

Reliability of task graph schedules with transient and fail-stop failures: complexity and algorithms

Anne Benoit, Louis-Claude Canon, Emmanuel Jeannot, Yves Robert

▶ **To cite this version:**

Anne Benoit, Louis-Claude Canon, Emmanuel Jeannot, Yves Robert. Reliability of task graph schedules with transient and fail-stop failures: complexity and algorithms. Journal of Scheduling, 2012, 15 (5), pp.615-627. 10.1007/s10951-011-0236-y . hal-00653477

HAL Id: hal-00653477

https://inria.hal.science/hal-00653477

Submitted on 19 Dec 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Journal of Scheduling manuscript No.
(will be inserted by the editor)

Reliability of task graph schedules with transient and fail-stop failures: complexity and algorithms

Anne Benoit · Louis-Claude Canon · Emmanuel Jeannot · Yves Robert

Received: date / Accepted: date

Abstract This paper deals with the reliability of task graph schedules 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. We fill a complexity gap of the scheduling literature: our main result is that this reliability problem is #P'-Complete (hence at least as hard as NP-Complete problems), both for transient and for fail-stop processor failures. We also study the evaluation of a restricted class of schedules, where a task cannot be scheduled before all replicas of all its predecessors have completed their execution. Although the complexity in this case with fail-stop failures remains open, we provide an algorithm to estimate the reliability while limiting evaluation costs, and we validate this approach through simulations.

Keywords Complexity results, algorithms, task graph schedules, reliability, fail-stop and transient failures.

1 Introduction

Since the landmark papers of Bannister and Trivedi (1983), Shatz et al (1992) and Kartik and Murthy (1997), numerous papers have dealt with reliability issues in task graph scheduling. A natural approach to cope with processor failures is to use redundancy for critical parts of the application (Barlow and Proschan 1967), which in the scheduling framework amounts to replicate the execution of some (or all) tasks. Replication will increase the probability that the execution is successful: only one successful copy of each task is needed when one or several failures take place during the execution. However, one must be able to evaluate the reliability of a given schedule with replication, in order to compare different possible mappings.

We note that at the application level, checkpoint/restart strategies are commonly used as another approach to recover from failures (Cappello et al 2009). Such mechanisms may turn out very costly, depending on the size of the application image, and the number of resources enrolled for execution. In any case, checkpointing is complementary to replication and these techniques do no exclude each other. In both cases, it is necessary to provide an optimized mapping of the application that minimizes the probability of failure.

In this paper, we focus on the problem of computing the reliability of a schedule, *i.e.*, the probability that its execution is successful. More precisely, we are given a schedule that executes an application task graph on a parallel system, and that executes some tasks more than once to achieve redundancy. Moreover, each scheduled task has a known probability of failure. An example of such a schedule is given on Fig. 1.

This problem has been partially addressed in the literature. It is known that if replication is not allowed, then the problem has a polynomial-time algorithm (Dongarra et al 2007; Jeannot et al 2008). A recent paper recognizes that it is difficult to compute the reliability of a schedule with replication, and proposes exponential time algorithms (Girault and Kalla 2009). To the best of our knowledge, the complexity of the problem with replication has never been established. We fill this complexity gap and show that the problem is indeed #P'-Complete, hence at least as hard as NP-Complete problems. The complexity class #P' is a natural extension of

Anne Benoit · Yves Robert
LIP, ENS Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07, France, and Institut Universitaire de France, E-mail: Anne.Benoit@ens-lyon.fr, Yves.Robert@ens-lyon.fr

Louis-Claude Canon
Nancy University, 34 Cours Léopold, CS 25233, 54052 Nancy Cedex, France and INRIA, E-mail: Louis-Claude.Canon@inria.fr

Emmanuel Jeannot
LaBRI and INRIA Bordeaux, 351 Cours de la Libération - Bât. A29b, 33405 Talence Cedex, France, E-mail: Emmanuel.Jeannot@inria.fr

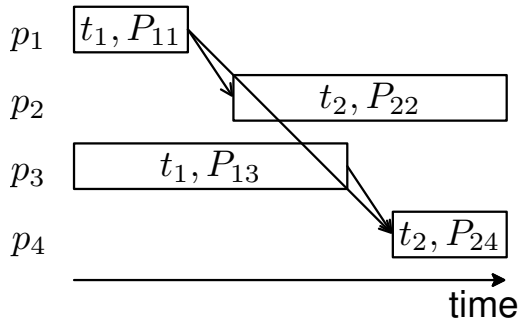


Fig. 1: General schedule of a chain of two tasks (t_1 and t_2) that are duplicated twice each. P_{ij} is the failure probability of task t_i on processor p_j.

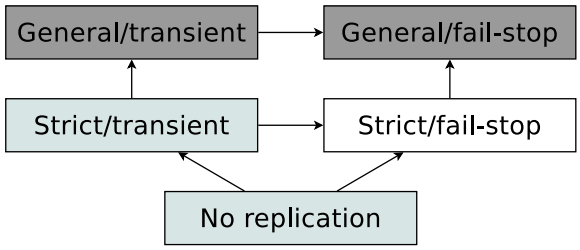


Fig. 2: Summary of complexity results of this paper. White: open problem. Light grey: solvable in polynomial time. Dark grey: #P'-Complete. Arrow: polynomial-time reduction.

