International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 6, Number 1, pages 2–26, January 2016

Checkpointing Strategies for Scheduling Computational Workflows

Guillaume Aupy
Laboratoire LIP, ENS Lyon, France

Anne Benoit
Laboratoire LIP, ENS Lyon, France

Henri Casanova
University of Hawai'i, USA

Yves Robert
Laboratoire LIP, ENS Lyon, France
& University Tennessee Knoxville, USA

Received: July 21, 2015
Revised: October 19, 2015
Accepted: December 1, 2015
Communicated by Susumu Matsumae

Abstract

We study the scheduling of computational workflows on compute resources that experience exponentially distributed failures. When a failure occurs, rollback and recovery is used to resume the execution from the last checkpointed state. The scheduling problem is to minimize the expected execution time by deciding in which order to execute the tasks in the workflow and deciding for each task whether to checkpoint it or not after it completes. We give a polynomial-time optimal algorithm for fork DAGs (Directed Acyclic Graphs) and show that the problem is NP-complete with join DAGs. We also investigate the complexity of the simple case in which no task is checkpointed. Our main result is a polynomial-time algorithm to compute the expected execution time of a workflow, with a given task execution order and specified to-be-checkpointed tasks. Using this algorithm as a basis, we propose several heuristics for solving the scheduling problem. We evaluate these heuristics for representative workflow configurations.

Keywords: checkpointing, scheduling, workflow, fault-tolerance, reliability

1 Introduction

Resilience has become a key concern when computing at large scales because enrolling more processors in an application execution leads to more frequent application failures [1]. (In this work we use the term "processor" in a broad sense to mean a processing elements of the platform on which one can run a portion of a parallel application, e.g., a multi-socket multi-core blade server.) Making each individual processor reliable, for instance via redundant hardware components, is costly. Since

costs are highly constrained when designing a parallel platform, one must typically use commercial-of-the-shelf (COTS) processors, the reliability of which is driven by the market. Consequently, each processor has a Mean Time Between Failure (MTBF), say μ, that varies from a few years to a century. When enrolling p processors to execute a tightly-coupled parallel application, a failure on any of the processors will cause an application failure. The overall MTBF of this set of processors is μ/p, which can be low (a few hours or less) when p is large. As a result, no matter how reliable the individual processors, there is a value of p above which failures become common rather than exceptional events.

The above considerations have prompted decades of research in the area of fault-tolerant computing. The most well-known approach is checkpoint-rollback-recovery, by which application state is saved to persistent storage at different points, e.g., periodically throughout execution [2, 3]. When a failure occurs, the application execution can be resumed from the most recently saved such state, or *checkpoint*. A well-studied question is that of the optimal checkpointing strategy [2, 3, 4]. Too infrequent checkpoints lead to wasteful re-computation when a failure occurs, but too frequent checkpoints lead to overhead during failure-free periods of the application execution. Checkpointing can happen in a coordinated or uncoordinated manner, and the advantages and drawbacks of both approaches are well-documented [5]. Checkpointing can be agnostic to the application, in which case full address space images are saved as checkpoints [6, 7]. Alternately, checkpointing can be application-aware so that only the application data truly needed to resume execution is saved. This latter approach is more efficient because less data needs to be saved, but requires modifying the implementation of the application [8].

In this work, we study the execution of workflow applications on platforms subject to processor failures. An application is structured as a Directed Acyclic Graph (DAG) in which each vertex represents a *tightly-coupled parallel* task and each edge represents a data dependency between tasks. This general model is relevant for many scientific workflows [9]. The difficulty of scheduling DAGs of parallel tasks, or applications with "mixed parallelism," without considering processor failures, has long been recognized [10]. It comes from the need to not only decide on a traversal of the task graph, as in classical scheduling problems, but also to decide how many processors should be assigned to each task. In addition, complex data redistributions must take place so that output data from one task can serve as input data to another task when the two tasks do not use the same number of processors. It is not clear how to model redistribution costs in practice and thus how to make judicious scheduling and processor allocation decisions [11, 12]. Because we consider processor failures, which makes the scheduling problem even more difficult, in this work we opt for a simplified scenario in which each task uses all the available processors. In other words, the workflow DAG is linearized and the tasks execute in sequence, using the whole fraction of the platform that is dedicated to the application. This scenario is representative of a large class of compute-intensive scientific applications whose workflow is partitioned into (typically large) tightly-coupled parallel computational kernels. Each parallel task is executed across all available processors, and produces output data that is kept in memory while executing its immediate successors in the DAG. Executing each task on all processors makes it possible to avoid complex data redistributions among tasks that use different numbers of processors [12]. While it would be possible to used checkpoint-roll-back recovery within each task, it would require either saving large checkpoints (application-agnostic) or modifying the implementation of the task (application-aware). Given that both approaches have drawbacks, we assume non-modified, and thus non-fault-tolerant, implementations for the tasks. Fault-tolerance must then be achieved by checkpointing the output data generated by each task once it completes. If there is a failure during a task execution, one must recover from the most recently saved checkpoints on all paths from the failed task upward to an entry task of the DAG, re-execute non-checkpointed predecessors of the task if necessary, and then re-execute the task itself. This is repeated until the task is successfully executed and its output possibly checkpointed.

We study the following problem. We are given a DAG of tasks and for each task we know how long it takes to compute its output, how long it takes to checkpoint its output, and how long it takes to recover its checkpointed output. We are given a platform with a known failure rate on which we want to execute the application. In which order should the tasks be executed and which tasks should be checkpointed? We call an answer to this twofold question a *schedule*. The objective is to

find a schedule that minimizes expected application execution time, or expected makespan. We call this problem DAG-ChkptSched.

