively, the role of processor p_{prop}, is to "propagate" the success of one task to its successors. This notion of "propagation" is best understood in the Connected instance: one edge might be successful, yet unreachable, in which case the failure of its ancestors must be "propagated" to it. Keeping track of failures (successes in the schedule) is mandatory for the reduction to be effective. Initially in Algorithm 1, the precedence constraints for the replicas on p_{prop} cannot be satisfied by the replicas scheduled on the processors p_{prop} or p_{sat}. But anytime a task t succeeds on its specific processor, the precedence constraints between t and its successors scheduled on p_{prop} are satisfied. If all the precedence constraint of these successors are satisfied, then they are successfully executed on p_{prop}. The success of a task is therefore "propagated" to its successors, which may succeed even if their replicas scheduled on their specific processors fail. Here, the key idea lies in the fact that each task scheduled on its specific processor finishes before that any of its successors scheduled on p_{prop} starts.

Moreover, with processor p_{sat}, the precedence constraints of each task scheduled on its specific processor are satisfied. Indeed, we want tasks scheduled on their specific processors to succeed independently of their precedence constraints. Therefore, all the ancestors of a task t_i must succeed before time S_i^i. Processor p_{sat} plays this role by successfully computing each task in a topological order. Remark that two distinct fully reliable processors are used because the model forbids to schedule the same task twice on the same processor.

Figure 3 depicts a schedule obtained with Algorithm 1. The initial DAG of the CONNECTED instance, and its transformation from an edge-failing problem into a vertex-failing problem, are shown in Figure 3(a). The generated schedule is represented in Figure 3(b). In this example, the breadth-first search generates five levels: $\{v_a\}$, $\{v_1, v_2\}$, $\{v_b, v_c\}$, $\{v_3, v_4\}$ and $\{v_d\}$. All the successors of the predecessors of one level are in the same level. For the level L_3, the three steps of the main loop consist in:
- scheduling the tasks of the level, v_3 and v_4, on p_{prop};
- scheduling the predecessors of these tasks, v_b and v_c, on p_{sat};
- scheduling the tasks in L_3 to their specific processors, p_3 and p_4.

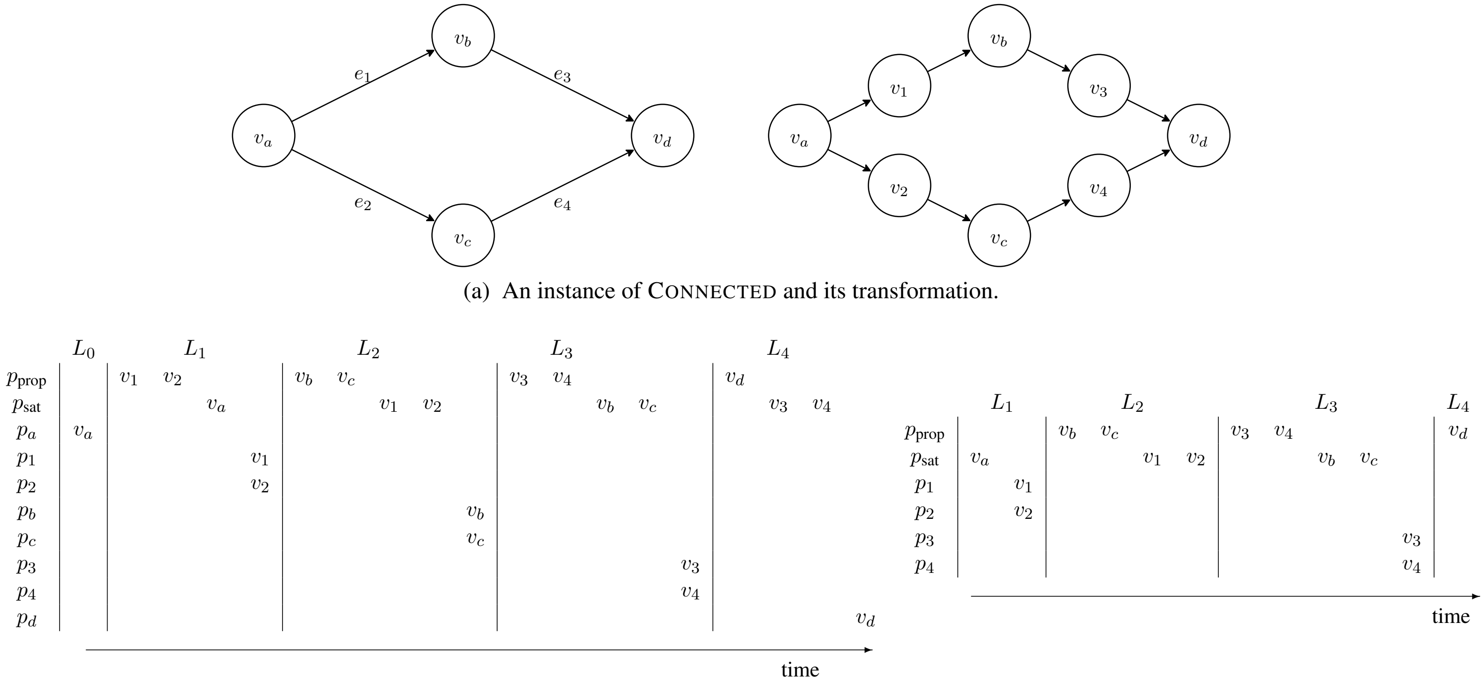
If v_1 is successful on p_1, then v_3 is successful on p_{prop}, which shows the "propagation" of task successes (corresponding to edge failures in the CONNECTED instance). Otherwise, v_3 can still be successfully computed on p_3. In the CONNECTED instance, it is equivalent to state that there is no path from the source to v_d if there is no path neither to v_3 nor to v_4. Finally, we observe that original vertices v_a, v_b, v_c and v_d are not represented in the SCHEDULE-RELIABILITY instance on their specific processor as their success probability is equal to 0.