#P, the class of counting problems corresponding to NP decision problems (Bodlaender and Wolle 2004), in which we can apply a polynomial function on the #P integer output (we need a rational number for the reliability).

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.

We further explore a particular class of schedules, which we call *strict* schedules. Strict schedules obey a simple rule, called *replication for reliability* in Girault et al (2009): 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. As only one replica of t needs to complete its execution before one of the replicas of t' starts its execution in a feasible schedule, it guarantees more precedence constraints than necessary. Schedules for which this rule is not enforced are called *general*. Fig. 1 is an example of such a general schedule.

With two failure types and two schedule classes, we are led to four variations of the problem. Fig. 2 summarizes known results on the complexity of reliability evaluation for these variations, with the following legend: light grey for polynomial time, white for open, and dark grey for #P'-Complete. An arrow means that the source problem is polynomial-time reducible to the destination problem. The major contribution of the paper is the #P'-Completeness of the problem for general schedules, for both failure types.

Another contribution of the paper is to provide a new approach to estimate the reliability of strict schedules. In the case of transient failures, it is known in the literature that evaluating the reliability is a polynomial-time problem: scheduling task graphs without replication has been studied in Dongarra et al (2007) and Jeannot et al (2008), while the case with replication is studied in Girault et al (2009). However, we are not aware of polynomial-time techniques to compute the reliability of strict schedules in the presence of fail-stop failures. The proposed approach applies to any strict schedule and is empirically validated on random instances.

The paper is organized as follows. We briefly overview related work on #P-Complete problems in Section 2. Then we present the models in Section 3, together with a little example intended to help understand the difficulty of computing the reliability of a schedule. The core contribution, namely the #P'-Completeness of reliability evaluation, is presented in Section 4. Section 5 is devoted to results for strict schedules. Finally, we conclude in Section 6.

2 Related work on #P-Complete problems

In some related work (Valiant 1979), Valiant proves that computing the number of acceptable solutions for the two terminal problem is #P-Complete. The work of Provan and Ball (1983) extends the above result for the case of DAGs, and shows that evaluating the probability of success in the two terminal case is #P'-Complete. However, their result does not imply anything about the complexity of the schedule reliability problem. Furthermore, it is interesting to remark that evaluating the reliability of a system is often performed through Reliability Block Diagrams (RBD) (Bream 1995). It is assumed in many papers such as in Girault et al (2009) that RBD evaluation has an exponential time. However, to the best of our knowledge, there is no formal complexity result to support this claim. Although it is possible to show that RBD evaluation is also #P'-Complete from Provan and Ball (1983), we can easily derive it from our results. However, in some cases, RBDs may have a special structure that allows for an evaluation in polynomial time.

3 Framework

Our main objective is to study the reliability of different types of schedules. First, we formalize the execution model by detailing the application and platform parameters in Section 3.1. Then, we characterize in Section 3.2 the failure model that specifies how processors may fail during the execution of any task. Next, we describe in Section 3.3 the replication mechanism consisting in scheduling some tasks several times. We are then able to provide the detailed formulas used to express the reliability of any schedule depending on its characteristics (Section 3.4). After a short discussion of the complexity represented by communications in this context (Section 3.5), we conclude in Section 3.6 with an example showing the combinatorial nature of reliability computations. All notations are summarized in Table 1.

3.1 Application and platform

The application and platform model is quite simple and borrowed from the scheduling literature (Brucker 2004). 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 unique source task t_1 and a unique 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. Note that 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 enforce this restriction here, and deal with arbitrary execution times. 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).

3.2 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 failures: 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 failures: a fail-stop failure is an unrecoverable failure that causes the processor to be down 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) (Shatz and Wang 1989; Zhu et al 2004). Fail-stop failures correspond to hardware resource crashes, or to the recovery of a loaned machine by her/his user during a cycle-stealing episode (Awerbuch et al 1996; Bhatt et al 1997; Rosenberg 2002).

All our results apply for general distributions, where the failure probabilities are arbitrary rational numbers.

The probability of any fail-stop failure occurring during processor idle times can be transferred to the failure probability of the next scheduled task without modifying the reliability of the schedules. The same idea can be applied if we consider specific transient failures that are undetected when the processor is idle until the next task starts its execution (whose execution would then be unsuccessful). Therefore, without loss of generality, we consider an equivalent model where no failure is assumed to happen during processor idle times.

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

3.3 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 number, the start-up time and the failure probability. Formally, $\pi : T \to 2^{P \times \mathbb{N} \times [0,1]}$ 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). Its start-up time is S_i^j, and its completion time 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, without loss of generality, it is not authorized to schedule twice the same task onto a given processor. This restriction has no impact on our results (scheduling a task

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
$\mathrm{Pred}(t_i)$	set of direct predecessors of task t_i
$P = \{p_j : j \in [1..m]\}$	set of processors
m	number of processors
$\pi : T \to 2^{P \times \mathbb{N} \times [0,1]}$	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
$\mathrm{rel}(\pi)$	reliability of schedule π

Table 1: List of notations.

twice on the same processor is at least as hard) but simplifies the notations (*e.g.*, for the completion times C_i^j).

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: $C_i^j \leq S_{i'}^{j'}$, where t_i is assigned to p_j, and $t_{i'}$ is assigned to $p_{j'}$. 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 5).