To the best of our knowledge, DAG-ChkptSched has only been answered for the very specific case in which the DAG is a linear chain [13]. For general DAGs, the problem is more difficult. In fact, even computing the expected makespan of a given schedule is difficult. This is surprising, because the ordering of the tasks is given by the schedule as well as the location of all checkpoints. But when computing the expected execution time of a task, one has to account for the state of all its predecessors, which depends upon when the last failure has occurred. In this context, we make the following contributions:

- We show that although DAG-ChkptSched can be solved in polynomial time for fork DAGs[1], its associated decision problem is NP-complete for join DAGs[2]. This result shows the intrinsic complexity of DAG-ChkptSched, but is largely expected as both the linearization of the DAG and the location of the checkpoints must be determined.
- We study the simple instance of the problem where no task is checkpointed. We show that optimal schedules for tree-shaped DAGs[3] are depth-first schedules, but the complexity for general DAGs remains open.
- We provide a polynomial-time algorithm for computing the expected makespan of a schedule. This algorithm gives a fundamental basis for designing and comparing scheduling heuristics for arbitrary DAGs.
- We propose a set of heuristics for solving DAG-ChkptSched for general DAGs and evaluate these heuristics quantitatively. To the best of our knowledge, these heuristics are unique in the literature since previous work lacked an algorithm to estimate the makespan of a schedule (except when the DAG is a linear chain [13]).

The rest of this paper is organized as follows. Section 2 provides an overview of related work. Section 3 is devoted to formally defining the problem and all model parameters. Section 4 discusses the complexity of several instances of DAG-ChkptSched: fork DAGs (polynomial), join DAGs (NP-hard) and no-checkpoint (open, but polynomial for tree-shaped DAGs). Section 5 provides our key result that DAG-ChkptSched for general DAGs belongs in NP: we give a polynomial-time algorithm to compute the expected makespan of a given schedule. Section 6 presents a set of heuristics for solving the problem with general DAGs. These heuristics are evaluated experimentally in Section 7. Finally, Section 8 summarizes our main findings and discusses directions for future work.

2 Related work

Resilience to faults is one of the major issues for current and upcoming large-scale parallel platforms. The most common fault-tolerance technique used in high performance computing is checkpoint-rollback-recovery [6, 7, 5, 2]. A large body of work has studied periodic coordinated checkpointing for a single divisible application. Given the simplicity of the divisible model, a wide range of results are available including first order formulas for the checkpointing period that minimizes execution time [2, 3] or more accurate formulas for Weibull failure distributions [14, 15, 16]. The optimal checkpointing period is known only for exponential failure distributions [17]. Dynamic programming heuristics for arbitrary distributions have been proposed [13, 17]. Gelenbe and Derochette [4] give a first-order approximation of the optimal period to minimize average response time. They compare it to the period obtained by Young [2] in a model where they do not consider one single long application and a fully-loaded system, but instead multiple small independent applications that

[1] A fork DAG with $n+1$ tasks has an entry task T_{entry}, n exit tasks $T_1, \ldots, T_n$, and n edges from T_{entry} to each T_i.

[2] A join DAG with $n+1$ tasks has n entry tasks $T_1, \ldots, T_n$, an exit task T_{exit}, and n edges from each T_i to T_{exit}.

[3] A tree-shaped DAG has an entry task (the root), and all other tasks have in-degree 1 (single parent).

arrive in the system following a Poisson process. Finally, Gelenbe and Hernández [18] compute the optimal checkpointing period that minimizes computational waste in the case of age-dependent failures: they assume that the failure rate follows a Weibull distribution and that each checkpoint is a renewal point.

Few authors have studied the resilience problem with workflows when checkpointing can only take place at the end of each task. Bouguerra et al. [19] have studied a restricted version of DAG-ChkptSched when the workflow is a linear chain (with a single processor). They propose a greedy heuristic to minimize the total execution time in case of arbitrary failures. As already mentioned, Toueg and Babaoglu [13] have computed the optimal execution time for a linear chain of tasks using a dynamic programming algorithm to decide which tasks to checkpoint.

Our work is not restricted to linear chains and, as seen in upcoming sections, removing this restriction makes the problem fundamentally more difficult. In fact, even when a schedule is given (hence both a linearization of the DAG and a list of tasks to checkpoint), it is difficult to determine which tasks to re-execute and from which tasks to recover after one or more failures have occurred during application execution.

3 Framework

We consider a (subset of a) parallel platform with p processors, where each processor is a processing element that is subject to its own individual failures. When a failure occurs a processor experiences a *downtime* before it can be used again. In a production system, this downtime corresponds to replacing the processor by a logical spare. Like most works in the literature, we simply assume that a downtime lasts D seconds, where D is a constant. We assume that failures are i.i.d. (independent and identically distributed) across the processors and that the failure inter-arrival time at each processor is exponentially distributed with Mean Time Between Failures (MTBF) $\mu_{proc} = 1/\lambda_{proc}$.

On the set of processors we want to execute a task-parallel application that is structured as a DAG $G = (V, E)$, where V is a set of vertices and E a set of edges. Each vertex is a tightly coupled data-parallel task that is executed on all p available processors. Consequently, in all that follows, we can view the set of processors as a single macro-processor that experiences exponentially distributed failures with parameter $\lambda = p\lambda_{proc}$, i.e., with MTBF $\mu = \mu_{proc}/p$. Each edge corresponds to a data dependencies between two tasks. Since no two tasks run simultaneously, the sequence of executed tasks corresponds to one of the (many) linearizations of the DAG, i.e., task sequences that respect data dependencies. The DAG has n vertices, and the task corresponding to the i-th vertex is denoted by T_i. A failure-free execution of task T_i on the p processors takes w_i seconds (the task's computational *weight*). This execution produces an output that can be checkpointed in c_i seconds, and can be recovered from a checkpoint in r_i seconds. If task T_i executes successfully, then its successor tasks in the DAG can begin execution immediately since T_i's output data is available in memory (distributed over the p processors). If the output of a task is saved to a checkpoint, we say that the task is checkpointed.