The proof of completeness relies on the following two lemmas. Lemma 1 proposes a simplification of Equation 3, while Lemma 2 introduces the correspondence between the failure events of the CONNECTED and SCHEDULE-RELIABILITY instances. Informally, Lemma 1 shows that the success of a task on p_{prop} depends on the successes of all its predecessors, either because they succeed on their specific processor or because their replica on p_{prop} is successful. This means that the success of a task (the failure of a path) is "propagated" to its successors. Lemma 2 shows that any task scheduled on its specific processor has all its precedence constraints satisfied due to the replicas scheduled on p_{sat}. The notation concerning the events that will be manipulated is summarized in Table 2.

Symbol	Definition
E_{ij}	event that processor p_j does not fail before the end time of replica t_i^j
F_{ij}	event that replica t_i^j is successfully processed
H_i	event that the node i is valid (for CONNECTED)
I_i	event that a path exists between the source and node i (for CONNECTED)

Table 2: List of notations for Lemmas 1 and 2.

Lemma 1. *Consider a schedule resulting from the reduction from a* CONNECTED *instance using Algorithm 1. Then, the success of any task $t_i \in T$ on processor p_{prop} is given by*



(a) An instance of CONNECTED and its transformation.

(b) Schedule built by Algorithm 1 for the instance (left) and after simplification (right).

Figure 3: Example of the reduction defined by Algorithm 1. There are 4 nodes in the original CONNECTED instance, and $n = 8$ tasks in the schedule.

$$F_{iprop} = \bigcap_{i' \in Pred(i)} F_{i'prop} \cup F_{i'i'}$$

Proof. Using Equation 3, we obtain

$$F_{i\text{prop}} = \left(\bigcap_{i' \in \text{Pred}(i)} \bigcup_{j', C_{i'}^{j'} \leq S_i^{\text{prop}}} F_{i'j'} \right) \cap E_{i\text{prop}}.$$

By construction, tasks never fail on p_{prop} or p_{sat} processors. Thus, $E_{i\text{prop}}$ occurs almost surely (with probability 1) and this term can be discarded. We further simplify by expanding the internal union. Each predecessor $t_{i'}$ of a task t_i is scheduled three times: on processor p_{prop}, except for the source; on processor p_{sat}, except for the sink; and on its specific processor. We now characterize which replicas $t_{i'}^{j}$ of the predecessor $t_{i'}$ are completed before t_i starts on p_{prop}, *i.e.*, $C_{i'}^{j} \leq S_i^{\text{prop}}$.