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 possible execution with a general schedule on the example in Fig. 1: 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: if 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 $t_{i'}$, we say that the replica pair $(t_i^j, t_{i'}^{j'})$ is *valid*; otherwise, we say that it 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, for each path in the task graph, there must be a list of replicas that correspond to the tasks of the path and such that each replica forms valid replica pairs with its neighbors in the list: 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 been successfully executed,
- while 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 replica 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.

The proposed scheduling mechanism is static: no adjustment is done during the execution, relatively to the replicas that succeed and the failures that occur. As such, failures do not require to be detected (the execution of the schedule is pursued until the end). Although dynamic approaches are more practical in presence of high uncertainty, pro-actively evaluating the reliability of the scheduling decisions is still required. Therefore, static and dynamic approaches are complementary and raise a similar evaluation problem.

3.4 Reliability

Similarly to Barlow and Proschan (1967), we consider the execution of the schedule until its first failure. As stated previously, the failure of one replica may not cause the schedule to fail. The reliability $\mathrm{rel}(\pi)$ 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:

1. $p_{j'}$ does not fail during the execution of $t_{i'}^{j'}$, and
2. for each predecessor $t_i \in \mathrm{Pred}(t_{i'})$ (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).

In order to formally define the reliability, we use several events that denote each a set of outcomes of the sample space. Usual notations of set theory are used to represent disjunction and conjunction (union and intersection, respectively). The following definitions are based on two types of events, which enable a formal and complete proof of our completeness result.

Let π be a schedule. Then

- R_{ij} denotes the event that processor p_j does not fail during the execution of t_i^j, and $\Pr[R_{ij}]$ denotes the probability of this event. 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 $\Pr[R_{ij}] = 0$ if task t_i is not assigned to processor p_j.
- U_{ij} denotes the event that replica t_i^j of task t_i is successful, and $\Pr[U_{ij}]$ denotes the probability of this event. Let $\Pr[U_{ij}] = 0$ if task t_i is not assigned to processor p_j. Otherwise, event U_{ij} is defined as follows:

$$U_{ij} = \begin{cases} R_{ij} & \text{if } \mathrm{Pred}(i) = \emptyset \\ \left(\bigcap_{i' \in \mathrm{Pred}(i)} \bigcup_{j', C_{i'}^{j'} \leq S_i^j} U_{i'j'} \right) \cap R_{ij} & \text{otherwise} \end{cases} \quad (1)$$

  Note that the initialisation only applies to $i = 1$, as t_1 is a unique source task. Note also that the set of predecessor copies has been restrained to valid replica pairs, *i.e.*, to any predecessor copy $t_{i'}^{j'}$ such that $C_{i'}^{j'} \leq S_i^j$.
- The reliability $\mathrm{rel}(\pi)$ of a general schedule π is the probability that at least one replica of the sink task is successful:

$$\mathrm{rel}(\pi) = \Pr\left[\bigcup_j U_{nj} \right]. \quad (2)$$

- The reliability $\mathrm{rel}(\pi)$ of a strict schedule π is the probability that at least one replica of each task is successful and is defined as

$$\mathrm{rel}(\pi) = \Pr\left[\bigcap_i \bigcup_j R_{ij} \right]. \quad (3)$$

  Note that the U_{ij} are not needed to compute the reliability for strict schedules because all replica pairs are valid.

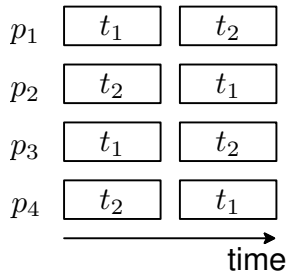
Finally, we assume for simplicity that the schedule is non-preemptive, but the proof of Theorem 1 shows that the #P'-Completeness result holds true for preemptive schedules.

3.5 Communications

We point out that edge failures and communication costs could easily be taken into account when evaluating the reliability of a schedule: replace each dependence edge $t_i^j \to t_{i'}^{j'}$ by two edges $t_i^j \to \mathrm{comm}_{ii'}^{jj'} \to t_{i'}^{j'}$, where $\mathrm{comm}_{ii'}^{jj'}$ is a new task whose execution time is the communication cost between the two replicas, and whose failure probability (either transient or fail-stop) can be freely chosen. In the case of fail-stop failures, each task $\mathrm{comm}_{ii'}^{jj'}$ must be scheduled on a processor of its own. The edge 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; otherwise, we model failures across interconnection links. Altogether, we can deal with communications by adding $|E|$ tasks and processors (where E is the set of edges).

3.6 Example

In this section, we deal with a toy example showing the difficulty of computing the reliability in the presence of 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. Fig. 3 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, P_{1j} denotes the probability that p_j fails during the execution of its first replica; P_{2j} denotes the probability that p_j fails during the execution of its second replica; and P_{3j} denotes 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 $a,b,c,d \in \{1,2,3\}$. 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



	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	$P_{11}(P_{12}(P_{13}P_{34} + P_{23}(P_{24} + P_{34}) + P_{33}) + P_{22}(P_{13}P_{34} + P_{23} + P_{33}) + P_{32}) + P_{21}(P_{12}((P_{13} + P_{23})(P_{24} + P_{34}) + P_{33}) + P_{22} + P_{32}) + P_{31}$

Fig. 3: Schedule with two independent tasks on four processors.

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 Fig. 3 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, and that it does not seem possible to factor the terms into a compact formula.