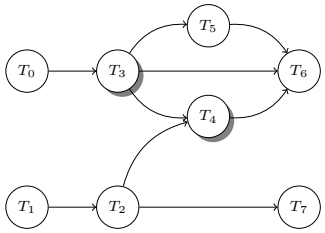


Figure 1: Example DAG. Tasks whose output is checkpointed (T_3 and T_4) are shadowed.

Informally, when a failure strikes during the execution of a task, all data that is stored in the processor memories is lost. Thus the output data of all tasks that have not been checkpointed are

lost. Of course to recompute the failed task, we only need to recompute those of its ancestors that have not been checkpointed. But we will also need to recompute all non-checkpointed tasks so as to proceed with the execution of their successors. It is thus necessary to remember when the last failure occured, explaining why the algorithm in Section 5, which computes the expected makespan, is so intricate.

More precisely, if a failure strikes during the execution of T_i, then T_i must be re-executed. This re-execution requires that the input data to T_i be available in memory. For each reverse path in the DAG from T_i back to an entry task, one must find the most recently executed checkpointed task. One must then recover from that checkpoint, and re-execute all the tasks that were executed after that checkpointed task, i.e., all tasks whose output was lost and that are ancestors of T_i along the reverse path. It may be that on such a path from T_i to an entry task no checkpointed task is found, in which case one must begin by re-executing the entry task. An example DAG is shown in Figure 1, for which tasks whose output is checkpointed are shadowed (T_3 and T_4). Consider the following linearization of the DAG: $T_0T_3T_1T_2T_4T_5T_6T_7$. Let us assume that the first and only failure occurs during the execution of T_5. To re-execute T_5, one needs to recover the checkpointed output of T_3. To execute T_6, one then needs to recover the checkpointed output of T_4 and use the output of T_5 that is now available in memory. This sequence of recoveries and re-executions must be re-attempted until T_6 executes successfully. Finally, the output of T_2 was lost due to the failure, and no task is checkpointed on the reverse path from T_7 to T_1. One must therefore re-execute T_1, T_2, and then finally T_7. This example is for a single failure occurrence and yet is not straightforward, hinting at the complexity of the problem in the general case.

As seen in the example, the DAG can have multiple entry tasks. The entry tasks (sometimes also referred to as sources), when restarted, do not have to recover any output from predecessors. In practice, each entry task would read the application's input data from disk, the overhead of which is included in the task's weight. The DAG can also have multiple exit tasks (sometimes also referred to as sinks). As soon as an exit task completes, it is removed from the DAG as well as any of its ancestors that have no remaining exit tasks as descendants. In practice, each exit task would write the application's output data to disk, and here again this overhead is included in the exit task's weight.

Executing the DAG in a fault-tolerant manner boils down to re-executing all the work that has been lost due to a failure, restarting from the most recent checkpoints if found and re-executing entry tasks otherwise. We enforce that the most recent checkpoint be used when recovering from a failure. It would be conceivable to ignore the checkpoints and, for instance, always re-execute the path completely from each entry task. This is only useful when the w_i values are small and the r_i values are large. Such situations are of dubious practical interest. It makes little sense to checkpoint a task if the time to recover the checkpoint is known to be longer than the time to re-execute that task. If this were the case, then the task could be fused with some of its predecessors for instance. So, in this work, when recovering from a failure, we enforce the use of the most recent checkpoints whenever possible.

Formally, let $\mathbb{E}[t(w;c;r)]$ denote the expected time to execute a task that would take w seconds in a fault-free execution and c seconds to checkpoint the output of this computation, with a recovery time of r seconds if a failure occurs during computation or checkpointing. If failures are exponentially distributed with mean $1/\lambda$, and the processor downtime is D, it is shown in [17, 20] that:

$$\mathbb{E}[t(w;c;r)] = e^{\lambda r}\left(\frac{1}{\lambda}+D\right)\left(e^{\lambda(w+c)}-1\right). \tag{1}$$

We make extensive use of this notation and this result in this work. It is crucial to note that the above formula is valid even if failures occur during checkpointing or recovery. Many works in the literature assume that checkpointing and recovery are failure-free, an assumption that is not realistic for large numbers of processors.

We define a *schedule* as a linearization of the DAG in which, for each task, it is specified whether the task's output should be checkpointed. The objective is to find the schedule that has the minimum expected makespan. Note that if $\lambda = 0$, i.e., if there are no failures, then one should do no checkpointing and all the linearizations of the DAG are equivalent. However, in the presence of

failures, there is the usual trade-off between spending too much time checkpointing or spending too much time recovering and re-executing.

4 Complexity

In this section, we present several theoretical results to assess the complexity of DAG-ChkptSched. First, in Section 4.1, we introduce a polynomial-time algorithm for fork DAGs. Then we establish the NP-completeness of the decision problem associated to DAG-ChkptSched for join DAGs in Section 4.2 and we exhibit particular cases that can be solved in polynomial time. We conclude this section by discussing the problem instance where no task is checkpointed. While the complexity of this (apparently simple) instance remains open for general DAGs, we show that for tree-shaped DAGs any depth-first schedule is optimal (Section 4.3).