Any task $t \in T$ (except the source) in the k-th level is scheduled on p_{prop} and on its specific processors at the k-th iteration. Thus, when any successor of t in the k'-th level with $k' > k$ is scheduled on p_{prop} at the k'-th iteration, t has already been finished on p_{prop} and on its specific processor. Formally, if $t_{i'}$ is a predecessor of t_i, then $C_{i'}^{\text{prop}} \leq S_i^{\text{prop}}$ and $C_{i'}^{i'} \leq S_i^{\text{prop}}$.

We now show by contradiction that each predecessor of any task t finishes its execution on p_{sat} after that t starts on p_{prop}, *i.e.*, $C_{i'}^{\text{sat}} > S_i^{\text{prop}}$. This allows the expansion of the internal union without considering the success of predecessors scheduled on p_{sat}.

In Algorithm 1, consider a task $t \in T$ whose depth is k. If t is not the source, t is scheduled on p_{prop} (on Line 5) at the k-th iteration because the breadth-first search puts t in the k-th level. Moreover, t is scheduled on p_{sat} (on Line 10) after the k-th iteration because all the successors of t are in the following levels. Now, suppose that t finishes on p_{sat} before that one of its successors t' in the k'-th level, with $k' > k$, start on p_{prop}. Then, t is scheduled on p_{sat} before that t' is scheduled on p_{prop} because task costs and time increments are all unitary. At the k'-th iteration, t' is scheduled on p_{prop} before any task is scheduled on p_{sat}. Therefore, t must have another successor whose depth is lower than k', otherwise t would not be scheduled on p_{sat} before the k'-th iteration. It implies that $k' > k + 1$, *i.e.*, there is one level that contains this other successor between the k-th and the k'-th levels. Thus, t' has a predecessor in the $k' - 1$-th level because the depth of t' is k'. This predecessor cannot be t because t is in the k-th level and $k < k' - 1$. We see that t has two successors, among which t', which has also two predecessors. There are two cases: either t corresponds to a vertex in the original DAG of the CONNECTED instance, or it corresponds to an edge transformed into a vertex. In the first case, it means that t' corresponds to an edge. However, the vertices resulting from the edges have only one predecessor, which contradicts the fact that t' has at least two ones. In the second case, t comes from an edge. But then it should have a single successor, instead of two ones. Therefore, there is no task that finishes on p_{sat} before that one of its successor starts on p_{prop}. □

Lemma 2. *Consider a schedule resulting from the reduction of a* CONNECTED *instance using Algorithm 1. For any task $t_i \in T$, assume that its specific processor succeeds during its execution (E_{ii}) whenever its corresponding vertex in the* CONNECTED *instance fails ($\overline{H_i}$), and reciprocally. Then, each task t_i succeeds on its specific processor if and only if its corresponding vertex in the* CONNECTED *instance fails,* i.e.*, $F_{ii} = \overline{H_i}$.*

Proof. We first prove that all the ancestors of task t_i are scheduled on processor p_{sat} in a topological order. More precisely, we show by induction on the levels, that each task of the first k levels starts on its specific processor after that all its ancestors have been scheduled in a topological order on p_{sat}. The basis for the induction is easily verified for $k = 1$. Indeed, the source task does not have any ancestor, therefore it is true. Now, assume the induction hypothesis to be true for a given k. Let t be a task in the $(k+1)$-th level. At the $(k+1)$-th iteration, t is scheduled on its specific processor (on Line 13) after its predecessors are scheduled on p_{sat} (on Line 10). These predecessors belong to the first k levels. Thus, their ancestors are scheduled in a topological order on p_{sat} during the first k iterations (by induction hypothesis). As task costs and time increments are unitary, tasks scheduled at the $(k + 1)$-th iteration start after that all earlier scheduled tasks have finished. Therefore, the ancestors of the predecessors of t are scheduled in a topological order on p_{sat} and the predecessors of t are scheduled afterwards in an arbitrary order on p_{sat}. As the predecessors of t belongs to the same level, they have the same depth and do not require to be scheduled in any specific order for their precedence constraints to be satisfied. Hence, t starts on its specific processor after all its ancestors have finished on p_{sat}.