4 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). We start with a definition of the #P' complexity class and formally state the problem before providing a fully detailed proof, which we decompose into several steps.

4.1 Problem statement

Informally, Valiant (1979) 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. There is a technical complication with schedule reliability problems, just as with network reliability problems (Provan and Ball 1983; Bodlaender and Wolle 2004): we are dealing with probability values, which are rational numbers in $[0,1]$, instead of dealing with integers as in Valiant (1979). Thus, we follow Bodlaender and Wolle (2004) 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 number. The formal definitions are the following:

Definition 1 (Complexity classes) Let Σ be a finite alphabet and Σ^* be the set of all strings over Σ.

- The class #P consists of the functions $f : \Sigma^* \Rightarrow \mathbb{N}$ such that there exists a nondeterministic polynomial-time Turing machine M such that for all inputs $x \in \Sigma^*$, $f(x)$ is the number of accepting paths of M.
- The class #P' consists of the functions $h : \Sigma^* \Rightarrow \Sigma^*$ such that there exists a function $f \in \#P$, $f : \Sigma^* \Rightarrow \mathbb{N}$, and a polynomial-time computable function $g : \mathbb{N} \times \Sigma^* \Rightarrow \Sigma^*$, which satisfy to $\forall x \in \Sigma^*$, $h(x) = g(f(x), x)$.

Definition 2 (SCHEDULE-RELIABILITY) Given a general schedule π, SCHEDULE-RELIABILITY is the problem of computing the reliability rel(π) as defined by Equations (1) and (2), for arbitrary rational failure probabilities $\Pr[R_{ij}]$.

We are ready to state the main result:

Theorem 1 SCHEDULE-RELIABILITY *is #P'-Complete.*

The motivation for introducing the class #P' should be clearer now: the problem is to compute the rational value rel(π) for a general schedule π, rather than the number of successful executions of π for all possible failure scenarios.

The proof of Theorem 1 shows that the result holds for both transient and fail-stop failures. The proof goes just as an NP-Completeness proof: we first prove in Lemma 1 that the problem belongs to the $\#P'$ class (SCHEDULE-RELIABILITY $\in \#P'$), and then we prove its hardness in Lemma 2 by reduction from another #P'-Complete problem, namely the CONNECTED problem, which we define below:

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

CONNECTED is #P'-Complete (Provan and Ball 1983, Problem 10 and Section 3). In fact, it is a slight variant of the two terminal network reliability problem in Provan and Ball (1983): instead of joining two arbitrary nodes of the graph, we join the source and the sink. The reduction from the original problem is straightforward.

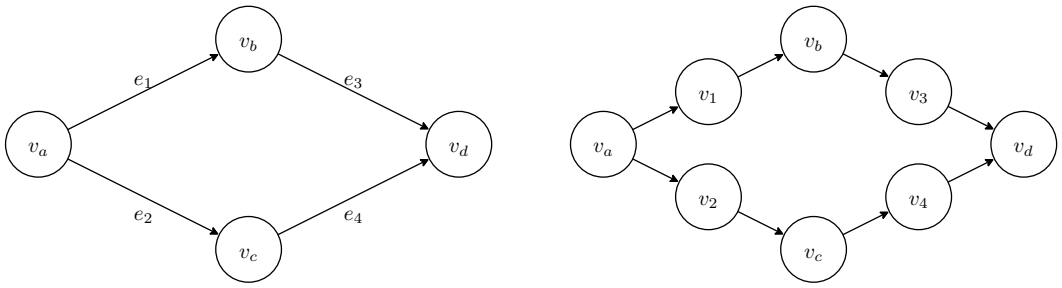


Fig. 4: An instance of CONNECTED and its transformation.

4.2 Class membership

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

Proof In order to prove that the SCHEDULE-RELIABILITY belongs to #P', we need to characterize the underlying 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 the 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 3 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.

This proves the membership of SCHEDULE-RELIABILITY to the #P' complexity class. □

4.3 Completeness

Lemma 2 SCHEDULE-RELIABILITY *is #P'-Hard.*

Proof The proof is rather involved. We start with an arbitrary instance $Inst_1$ of CONNECTED and we build an instance $Inst_2$ of SCHEDULE-RELIABILITY such that the probability that the source and the sink nodes are joined by a path of non-failing edges in $Inst_1$ is equal to the unreliability of the general schedule in $Inst_2$. For better readability, we divide the reduction into several steps.

Step 1*: Transformation of* $Inst_1$. We first transform the DAG $G = (A, B)$ with a source and sink nodes of $Inst_1$, and provide some formal notations. We move from an edge-failing problem to a vertex-failing problem. These vertices will correspond to scheduled tasks in $Inst_2$. Each edge (i, j) in B 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, as illustrated on Fig. 4: the instance of CONNECTED has four nodes, and the transformed instance has eight nodes. 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 G.