4.1 Fork DAGs

Theorem 1. DAG-ChkptSched *can be solved in linear time for a fork DAG.*

Proof. We consider a fork DAG with an entry task T_{entry} and n exit tasks $T_1, \ldots, T_n$. If T_{entry} is checkpointed, then when T_i fails we recover the checkpoint and try again. If T_{entry} is not checkpointed, then we re-execute T_{entry} but *without* re-executing the T_i tasks that have already completed. The question is to decide whether or not T_{entry} should be checkpointed, and to decide for the ordering of the n exit tasks.

We renumber the tasks so that task T_i is the i^{th} task executed in the linearization. Let X_i be the random variable that corresponds to the execution time between the end of the first successful execution of task T_{i-1} and the end of the first successful execution of task T_i. Let X_0 be the random variable that corresponds to the execution time of T_{entry} followed by a checkpoint. *Note that the case where T_{entry} is not checkpointed is equivalent to considering $c_{entry} = 0$, $r_{entry} = w_{entry}$.* The expected execution time of the DAG is $\mathbb{E}[\sum_{i=0}^{n} X_i]$.

By definition, $\mathbb{E}[X_0] = \mathbb{E}[t\,(w_{\text{entry}}; c_{\text{entry}}; 0)]$. Furthermore, it is straightforward to see that at the beginning of interval X_i, the output of T_{entry} is still available in memory (meaning that there has been no fault between the end of the last recovery –or execution– of T_{entry} and the beginning of T_i). As a result $\mathbb{E}[X_i] = \mathbb{E}[t\,(w_i; 0; r_{\text{entry}})]$.

The execution time of the schedule does not depend on the linearization of the tasks. This is because, with the assumption that failures are exponentially distributed, the set of tasks that follow the checkpoint can be executed in any order.

In conclusion, if $\mathbb{E}[t\,(w_{\text{entry}}; 0; 0)] + \sum_{i=1}^{n} \mathbb{E}[t\,(w_i; 0; w_{\text{entry}})] > \mathbb{E}[t\,(w_{\text{entry}}; c_{\text{entry}}; 0)] + \sum_{i=1}^{n} \mathbb{E}[t\,(w_i; 0; r_{\text{entry}})]$ then T_{entry} should be checkpointed, otherwise it should not. □

4.2 Join DAGs

Consider a join DAG with a single exit task T_{exit} and n entry tasks $T_1, \ldots, T_n$. We denote by I_{CKPT}, resp. I_{NCKPT}, the subset of $\{T_1, \ldots, T_n\}$ composed of the tasks that are checkpointed, resp. not checkpointed.

We first introduce Lemmas 1 and 2 which help characterize the structure of the optimal solution:

Lemma 1. *In an optimal schedule, the tasks in I_{CKPT} are executed before the tasks in I_{NCKPT}. When a failure occurs, the recoveries from the previously executed tasks in I_{CKPT} are executed after the last task from I_{CKPT}.*

Lemma 2. *Given the two sets I_{CKPT} and I_{NCKPT}, we can compute the optimal expected makespan, which is achieved by scheduling the tasks in I_{CKPT} in non-increasing values of $g(i)$, where*

$$g(i) = e^{-\lambda(w_i + c_i + r_i)} + e^{-\lambda r_i} - e^{-\lambda(w_i + c_i)}.$$

Proof. The order in which the tasks from I_{NCKPT} and recoveries are executed does not matter. This is because all these must be executed consecutively without failures, followed by T_{exit}. The probability of a correct execution simply depends on the sum of the corresponding w_i and c_i, not their order.

Let us now consider the expected execution time **for a given schedule order σ of the tasks from** I_{CKPT} (meaning that in the schedule, $T_{\sigma(1)}$ is scheduled before $T_{\sigma(2)}$, ..., scheduled before $T_{\sigma(|I_{\text{CKPT}}|)}$).

The time to execute a task $T_i \in I_{\text{CKPT}}$ and its checkpoint is independent of the rest of the computation and is equal to

$$\mathbb{E}[t(w_i, 0; c_i)] = \left(\frac{1}{\lambda} + D\right)\left(e^{\lambda(w_i+c_i)} - 1\right).$$

The expected time to execute the NCKPT tasks, the recoveries, and T_{exit} depends on when the last failure occurred. This is because the number of recoveries to perform will differ depending on when that failure occurred. Let us call $W_{\text{NCKPT}} = \sum_{i\in I_{\text{NCKPT}}} w_i + w_{\text{exit}}$; this is a constant amount of work that needs to be done regardless of when the last fault occurred:

- If the last fault occurred during the computation of the NCKPT tasks, recoveries or exit task, then all recoveries should be done and the expected time to execute the NCKPT tasks, the recoveries, and T_{exit} is:

$$t_0 = \left(\frac{1}{\lambda} + D\right)\left(e^{\lambda\left(W_{\text{NCKPT}} + \sum_{i\in I_{\text{CKPT}}} r_i\right)} - 1\right).$$

- If the last fault occurred during the computation of the k^{th} checkpointed task (event E_k), then we first execute only the $k-1$ first recoveries. With probability $p_k^{(\sigma)} = e^{-\lambda\left(W_{\text{NCKPT}} + \sum_{i=1}^{k-1} r_{\sigma(i)}\right)}$ there is no subsequent failure, otherwise there is a failure and all recoveries should be re-executed. The expected time is thus:

$$\begin{aligned} t_k^{(\sigma)} &= p_k^{(\sigma)}\left(W_{\text{NCKPT}} + \sum_{i=1}^{k-1} r_{\sigma(i)}\right) + \left(1 - p_k^{(\sigma)}\right) \times \\ &\quad \left(\mathbb{E}[t_{lost}(W_{\text{NCKPT}} + \sum_{i=1}^{k-1} r_{\sigma(i)})] + D + t_0\right) \\ &= \left(1 - p_k^{(\sigma)}\right)\left(\frac{1}{\lambda} + D + t_0\right), \end{aligned}$$

because $\mathbb{E}[t_{lost}(w)] = 1/\lambda - w/(e^{\lambda w} - 1)$.