As a consequence, all the ancestors of task t_i are scheduled on p_{sat} and finish before that t_i starts on its specific processor, p_i. Additionally, the ancestors of t_i succeed with probability 1 because tasks scheduled on p_{sat} never fail (Line 10). Thus, when t_i starts its execution on p_i, all its precedence constraints are almost surely satisfied. Moreover, there is no other task scheduled on p_i. Therefore, the success of t_i depends only on its execution, *i.e.*, $F_{ii} = E_{ii} = \overline{H_i}$. □

Proposition 2. SCHEDULE-RELIABILITY *is #P'-Complete.*

Proof. The goal is to show that the success probability of an arbitrary CONNECTED instance is equal to the failure probability of the SCHEDULE-RELIABILITY instance obtained with the reduction of Algorithm 1. More precisely, we want to prove that $I_n = \overline{\bigcup_j F_{nj}} = \overline{F_{n\text{prop}}}$ (recall that task n fails almost surely on its specific processor, and that it is not scheduled on processor p_{sat}). Using Lemma 1, we have directly

$$F_{i\text{prop}} = \bigcap_{i' \in \text{Pred}(i)} F_{i'\text{prop}} \cup F_{i'i'}$$

The assumption of Lemma 2 holds by construction: all events H_i are independent, all events E_{ij} are indeed independent, and the probabilities are identical, *i.e.*, $\Pr[E_{ii}] = 1 - \Pr[H_i]$ for all i. When applying Lemma 2, the relation between Equations 3 and 5 becomes clearer: indeed, we obtain

$$\overline{F_{i\text{prop}}} = \bigcup_{i' \in \text{Pred}(i)} \overline{F_{i'\text{prop}}} \cap H_{i'}$$

We proceed by induction and show that for each task t_i, the success of vertex i is equivalent to the failure of t_i on processor p_{prop}, *i.e.* $\forall i, I_i = \overline{F_{i\text{prop}}}$. For $i = 1$, the source node is not scheduled on processor p_{prop} because it does not have any predecessor. Hence, $F_{i\text{prop}}$ never occurs. On the other hand, the source vertex is present in the original CONNECTED instance and always succeeds, implying that I_i always occurs. Therefore, the basis of the induction is verified.

Without loss of generality, assume that tasks are sorted in a topological order. For a task t_i, we suppose that $I_{i'} = \overline{F_{i'\text{prop}}}$ is true for $1 \leq i' \leq i$. Let us show that it is also true for $k = i + 1$. We have $I_k = \bigcup_{k' \in \text{Pred}(k)} I_{k'} \cap H_{k'}$ and $\overline{F_{k\text{prop}}} = \bigcup_{k' \in \text{Pred}(k)} \overline{F_{k'\text{prop}}} \cap H_{k'}$. Since tasks are traversed in a topological order, $k' < k \leq i$, and by induction hypothesis, $I_{k'} = \overline{F_{k'\text{prop}}}$. We derive that $I_k = \overline{F_{k\text{prop}}}$, as desired.

We have shown that the reduction algorithm is correct. Assessing its space polynomiality is done by counting the number of processors used, the number of replicas scheduled and the space required to store the probabilities. The algorithm schedules each of the $n = |A| + |B|$ tasks at most three times on $n + 2$ processors. Probabilities are computed and stored through a basic arithmetic operation ($y \leftarrow 1 - x$). For the time complexity, the number of calls to Lines 5 and 13 are linear in n. Finally, using an adequate data structure, the condition on Line 9 can be checked in constant time, and Line 10 is called a number of times linear in n. This concludes the whole proof. □

Proof. The goal is to show that the success probability of an arbitrary CONNECTED instance is equal to the failure probability of the SCHEDULE-RELIABILITY instance obtained with the reduction of Algorithm 1. More precisely, we want to prove that $I_n = \overline{\bigcup_j F_{nj}} = \overline{F_{n\text{prop}}}$ (recall that task t_n fails almost surely on its specific processor, and that it is not scheduled on processor p_{sat}).

Without loss of generality, assume that tasks are sorted in a topological order. We proceed by induction and show that for each task t_k, the success of vertex k is equivalent to the failure of t_k on processor p_{prop}, *i.e.*, $\forall k, I_k = \overline{F_{k\text{prop}}}$.

For $k = 1$, the source node is not scheduled on processor p_{prop} because it does not have any predecessor. Hence, $F_{k\text{prop}}$ never occurs. On the other hand, the source vertex is present in the original CONNECTED instance and always succeeds, implying that I_k always occurs. Hence, $I_1 = \overline{F_{1\text{prop}}}$.