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 N_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 V_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 $\Pr[V_n]$, where $V_1 = N_1$ and V_i is defined recursively for $i > 1$ as

$$V_i = \bigcup_{i' \in \mathrm{Pred}(i)} (V_{i'} \cap N_{i'}). \tag{4}$$

Step 2*: Construction of* $Inst_2$. A task is created in $Inst_2$ for each vertex of the transformed version of $Inst_1$. 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

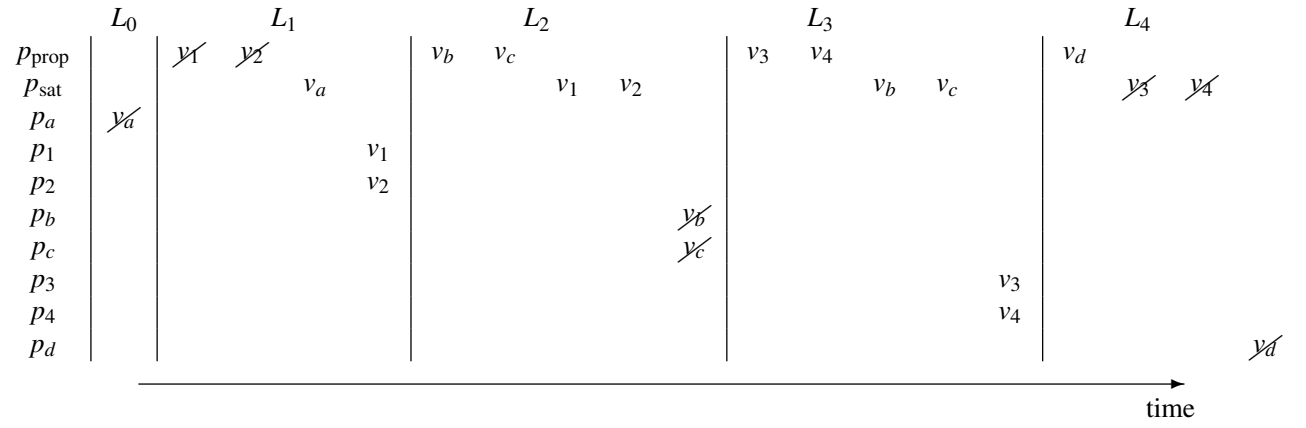


Fig. 5: Schedule built by the reduction algorithm for the instance of Fig. 4. Canceled replicas can be discarded as they do not impact the schedule reliability. It is the case for vertices v_a, v_b, v_c and v_d that are scheduled on their specific processors and all have a zero probability of success.

the failure probability of the schedule created by the reduction algorithm (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 vertices fail in CONNECTED). If the schedule is globally successful, then no path is successful in the CONNECTED instance.

The reduction algorithm 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. 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.

Algorithm 1: Reduction of a CONNECTED instance into a SCHEDULE-RELIABILITY instance

```
1  Partition the vertices into levels with a breadth-first search
   L = (L_0, L_1, L_2, ...)
2  time = 0
3  forall l ∈ L do
4      forall i ∈ l do
5          if Pred(i) ≠ ∅ then
6              π(i) = {(p_prop, time, 0)}
7              time = time + 1
8          else
9              π(i) = ∅
10     forall i ∈ l do
11         forall i' ∈ Pred(i) do
12             if i' is not scheduled on p_sat then
13                 π(i') = π(i') ∪ {(p_sat, time, 0)}
14                 time = time + 1
15     forall i ∈ l do
16         π(i) = π(i) ∪ {(p_i, time, 1 − Pr[N_i])}
17     time = time + 1
```

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 vertex 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, 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 constraints 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. Note that two distinct fully reliable processors are used because the model forbids to schedule the same task twice on the same processor.

In the example of Fig. 5, 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 level L_3, the three steps of the main loop of the reduction algorithm 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}; and 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 on p_3. In both cases, the schedule is successful if v_d succeeds on p_{prop}, *i.e.*, if both v_3 and v_4 are successful. 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.

Step 3*: Equivalence of $Inst_1$ and $Inst_2$.* We now show that the success probability of $Inst_1$ is equal to the failure probability of $Inst_2$. The roadmap is the following. We first propose in Lemma 3 a simplification of the recursive equation (1) defining the reliability of a general schedule, which 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.

Then, we introduce the correspondence between the failure events of the CONNECTED and SCHEDULE-RELIABILITY instances: we show that any task scheduled on its specific processor has all its precedence constraints satisfied due to the replicas scheduled on p_{sat} (see Lemma 4).

Building upon these two lemmas, we can then prove the equivalence of solutions by showing that

$$V_n = \overline{\bigcup_j U_{nj}} = \overline{U_{n\text{prop}}}.$$

Notations concerning the events that will be manipulated are summarized in Table 2.

Step 3.1*: Simplifying probabilities.*

Lemma 3 *Consider the schedule of $Inst_2$. Then, the success of any task $t_i \in T$ on processor p_{prop} is given by $U_{iprop} = \bigcap_{i' \in Pred(i)} (U_{i'prop} \cup U_{i'i'})$.*

Proof Using Equation (1), we obtain

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

By construction, tasks never fail on p_{prop} or p_{sat} processors. Thus, $R_{i\text{prop}}$ 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 processor 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 any task finishes its execution on p_{sat} after that any of its successors 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, then 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 12) 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}. □

Step 3.2*: Correspondence between failure events*

Lemma 4 *Consider the schedule of $Inst_2$. For any task $t_i \in T$, assume that its specific processor succeeds during its execution (R_{ii}) whenever its corresponding vertex in the* CONNECTED *instance fails ($\overline{N_i}$), and reciprocally. Then, each*

Symbol	Definition
R_{ij}	event that processor p_j does not fail before the end time of replica t_i^j
U_{ij}	event that replica t_i^j is successfully processed
N_i	event that the node i is valid (for CONNECTED)
V_i	event that a path exists between the source and node i (for CONNECTED)
$\Pr[X]$	probability of event X

Table 2: List of notations for Lemmas 3 and 4.

task t_i succeeds on its specific processor if and only if its corresponding vertex in the CONNECTED *instance fails,* i.e.*,* $U_{ii} = \overline{N_i}$.