Finally, we have seen in Lemma 1 that a schedule proceeds in two distinct phases: first we execute the tasks in I_{CKPT} (with known execution time), then we execute the tasks in I_{NCKPT}, necessary recoveries and exit task. At the end of the first phase depending on when the last fault occurred, we are in either one of the events $E_1, \ldots, E_{|I_{\text{CKPT}}|}$. Precisely, with probability $q_i^{(\sigma)}$ we are in the event E_i, where

$$\begin{cases} q_1^{(\sigma)} = e^{-\lambda \sum_{j=2}^{|I_{\text{CKPT}}|}(w_{\sigma(j)}+c_{\sigma(j)})}, \\ q_{i\neq 1}^{(\sigma)} = \left(1 - e^{-\lambda(w_{\sigma(i)}+c_{\sigma(i)})}\right) e^{-\lambda \sum_{j=i+1}^{|I_{\text{CKPT}}|}(w_{\sigma(j)}+c_{\sigma(j)})}. \end{cases}$$

Finally, the expected execution of the second phase is $\sum_{i=1}^{|I_{\text{CKPT}}|} q_i^{(\sigma)} t_i^{(\sigma)}$, and the total expected execution time is:

$$\begin{aligned} t_\sigma &= \sum_{i\in I_{\text{CKPT}}}\left(\frac{1}{\lambda} + D\right)\left(e^{\lambda(w_i+c_i)}\right) + \sum_{i=1}^{|I_{\text{CKPT}}|} q_i^{(\sigma)}\left(1 - p_i^{(\sigma)}\right)\left(\frac{1}{\lambda} + D + t_0\right) \\ t_\sigma &= \left(\frac{1}{\lambda} + D\right)\left(\sum_{i\in I_{\text{CKPT}}}\left(e^{\lambda(w_i+c_i)}\right) + e^{\lambda\left(W_{\text{NCKPT}} + \sum_{i\in I_{\text{CKPT}}} r_i\right)} \sum_{i=1}^{|I_{\text{CKPT}}|} q_i^{(\sigma)}\left(1 - p_i^{(\sigma)}\right)\right). \end{aligned} \tag{2}$$

Now that we have computed the expected execution time for a given order, let us **focus on finding the optimal order**: for a given schedule σ of the tasks in I_{CKPT}, let us compare its execution time to the same schedule where $T_{\sigma(i)}$ and $T_{\sigma(i+1)}$ are permuted (ϕ such that $\phi(i) = \sigma(i+1)$, $\phi(i+1) = \sigma(i)$ and $\phi(j) = \sigma(j)$ for all other j). One can notice that for $j \neq i, i+1$, then $q_j^{(\sigma)} = q_j^{(\phi)}$ and $t_j^{(\sigma)} = t_j^{(\phi)}$. Therefore:

$$\begin{aligned}\frac{t_\sigma - t_\phi}{\frac{1}{\lambda} + D + t_0} &= q_i^{(\sigma)}\left(1 - p_i^{(\sigma)}\right) - q_i^{(\phi)}\left(1 - p_i^{(\phi)}\right) \\ &\quad + q_{i+1}^{(\sigma)}\left(1 - p_{i+1}^{(\sigma)}\right) - q_{i+1}^{(\phi)}\left(1 - p_{i+1}^{(\phi)}\right) \ .\end{aligned}$$

Let us first consider the case where $i \neq 1$. For convenience we define $R = \sum_{j=1}^{i-1} r_{\sigma(j)}$ and $W = \sum_{j=i+2}^{|I_{\text{CKPT}}|}(w_{\sigma(j)} + c_{\sigma(j)})$ ($W = 0$ when $i+1 = |I_{\text{CKPT}}|$). Then:

$$\begin{aligned}\frac{t_\sigma - t_\phi}{\frac{1}{\lambda} + D + t_0} &= \left(1 - e^{-\lambda(w_{\sigma(i)} + c_{\sigma(i)})}\right) e^{-\lambda\left(W + w_{\sigma(i+1)} + c_{\sigma(i+1)}\right)} \left(1 - e^{-\lambda(W_{\text{NCKPT}} + R)}\right) \\ &\quad - \left(1 - e^{-\lambda(w_{\sigma(i+1)} + c_{\sigma(i+1)})}\right) e^{-\lambda\left(W + w_{\sigma(i)} + c_{\sigma(i)}\right)} \left(1 - e^{-\lambda(W_{\text{NCKPT}} + R)}\right) \\ &\quad + \left(1 - e^{-\lambda(w_{\sigma(i+1)} + c_{\sigma(i+1)})}\right) e^{-\lambda W} \left(1 - e^{-\lambda\left(W_{\text{NCKPT}} + R + r_{\sigma(i)}\right)}\right) \\ &\quad - \left(1 - e^{-\lambda(w_{\sigma(i)} + c_{\sigma(i)})}\right) e^{-\lambda W} \left(1 - e^{-\lambda\left(W_{\text{NCKPT}} + R + r_{\sigma(i+1)}\right)}\right) \ , \\ \frac{e^{\lambda W}\left(t_\sigma - t_\phi\right)}{\frac{1}{\lambda} + D + t_0} &= \left(e^{-\lambda(w_{\sigma(i+1)} + c_{\sigma(i+1)})} - e^{-\lambda(w_{\sigma(i)} + c_{\sigma(i)})}\right) \left(1 - e^{-\lambda(W_{\text{NCKPT}} + R)}\right) \\ &\quad + \left(e^{-\lambda r_{\sigma(i+1)}} - e^{-\lambda r_{\sigma(i)}}\right) e^{-\lambda(W_{\text{NCKPT}} + R)} \\ &\quad - \left(e^{-\lambda(w_{\sigma(i+1)} + c_{\sigma(i+1)})} - e^{-\lambda(w_{\sigma(i)} + c_{\sigma(i)})}\right) \\ &\quad + \left(e^{-\lambda\left(r_{\sigma(i+1)} + w_{\sigma(i+1)} + c_{\sigma(i+1)}\right)} - e^{-\lambda\left(r_{\sigma(i)} + w_{\sigma(i)} + c_{\sigma(i)}\right)}\right) e^{-\lambda(W_{\text{NCKPT}} + R)} \\ &= e^{-\lambda(W_{\text{NCKPT}} + R)}\left(g(\sigma(i+1)) - g(\sigma(i))\right) \ ,\end{aligned}$$

where $g : i \mapsto e^{-\lambda(w_i + c_i + r_i)} + e^{-\lambda r_i} - e^{-\lambda(w_i + c_i)}$. In the case when $i = 1$, similarly we obtain:

$$\begin{aligned}\frac{t_\sigma - t_\phi}{\frac{1}{\lambda} + D + t_0} &= e^{-\lambda\left(W + w_{\sigma(i+1)} + c_{\sigma(i+1)}\right)} \left(1 - e^{-\lambda(W_{\text{NCKPT}} + R)}\right) \\ &\quad - e^{-\lambda\left(W + w_{\sigma(i)} + c_{\sigma(i)}\right)} \left(1 - e^{-\lambda(W_{\text{NCKPT}} + R)}\right) \\ &\quad + \left(1 - e^{-\lambda(w_{\sigma(i+1)} + c_{\sigma(i+1)})}\right) e^{-\lambda W} \left(1 - e^{-\lambda\left(W_{\text{NCKPT}} + R + r_{\sigma(i)}\right)}\right) \\ &\quad - \left(1 - e^{-\lambda(w_{\sigma(i)} + c_{\sigma(i)})}\right) e^{-\lambda W} \left(1 - e^{-\lambda\left(W_{\text{NCKPT}} + R + r_{\sigma(i+1)}\right)}\right) \ , \\ \frac{e^{\lambda W}\left(t_\sigma - t_\phi\right)}{\frac{1}{\lambda} + D + t_0} &= e^{-\lambda(W_{\text{NCKPT}} + R)}\left(g(\sigma(i+1)) - g(\sigma(i))\right) \ .\end{aligned}$$

We conclude that in the optimal schedule, the set of tasks in I_{CKPT} should be sorted by non-increasing g values. □

The first consequence of Lemmas 1 and 2 is that given the two sets I_{CKPT} and I_{NCKPT}, we can construct the optimal solution in polynomial time. Furthermore, from Lemma 2 and Equation (2) we have:

Corollary 1. *When $r_i = 0$ for all i, task ordering does not matter. The optimal expected execution time is then:*

$$\left(\frac{1}{\lambda} + D\right)\left(\sum_{i \in I_{\text{CKPT}}}\left(e^{\lambda(w_i + c_i)} - 1\right) + \left(e^{\lambda\left(\sum_{i \in I_{\text{NCKPT}}} w_i + w_{exit}\right)} - 1\right)\right) . \tag{3}$$

Theorem 2. DAG-ChkptSched *for join DAGs is NP-complete.*

Proof. Consider the associated decision problem: given a join DAG, λ, and a bound on the expected execution time, can we find the sets I_{Ckpt} and I_{NCkpt}, and the order in which the tasks are executed such that the bound on the expected execution time is respected? The problem is clearly in NP: we have shown in Lemma 2 that given the sets I_{Ckpt} and I_{NCkpt} we can compute the expected execution time in polynomial time (via an analytical formula).

To establish the NP-completeness, we use a reduction from SUBSET-SUM [21]. Let $\mathcal{I}_1$ be an instance of SUBSET-SUM: given n strictly positive integers $w_1, \ldots, w_n$, and a positive integer X, does there exist a subset I of $\{1, \ldots, n\}$ such that $\sum_{i \in I} w_i = X$? Let $S = \sum_{i=1}^{n} w_i$.

We build the following instance $\mathcal{I}_2$ of our problem. We have a join DAG with n entry tasks $T_1, \ldots, T_n$ and an exit task T_{exit} with $w_{\text{exit}} = 0$ and, for all tasks $T_i, i = 1..n$,

$$\begin{cases} w_i = & w_i \ , \\ c_i = & (X - w_i) + \frac{1}{\lambda} \log \left(\lambda w_i + e^{-\lambda X} \right) \ , \\ r_i = & 0 \ . \end{cases}$$

We assume that $\lambda \geq \frac{1}{\min_i w_i}$, so that for all i, $c_i > 0$. Finally, we assume that the bound on the expected execution time is: $t_{\min} = \lambda e^{\lambda X} (S - X) + e^{\lambda X} - 1$.