For an arbitrary $1 < k \leq n$, let us assume that it is true for all $1 \leq i < k$. We have:

$$\begin{aligned}
I_k &= \textstyle\bigcup_{k' \in \text{Pred}(k)} I_{k'} \cap H_{k'} && \text{by Equation 5} \\
&= \textstyle\bigcup_{k' \in \text{Pred}(k)} \overline{F_{k'\text{prop}}} \cap H_{k'} && \text{by induction hypothesis} \\
&= \textstyle\bigcup_{k' \in \text{Pred}(k)} \overline{F_{k'\text{prop}}} \cap \overline{F_{k'k'}} && \text{by Lemma 2} \\
&= \overline{\textstyle\bigcap_{k' \in \text{Pred}(k)} F_{k'\text{prop}} \cup F_{k'k'}} && \text{by Morgan's Law} \\
&= \overline{F_{k\text{prop}}} && \text{by Lemma 1}
\end{aligned}$$

We have shown that the reduction algorithm is correct. Assessing its space polynomiality is done by counting the number of processors used, the number of replicas scheduled and the space required to store the probabilities. The algorithm schedules each of the $n = |A| + |B|$ tasks at most three times on $n + 2$ processors. Probabilities are computed and stored through a basic arithmetic operation ($y \leftarrow 1 - x$). For the time complexity, the number of calls to Lines 5 and 13 are linear in n. Finally, using an adequate data structure, the condition on Line 9 can be checked in constant time, and Line 10 is called a number of times linear in n. This concludes the whole proof. □

The proof does not depend upon whether failures are transient or fail-stop, hence Theorem 1 is valid for any general schedule. Also, failure probabilities can be arbitrary rational numbers. Altogether, the previous complexity result is relevant to quite a wide classe of DAG scheduling problems with replication.

Finally, the reduction proof shows that evaluating the reliability of any CONNECTED instance exactly amounts to evaluating the unreliability of the schedule generated by the reduction. We deduce from [11, Problem 10 and Section 3] that approximating the reliability of a general schedule up to an arbitrary quantity ε or to an arbitrary ratio α also is a #P'-Complete problem.

4 Complexity of strict schedules

In this section, we recall that computing the reliability of a strict schedule has polynomial complexity with transient failures. The complexity remains open with fail-stop failures, but we propose an exponential evaluation scheme whose complexity can be lowered as much as necessary if only an estimation of the reliability is required.

4.1 Transient failures

Recall from Section 2.4 that E_{ij} denotes the event that processor p_j does not fail during the execution of its replica t_i^j of task t_i. Equation 1 states that a strict schedule π is successful if and only at least one replica of each task is executed without failure: $\mathrm{rel}(\pi) = \Pr\left[\bigcap_i \bigcup_j E_{ij}\right]$

To evaluate $\mathrm{rel}(\pi)$, we apply Morgan's law $\overline{X \cup Y} = \overline{X} \cap \overline{Y}$ and write $\mathrm{rel}(\pi) = \Pr\left[\bigcap_i \overline{\bigcap_j \overline{E_{ij}}}\right]$ The events $\overline{E_{ij}}$ are independent: they correspond to failures of replicas of the same task on distinct processors. In addition, because the failures are transient, the existence of a successful replica for each task also constitute independent events. We derive that

$$\mathrm{rel}(\pi) = \Pr\left[\bigcap_i \overline{\bigcap_j \overline{E_{ij}}}\right] = \prod_i \left(1 - \prod_j \left(1 - \Pr[E_{ij}]\right)\right) \tag{6}$$

Equation 6 is well known [7]. With Poisson processes, it simplifies further into

$$\mathrm{rel}(\pi) = \prod_i \left(1 - \prod_j \exp^{-\lambda_j w_i^j}\right) = \prod_i \left(1 - \exp^{-\sum_j \lambda_j w_i^j}\right) \tag{7}$$

We retrieve the polynomial complexity of reliability evaluation in the case without replication. For each task t_i, the product $\prod_j \left(1 - \Pr[E_{ij}]\right)$ in Equation 6 reduces to a single term, that corresponding to the processor assigned to the task.