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 $k = 0$. 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 15) after its predecessors are scheduled on p_{sat} (on Line 12). 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 that are not yet scheduled on p_{sat} are scheduled on it in an arbitrary order at the k-th iteration. As these 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 (see Line 12). 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.*, $U_{ii} = R_{ii} = \overline{N_i}$. □

Step 3.3*: Equivalence of both instances.* We now show that the success probability of $Inst_1$ is equal to the failure probability of $Inst_2$. More precisely, we prove that $V_n = \overline{\bigcup_j U_{nj}} = \overline{U_{n\text{prop}}}$ (recall that task t_n fails almost surely on its specific processor, and that it is not scheduled on processor p_{sat}).

The relation between the definitions of V_i (Equation (4)) and U_{ij} (Equation (1) in Section 3) is obtained using Morgan's law $\overline{X \cup Y} = \overline{X} \cap \overline{Y}$ and Lemmas 3 and 4.

Without loss of generality, assume that tasks are sorted in a topological order. 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.*, $\forall i, V_i = \overline{U_{i\text{prop}}}$.

For $i = 1$, the source node is not scheduled on processor p_{prop} because it does not have any predecessor. Hence, $U_{i\text{prop}}$ never occurs. On the other hand, the source vertex is present in the original CONNECTED instance and always succeeds, implying that V_i always occurs. Therefore, the basis of the induction is verified, *i.e.*, $V_1 = \overline{U_{1\text{prop}}}$.

For a task t_i, we suppose that $V_k = \overline{U_{k\text{prop}}}$ is true for $1 \leq k < i$. Let us show that $V_i = \overline{U_{i\text{prop}}}$ is also true.

$$
\begin{aligned}
V_i &= \textstyle\bigcup_{i' \in \text{Pred}(i)} V_{i'} \cap N_{i'} && \text{by Equation (4)} \\
&= \textstyle\bigcup_{i' \in \text{Pred}(i)} \overline{U_{i'\text{prop}}} \cap N_{i'} && \text{by induction hypothesis} \\
&= \textstyle\bigcup_{i' \in \text{Pred}(i)} \overline{U_{i'\text{prop}}} \cap \overline{U_{i'i'}} && \text{by Lemma 4} \\
&= \overline{\textstyle\bigcap_{i' \in \text{Pred}(i)} U_{i'\text{prop}} \cup U_{i'i'}} && \text{by Morgan's Law} \\
&= \overline{U_{i\text{prop}}} && \text{by Lemma 3}
\end{aligned}
$$

In the second line, we have used that $i' < i$ because tasks are traversed in a topological order. In the third line, the assumption of Lemma 4 holds by construction: all events N_i are independent, all events R_{ij} are indeed independent, and the probabilities are identical, *i.e.*, $\Pr[R_{ii}] = 1 - \Pr[N_i]$ for all i.

We have shown that the reduction algorithm is correct. Assessing its space polynomial complexity 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 15 are linear in n. Finally, using an adequate data structure, the condition on Line 11 can be checked in constant time, and Line 12 is called a number of times linear in n. This concludes the proof. □

As already mentioned, the proof of Theorem 1 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. As the proof

only requires processors to be identical and tasks to have unitary costs, the complexity of the problem is related to the DAG structure only. Altogether, the previous complexity result is relevant to quite a wide class of DAG scheduling problems with replication. For instance the result is also valid for preemptive schedules: interrupting the execution of a task may modify the probability of failure of that task, but the proof handles arbitrary failure probabilities.

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 (Provan and Ball 1983, Problem 10 and Section 3) the following corollary:

Corollary 1 *Approximating the reliability of a general schedule up to an arbitrary quantity ε or to an arbitrary ratio α is #P'-Complete.*

5 Complexity of strict schedules

For *transient failures*, a closed-form formula for computing the reliability is provided in Girault et al (2009). This formula can be evaluated in polynomial time, and it can be further simplified in case of Poisson processes.

We focus now on *fail-stop failures*. While the case without replication still has polynomial-time 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. However, we propose an exponential evaluation scheme whose complexity can be lowered as much as necessary, if only an estimation of the reliability is required. Simulation results allow us to assess the effectiveness of this method.

5.1 Evaluation scheme

The equation defining the reliability of a strict schedule (see Section 3, Equation (3)) cannot directly be expanded for evaluating the reliability of a strict schedule with fail-stop failures, because events R_{ij} are no longer independent. This is why we propose an alternative formulation of $\mathrm{rel}(\pi)$ using the event G_i defined as follows.

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 $G_i = \overline{\bigcap_j \overline{R_{ij}}} \cap G_{i-1}$. 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. As tasks are numbered according to some topological order, all the precedence constraints of t_i are satisfied if G_{i-1} occurs.

This latter formulation allows us to obtain a recursive expression for evaluating $\mathrm{rel}(\pi)$. 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 to pay is that we have only an estimation of the reliability instead of the exact value.