Let us show that $\mathcal{I}_2$ has a solution $(I_{\text{Ckpt}}, I_{\text{NCkpt}})$ if and only if we can find a set of tasks I_{NCkpt} such that $\sum_{i \in I_{\text{NCkpt}}} w_i = X$. We will thus have shown that $\mathcal{I}_2$ has a solution if and only if $\mathcal{I}_1$ has one, the set I_{NCkpt} from $\mathcal{I}_2$ being the set I from $\mathcal{I}_1$ if such a set exists.

Let us call $W = \sum_{i \in I_{\text{NCkpt}}} w_i$. We have seen in Equation (3) that the expected execution time is:

$$\begin{aligned} \mathbb{E}[T] &= \sum_{i \in I_{\text{Ckpt}}} \left(e^{\lambda (w_i + c_i)} - 1 \right) + \left(e^{\lambda (\sum_{i \in I_{\text{NCkpt}}} w_i + w_{\text{exit}})} - 1 \right) \\ &= \sum_{i \in I_{\text{Ckpt}}} \lambda e^{\lambda X} w_i + \left(e^{\lambda W} - 1 \right) \\ &= \lambda e^{\lambda X} (S - W) + e^{\lambda W} - 1 \ . \end{aligned}$$

We can differentiate $\mathbb{E}[T]$ with respect to W: $\mathbb{E}[t(W)]' = -\lambda e^{\lambda X} + \lambda e^{\lambda W}$. This function is increasing, and equal to 0 when $W = X$. Therefore $\mathbb{E}[T]$ is minimum when $W = X$, and its value is exactly $t_{\min}$. We conclude that $(I_{\text{Ckpt}}, I_{\text{NCkpt}})$ is a solution to $\mathcal{I}_2$ if and only if $\sum_{i \in I_{\text{NCkpt}}} w_i = X$, which concludes the proof. □

4.3 Without checkpoints

In this section, we investigate the simple instance of the problem where no task is checkpointed. Given a DAG, the problem then reduces to finding the optimal linearization. We start with a few examples to show that this problem is not as simple as it appears at first sight.

Consider the trivial DAG in Figure 2. There are three tasks T_1, T_2, T_3 and a unique dependency edge from T_1 to T_3. Once T_3 has been successfully completed, we can remove tasks T_1 and T_3 from the DAG, hence the linearization (T_1, T_3, T_2) is optimal while (T_1, T_2, T_3) is not. Of course the linearization (T_2, T_1, T_3) is optimal too. This small example gives us some intuition to the design of efficient schedules:

1. Choose an output task T (a task without any successor in the DAG), execute all predecessors of T in sequence, and then T itself.
2. Repeat the above step until T is successfully completed.
3. Prune the DAG by removing T, and possibly other tasks that have become output tasks.
4. Repeat.

Consider now the bipartite DAG in Figure 3. There are 3 input tasks, and 3 output tasks. It is not clear which output task to select and execute first, because each output task has a predecessor set (of size two) which is intersecting that of another output task. The problem of finding the best

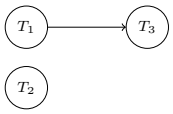


Figure 2: First example with a single dependency edge.

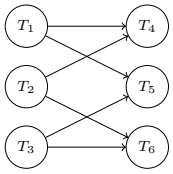


Figure 3: Second example with a bipartite DAG.

linearization seems of a combinatorial nature. We have not been able to determine its complexity but we conjecture that this problem is NP-complete, even for bipartite DAGs.

Although the complexity of the problem remains open, we expect depth-first linearizations to be more efficient than other traversals such as breadth-first ones. To further support this intuitive claim, we show that for tree-shaped DAGs[4] any depth-first traversal of the DAG leads to an optimal linearization. Note that we assess the efficiency of depth-first traversals for general DAGs empirically in Section 7.

Theorem 3. *When no task is checkpointed, a linearization for* DAG-ChkptSched *on a tree-shaped DAG is optimal if and only if it is a depth-first traversal.*

Algorithm 1 Depth-First Traversal on a tree

```
1: procedure Pre-order(T)
2:     Execute the root element (or current element)
3:     for x child of T do
4:         Let Tree(x) be the subtree of T rooted in x
5:         Traverse Tree(x) by recursively calling Pre-order.
6:     end for
7: end procedure
```

Proof. We begin by giving a characterization of a depth-first traversal (DFT) on a tree. The linearization of a tree by a schedule σ is a DFT if any subtree that started being executed is finished before the algorithm starts the execution of a node not contained in this subtree. Obviously, because of data dependencies, any DFT is a pre-order traversal.

Let $\mathcal{G} = (V, E)$ be a tree of n tasks, rooted in T_1, and let σ be a traversal of $\mathcal{G}$ (without any checkpoint). Let E_σ be the expected execution time of the traversal σ. If T_i is executed in the $j = \sigma^{-1}(i)^{\text{th}}$ position, then we define X_i as the random variable that corresponds to the execution time between the end of the first successful execution of task $T_{\sigma(j-1)}$ (the predecessor of T_i in the schedule σ) and the end of the first successful execution of T_i. Intuitively, X_i is the time necessary to execute T_i in the schedule.

We have $E_\sigma = \mathbb{E}[\sum_{i=1}^{n} X_{\sigma(i)}]$. Indeed, the time elapsed between the beginning of the execution and the end of the first execution of the first task is $X_{\sigma(1)}$, then the time elapsed between the end

[4]We deal with out-trees here. For in-trees, the problem is trivial since all linearizations are optimal: either all tasks (including the root task) are successfully executed, or the schedule must restart from scratch.

of the first execution of the $(i-1)^{\text{th}}$ task and the time between the end of the first execution of the i^{th} task is $X_{\sigma(i)}$. Overall the $X_{\sigma(i)}$s do not overlap and cover the whole execution. Hence we have: $E_\sigma = \mathbb{E}[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} \mathbb{E}[X_i]$ by linearity of the expectation.