4.2 Fail-stop failures

In this section, we focus on fail-stop failures. While the case without replication still has polynomial complexity, the case with replication is open (to the best of our knowledge). We conjecture that evaluating the reliability of strict schedules has the same complexity as that of general schedules, but we have been unable to prove it.

Equation 1 cannot be directly expanded for evaluating the reliability of a strict schedule with fail-stop failures, because events E_{ij} are no longer independent. This is why we propose an alternative formulation of $\mathrm{rel}(\pi)$ in Equation 8. This latter formulation allows us to obtain a recursive expression for evaluating rel(π). Because the complexity of the evaluation scheme is exponential in the number m of processors, we propose to control this complexity by limiting the scope of the recursive evaluations. The price ot pay is that we have only an estimation of the reliability instead of the exact value.

Let G_i be the event that all tasks with an index lower than i have at least one correct replica. Then, G_0 always occurs and G_i is defined recursively for $i \geq 1$ as

$$G_i = \overline{\bigcap_j \overline{E_{ij}}} \cap G_{i-1} \tag{8}$$

Event G_i occurs if and only if at least one processor $p_j \in P$ does not fail during the execution of its replica t_i^j, and if each of the first $i-1$ tasks has been successfully processed. Because tasks are numbered according to some topological order, all the precedence constraints of t_i are satisfied if G_{i-1} occurs.

We now state that the reliability of the schedule is given by rel$(\pi) = \Pr[G_n]$. In order to obtain useful derivations, we introduce an event $\mathcal{E}$ which is an arbitrary intersection of events $\overline{E_{ij}}$:

$$\begin{aligned}
\Pr\left[G_i \mid \mathcal{E}\right] &= \Pr\left[\overline{\bigcap_j \overline{E_{ij}}} \cap G_{i-1} \mid \mathcal{E}\right] \\
&= \Pr\left[\overline{\bigcap_j \overline{E_{ij}}} \mid G_{i-1} \cap \mathcal{E}\right] \times \Pr\left[G_{i-1} \mid \mathcal{E}\right] \\
&= \left(1 - \Pr\left[\bigcap_j \overline{E_{ij}} \mid G_{i-1} \cap \mathcal{E}\right]\right) \times \Pr\left[G_{i-1} \mid \mathcal{E}\right] \\
&= \left(1 - \frac{\Pr\left[\bigcap_j \overline{E_{ij}} \cap G_{i-1} \mid \mathcal{E}\right]}{\Pr\left[G_{i-1} \mid \mathcal{E}\right]}\right) \times \Pr\left[G_{i-1} \mid \mathcal{E}\right] \\
&= \left(1 - \frac{\Pr\left[G_{i-1} \mid \bigcap_j \overline{E_{ij}} \cap \mathcal{E}\right]}{\Pr\left[G_{i-1} \mid \mathcal{E}\right]} \Pr\left[\bigcap_j \overline{E_{ij}} \mid \mathcal{E}\right]\right) \times \Pr\left[G_{i-1} \mid \mathcal{E}\right] \\
&= \left(1 - \frac{\Pr\left[G_{i-1} \mid \bigcap_j \overline{E_{ij}} \cap \mathcal{E}\right]}{\Pr\left[G_{i-1} \mid \mathcal{E}\right]} \prod_j \Pr\left[\overline{E_{ij}} \mid \mathcal{E}\right]\right) \times \Pr\left[G_{i-1} \mid \mathcal{E}\right]
\end{aligned} \tag{9}$$

The last line is obtained by observing that all the events of the intersection $\bigcap_j \overline{E_{ij}}$ concern distinct processors and are independent. Let $\mathcal{E}' = \bigcap_j \overline{E_{ij}} \cap \mathcal{E}$. Then, the calculation of $\Pr\left[G_i \mid \mathcal{E}\right]$ depends on $\Pr\left[G_{i-1} \mid \mathcal{E}\right]$, $\Pr\left[G_{i-1} \mid \mathcal{E}'\right]$ and on some elementary probabilities, *i.e.*, $\Pr\left[\overline{E_{ij}} \mid \mathcal{E}\right]$. Note that $\Pr[G_0 \mid \mathcal{E}] = 1$ for all $\mathcal{E}$ because G_0 always occurs. Therefore, $\Pr[G_n]$ can be computed recursively.