We now state that the reliability of the schedule is given by $\mathrm{rel}(\pi) = \Pr[G_n]$. In order to obtain useful derivations, we introduce an event $\mathscr{E}$, which is an arbitrary intersection of events $\overline{R_{ij}}$. Let $\mathscr{E}' = \bigcap_j \overline{R_{ij}} \cap \mathscr{E}$. Then, the calculation of $\Pr[G_i \mid \mathscr{E}]$ depends on $\Pr[G_{i-1} \mid \mathscr{E}]$, $\Pr[G_{i-1} \mid \mathscr{E}']$ and on some elementary probabilities, *i.e.*, $\Pr\left[\overline{R_{ij}} \mid \mathscr{E}\right]$:

$$\Pr[G_i \mid \mathscr{E}] = \Pr\left[\overline{\bigcap_j \overline{R_{ij}}} \cap G_{i-1} \mid \mathscr{E}\right]$$

$$= \Pr\left[\overline{\bigcap_j \overline{R_{ij}}} \mid G_{i-1} \cap \mathscr{E}\right] \times \Pr[G_{i-1} \mid \mathscr{E}]$$

$$= \left(1 - \Pr\left[\bigcap_j \overline{R_{ij}} \mid G_{i-1} \cap \mathscr{E}\right]\right) \times \Pr[G_{i-1} \mid \mathscr{E}]$$

$$= \left(1 - \frac{\Pr\left[\bigcap_j \overline{R_{ij}} \cap G_{i-1} \mid \mathscr{E}\right]}{\Pr[G_{i-1} \mid \mathscr{E}]}\right) \times \Pr[G_{i-1} \mid \mathscr{E}]$$

$$= \left(1 - \frac{\Pr\left[G_{i-1} \mid \bigcap_j \overline{R_{ij}} \cap \mathscr{E}\right]}{\Pr[G_{i-1} \mid \mathscr{E}]} \Pr\left[\bigcap_j \overline{R_{ij}} \mid \mathscr{E}\right]\right) \times \Pr[G_{i-1} \mid \mathscr{E}]$$

$$= \left(1 - \frac{\Pr[G_{i-1} \mid \mathscr{E}']}{\Pr[G_{i-1} \mid \mathscr{E}]} \Pr\left[\bigcap_j \overline{R_{ij}} \mid \mathscr{E}\right]\right) \times \Pr[G_{i-1} \mid \mathscr{E}]$$

$$= \left(1 - \frac{\Pr[G_{i-1} \mid \mathscr{E}']}{\Pr[G_{i-1} \mid \mathscr{E}]} \prod_j \Pr\left[\overline{R_{ij}} \mid \mathscr{E}\right]\right) \times \Pr[G_{i-1} \mid \mathscr{E}]$$

The last line is obtained by observing that all the events of the intersection $\bigcap_j \overline{R_{ij}}$ concern distinct processors and are independent.

Note that $\Pr[G_0 \mid \mathscr{E}] = 1$ for all $\mathscr{E}$ because G_0 always occurs. Therefore, $\Pr[G_n]$ can be computed recursively.

Before analyzing the complexity of this evaluation scheme, we introduce a mechanism for simplifying intersections of events $\overline{R_{ij}}$. Indeed, any event $\mathscr{E} = \overline{R_{ij}} \cap \overline{R_{i'j}} \cap \ldots$ which is the intersection of at least two events $\overline{R_{\cdot j}}$ 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 $\overline{R_{\cdot j}}$ 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 $\Pr[G_i \mid \mathscr{E}]$ where $\mathscr{E}$ 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 $\mathscr{E}$ (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 $\mathscr{E}$ to k events. This is done by removing some of the events $\overline{R_{ij}}$ from $\mathscr{E}$ when the size of the intersection grows too large. Formally, we estimate that any new intersection $\mathscr{E}'$ is equal to the intersection of at most k events among $\bigcap_j \overline{R_{ij}} \cap \mathscr{E}$. 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 $\Pr[G_i \mid \mathscr{E}']$. This heuristic is supported by the bound $\Pr\left[G_i \mid \mathscr{E} \cap \overline{R_{ij}}\right] \leq \Pr[G_i \mid \mathscr{E}]$ and would locally minimize the error done in the estimation. It provides a lower bound of the reliability for $k = 0$ and an exact value for $k = m$ (an upper bound can be obtained by considering fail-stop failures as transient ones). For other values of k, however, we have no guarantee. The resulting complexity drops down to $O(n^{k+1})$ with $k \in [0..m]$.

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 Lemma 1 for further details on this approach). To the best of our knowledge, we are not aware of any procedure for evaluating the reliability of strict schedules with fail-stop failures in polynomial time (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.

5.2 Simulation results

Simulations were conducted in order to assess this evaluation method. For each simulation, a random DAG is generated with 20 tasks, 30 edges (plus some edges to ensure that both the source and the sink are unique). Each cost is unitary and the platform consists of five homogeneous processors with Poissonian failure rates uniformly drawn from the interval $[0, 0.05]$. Each task is scheduled twice on randomly chosen processors. On total, 300 schedules are obtained and their exact reliabilities lie in the range $[0.2, 1]$. Note that due to the high cost of the approach, increasing the number of processors or the number of tasks makes it intractable to test each possible value for the parameter k. The chosen values enable each simulation to last less than one hour.