Before analyzing the complexity of this evaluation method, a mechanism for simplifying intersections of events $\overline{E_{ij}}$ must be introduced. Indeed, any event

$$\mathcal{E} = \overline{E_{ij}} \cap \overline{E_{i'j}} \cap \ldots$$

which is the intersection of at least two events $\overline{E_{\cdot j}}$ concerning the same processor p_j, can be reduced to a more concise definition. Only one event per processor is needed: with fail-stop failures, as soon as a processor has failed, it remains down until the end of the schedule. Hence, the event $\overline{E_{\cdot j}}$ which concerns the first task scheduled on p_j is the only one to be considered for each processor $p_j \in P$. Consequently, we never compute any probability $\Pr[G_i \mid \mathcal{E}]$ where $\mathcal{E}$ is an intersection of more than m events.

The complexity of recursive evaluation is $O(n^{m+1})$. Indeed, there are n events G_i and for each of them, there are $(n+1)^m$ distinct intersections $\mathcal{E}$ (at most m elements, and each may concern any of the n tasks). We propose to control the exponent of the complexity cost by making some estimations. We limit the size of any intersection $\mathcal{E}$ to k events. This is done by removing some of the events $\overline{E_{ij}}$ from $\mathcal{E}$ when the size of the intersection grows too large. Formally, we estimate that any new intersection $\mathcal{E}'$ is equals to the intersection of at most k events among $\bigcap_j \overline{E_{ij}} \cap \mathcal{E}$. Several choices are possible. We either select the remaining k events arbitrarily, or we apply some heuristic procedure. As an example, we may be interested by selecting the subset of size k that gives the lowest probability for $\Pr[G_i \mid \mathcal{E}']$. This heuristic is supported by the bound $\Pr\left[G_i \mid \mathcal{E} \cap \overline{E_{ij}}\right] \leq \Pr[G_i \mid \mathcal{E}]$ and would minimize the error done in the estimation locally. The resulting complexity drops down to $O(n^{k+1})$ with $k \in [0..m]$.

It is worth noting that with $k = 0$, the estimation is a lower bound of $\text{rel}(\pi)$ and is equivalent to Equation 6. It would be misleading to conclude that fail-stop failures lead to more reliable schedules than transient failures, because the definitions of E_{ij} are slightly different for each failure type. Actually, we know that the reliability of a strict schedule with fail-stop failures is upper bounded by the reliability of its equivalent schedule with transient failures. For larger values of k, however, we do not have bounds.

This reformulation of the reliability, and the derivations that follow, still end up with an exponential time estimation scheme. Another approach, still exponential, consists in considering all the possible choices (see the proof of Proposition 1 for further details on this approach). To the best of our knowledge, we are not aware of any polynomial procedure for evaluating the reliability of strict schedules with fail-stop failures (even when restricting the workload to independent tasks or to chain of tasks). We conjecture that this problem is #P'-Complete just as it is the case for general schedules.

5 Related work

In [14], Valiant proves that computing the number of acceptable solutions for the two terminal problem is #P-Complete. In [11] Provan and Ball extend the above result for the case of DAGs, and show that evaluating the probability of success in the two terminal case is #P-Complete (more precisely, #P'-Complete using the terminology of this paper).

Evaluating the reliability of a system is often performed through Reliability Block Diagrams (RBD) [3]. It is assumed in many papers such as [8] that RBD evaluation has exponential cost. However, to the best of our knowledge there is no formal complexity result to support this claim. Since it is clear that CONNECTED can be reduced in polynomial time to RBD evaluation (see Definition 3), we proved in this paper that RBD evaluation is also #P'-Complete. However, in some cases, RDBs may have a special structure that allows for an polynomial evaluation (*e.g.* strict schedules and transient failures).

Scheduling task graphs with the goal of minimizing the makespan and maximizing the reliability, without replication, has been studied in [5, 9]. The case with replication is studied in [8]. However, as the authors focus on *strict* schedules with *transient* failures, they are able to use a polynomial algorithm to evaluate the reliability.

6 Conclusion

Figure 4 summarizes known results on the complexity of reliability evaluation. The major contribution of the paper is the #P'-Completeness of the problem for general schedules, for both failure types. While the strict/fail-stop combination remains open, we have provided bounds, as well as a method to approximate the reliability while limiting evaluation costs.

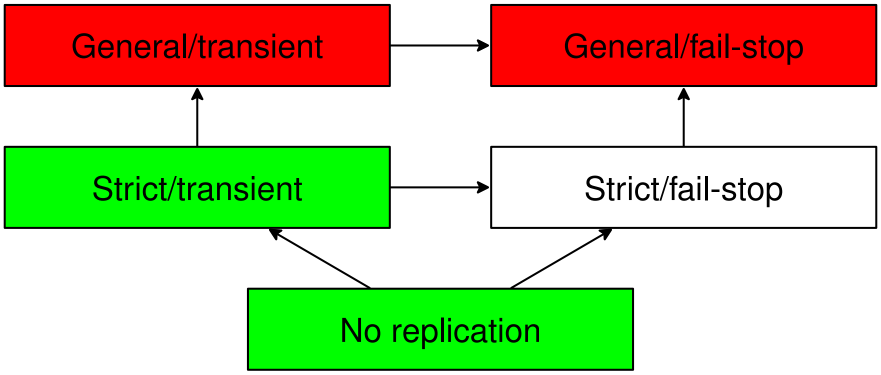


Figure 4: Summary of problem complexities (green for polynomial, white for open, and red for #P'-Complete).

Future work will be devoted to close the complexity gap. We conjecture that the strict/fail-stop combination is #P'-Complete too, but we have been unable to prove it. An important research direction is to provide guaranteed approximations for the general case, with either failure type: can we derive a procedure to approximate the reliability within a prescribed bound while limiting the evaluation cost to some polynomial function of the application/platform parameters?

Acknowledgment

Anne Benoit and Yves Robert are with the Institut Universitaire de France. This work was supported in part by the ANR StochaGrid project and by the Inria ALEAE project.

References

[1] J. Bannister and K. S. Trivedi. Task allocation in fault-tolerant distributed systems. *Acta Informatica*, 20:261–281, 1983.

[2] H. L. Bodlaender, H. L. Bodlaender, T. Wolle, and T. Wolle. A note on the complexity of network reliability problems. *IEEE Trans. Inf. Theory*, 47:1971–1988, 2004.

[3] B. Bream. Reliability Block Diagrams and Reliability Modeling. Technical report, Office of Safety and Mission Assurance, NASA Lewis Research Center, 1995.

[4] P. Brucker. *Scheduling Algorithms*. Springer-Verlag, 2004.

[5] J. J. Dongarra, E. Jeannot, E. Saule, and Z. Shi. Bi-objective Scheduling Algorithms for Optimizing Makespan and Reliability on Heterogeneous Systems. In *19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'07)*, San Diego, CA, USA, June 2007.

[6] A. Girault and H. Kalla. A novel bicriteria scheduling heuristic providing a guaranteed global system failure rate. *IEEE Trans. Dependable Secure Computing*, 2009. To appear, available as `doi.ieeecomputersociety.org/10.1109/TDSC.2008.50`.

[7] A. Girault, E. Saule, and D. Trystram. Reliability versus performance for critical applications. *J. Parallel Distributed Computing*, 69(3):326–336, 2009.

[8] A. Girault, E. Saule, and D. Trystram. Reliability versus performance for critical applications. *Journal of Parallel and Distributed Computing*, 69(3):326–336, Mar. 2009.

[9] E. Jeannot, E. Saule, and D. Trystram. Bi-Objective Approximation Scheme for Makespan and Reliability Optimization on Uniform Parallel Machines. In *The 14th International Euro-Par Conference on Parallel and Distributed Computing (Euro-Par 2008)*, Las Palmas de Gran Canaria, Spain, Aug. 2008.

[10] S. Kartik and C. S. R. Murthy. Task allocation algorithms for maximizing reliability of distributed computing systems. *IEEE Trans. Computers*, 46(6):719–724, 1997.

[11] J. S. Provan and M. O. Ball. The complexity of counting cuts and of computing the probability that a graph is connected. *SIAM J. Computing*, 12(4):777–788, 1983.

[12] S. Shatz and J. Wang. Models & algorithms for reliability-oriented task-allocation in redundant distributed-computer systems. *IEEE Trans. Reliability*, 38(1):16–27, 1989.

[13] S. Shatz, J. Wang, and M. Goto. Task allocation for maximizing reliability of distributed computer systems. *IEEE Trans. Computers*, 41(9):1156–1168, 1992.

[14] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8(3):410–421, 1979.