Fig. 6 depicts the running times of the method (black points) for each chosen value k. As expected, times grow exponentially with k. For each value of k, the absolute difference between the true reliability and the estimated one is represented with boxplots. In a boxplot, the bold line is the median, the box shows the quartiles, the bars show the whiskers (1.5 times the interquartile range from the box) and additional points are outliers. We see that increasing k leads to more precise results except for $k = 2$. For $k = 3$, the median is 0.86%.

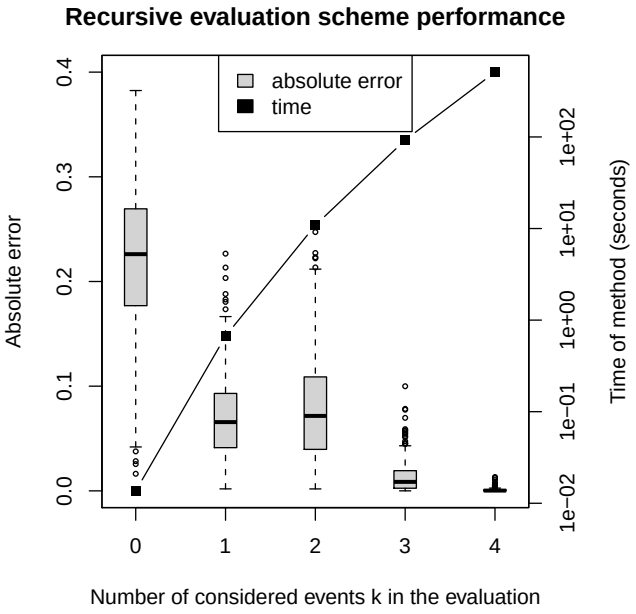


Fig. 6: Simulation results (20 tasks on five processors).

The method proposed in this section provides an estimation of the reliability of a strict schedule when failures are fail-stop. The precision of this estimation increases with the value of k. For high values of k, the method produces accurate results.

6 Conclusion

Fig. 2 summarizes known results on the complexity of reliability evaluation. The #P'-Completeness of evaluating the reliability of general schedules holds true both for transient and for fail-stop failures, and constitutes the major contribution of the paper. Moreover, this result holds for more general cases such as for preemptive schedules and schedules with communications. While the strict/fail-stop combination remains open, we have provided a method to estimate the reliability while limiting evaluation costs, from which bounds can be derived (with $k = 0$).

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 approx-

imations 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 time to some polynomial function of the application/platform parameters? Finally, we plan to study methods for effectively constructing reliable schedules based on a relevant evaluation mechanism. Whereas the first step would be to develop static scheduling algorithms, dynamic strategies could also provide interesting insights.

Acknowledgment: This work was supported in part by the ANR *StochaGrid* and *RESCUE* projects, and by the INRIA *ALEAE* project. We would like to thank the associate editor and the reviewers for their comments and suggestions, which greatly improved the final version of this paper.

References

Awerbuch B., Azar Y., Fiat A., Leighton F. T. (1996) Making commitments in the face of uncertainty: How to pick a winner almost every time. In: 28th ACM SToC, pp 519–530

Bannister J., Trivedi K. S. (1983) Task allocation in fault-tolerant distributed systems. Acta Informatica 20:261–281

Barlow R. E., Proschan F. (1967) Mathematical theory of reliability. John Wiley, New York

Bhatt S., Chung F., Leighton F., Rosenberg A. (1997) On optimal strategies for cycle-stealing in networks of workstations. IEEE Trans Computers 46(5):545–557

Bodlaender H. L., Wolle T. (2004) A note on the complexity of network reliability problems. IEEE Trans Inf Theory 47:1971–1988

Bream B. (1995) Reliability Block Diagrams and Reliability Modeling. Tech. rep., Office of Safety and Mission Assurance, NASA Lewis Research Center

Brucker P. (2004) Scheduling Algorithms. Springer-Verlag

Cappello F., Geist A., Gropp B., Kale L., Kramer B., Snir M. (2009) Toward Exascale Resilience. Int Journal of High Performance Computing Applications 23(4):374–388

Dongarra J., Jeannot E., Saule E., Shi Z. (2007) Bi-objective Scheduling Algorithms for Optimizing Makespan and Reliability on Heterogeneous Systems. In: 19th ACM Symp. on Parallelism in Algo. and Archi. (SPAA'07), San Diego, CA, USA

Girault A., Kalla H. (2009) A novel bicriteria scheduling heuristic providing a guaranteed global system failure rate. IEEE Trans Dependable Secure Computing 6(4):241–254

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

Jeannot E., Saule E., Trystram D. (2008) Bi-Objective Approximation Scheme for Makespan and Reliability Optimization on Uniform Parallel Machines. In: The 14th Int. Euro-Par Conf. on Parallel and Distributed Computing, Spain

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

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

Rosenberg A. L. (2002) Optimal schedules for cycle-stealing in a network of workstations with a bag-of-tasks workload. IEEE Trans Parallel Distrib Syst 13(2):179–191

Shatz S., Wang J. (1989) Models and algorithms for reliability-oriented task-allocation in redundant distributed-computer systems. IEEE Trans Reliability 38(1):16–26

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

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

Zhu D., Melhem R., Mossé D. (2004) The effects of energy management on reliability in real-time embedded systems. In: International Conference on Computer Aided Design, ICCAD'04, San Jose (CA), USA, pp 35–40