For any task T_i, we give a lower bound to $\mathbb{E}[X_i]$: it is minimized when all necessary data is stored in memory, and when the amount of work to execute is only w_i if there are no failure, and $\sum_{j\in\text{ancestors}(i)} w_j$ if a failure occurred. Hence,

$$\mathbb{E}[X_i] \geq \mathbb{E}[t\left(w_i; 0; \sum_{j\in\text{ancestors}(i)} w_j\right)] .$$

Finally, because we are considering an execution without any checkpoint, then at the end of a successful execution of a given task T_i, necessarily all its ancestors are alive (with output stored in memory) and do not have to be recomputed were we execute one of their successors. Let us now prove the result.

Assume the linearization strategy σ is a DFT. In this case, we show that for all i

$$\mathbb{E}[X_i] = \mathbb{E}[t\left(w_i; 0; \sum_{j\in\text{ancestors}(i)} w_j\right)].$$

Note first that in any linearization strategy, $\sigma(1) = 1$: the root of the tree is executed first because of precedence constraints, and $\mathbb{E}[X_1] = \mathbb{E}[t\,(w_1; 0; 0)]$. Consider T_{i_0} a task with $i_0 \neq 1$. Let j_0 be its rank in the execution: $\sigma(j_0) = i_0$. Then consider i_1, the direct predecessor of i_0 in the tree DAG, and let j_1 be its rank in the execution: $\sigma(j_1) = i_1$. By definition of the precedence graph, necessarily, $j_1 < j_0$: the first execution of T_{i_1} is necessarily anterior to the first execution of T_{i_0}.

By definition of a DFT, any task executed between j_1 and j_0 belongs to the subtree rooted in T_{i_1}, Tree(T_{i_1}). Then in particular, $T_{\sigma(i_0-1)}$ is a successor of T_{i_1}, and at the end of its execution, all the tasks on the path from T_1 to T_{i_1} are alive (with output stored in memory). Then, $\mathbb{E}[X_{i_0}] = \mathbb{E}[t\left(w_{i_0}; 0; \sum_{j\in\text{ancestors}(i_0)} w_j\right)]$.

As a result, the execution time of the schedule is:

$$E_\sigma = \sum_{i=1}^{n} \mathbb{E}[t\left(w_i; 0; \sum_{j\in\text{ancestors}(i)} w_j\right)] ,$$

which matches the lower bound for each task, hence showing the optimality of the schedule.

Assume the linearization strategy is not a DFT. In this case, according to the characterization of a DFT, there exists a subtree Tree(T_{i_0}), not reduced to a single node T_{i_0}, such that:

- the root of the subtree was executed at position $j_0 = \sigma^{-1}(i_0)$,
- there exist $j^{out} < j^{in}$ such that $T_{\sigma(j^{out})} \notin \text{Tree}(T_{i_0})$ and $T_{\sigma(j^{in})} \in \text{Tree}(T_{i_0})$.

Intuitively, the task $T_{\sigma(j^{out})}$ is executed in the middle of the tree rooted in T_{i_0}. Let T_{i_1} the first successor of T_{i_0} executed in the schedule after $T_{\sigma(j^{out})}$, and let $j_1 = \sigma^{-1}(i_1)$. Clearly, $j^{out} < j_1 \leq j^{in}$ and $T_{\sigma(j_1-1)} \notin \text{Tree}(i_0)$.

Hence with probability at least $p_1 = 1-e^{-\lambda w_{\sigma(j_1-1)}}$, there is a failure during $X_{\sigma(j_1-1)}$ (the amount of work to do during $X_{\sigma(j_1-1)}$ is at least $w_{\sigma(j_1-1)}$), in which case at the beginning of X_{i_1}, there is at least $w_{i_0} + w_{i_1}$ units of work to do (T_{i_0} would not have been recomputed during $X_{\sigma(j_1-1)}$). Note that this is independent of anything that happened in the past and only depends on the schedule, potentially p_1 is greater than this value, and the amount of work that needs to be recomputed is

also greater than this value. This shows that

$$\mathbb{E}[X_{i_1}] \geq (1-p_1)\mathbb{E}[t\left(w_{i_1}; 0; \sum_{j\in\text{ancestors}(i_1)} w_j\right)] + p_1\mathbb{E}[t\left(w_{i_0}+w_{i_1}; 0; \sum_{j\in\text{ancestors}(i_1)} w_j\right)]$$
$$> \mathbb{E}[t\left(w_{i_1}; 0; \sum_{j\in\text{ancestors}(i_1)} w_j\right)] .$$

Finally, we obtain that:

$$E_\sigma > \sum_{i=1}^{n} \mathbb{E}[t\left(w_i; 0; \sum_{j\in\text{ancestors}(i)} w_j\right)] ,$$

and the schedule is not optimal (any DFT schedule has a lower expected execution time). □

5 Evaluating a schedule for a general DAG

In this section, we consider a general DAG and a given schedule that specifies both a linearization of the DAG and which tasks are checkpointed. Our main contribution is to provide a polynomial-time algorithm to compute the expected execution time of the DAG with this schedule. For simplicity, we renumber the tasks so that task T_i is the i^{th} task executed in the linearization of the DAG.

Theorem 4. *Given a DAG, and a schedule for this DAG, it is possible to compute the expected execution time in polynomial time.*

Proof. Let X_i be the random variable that corresponds to the execution time between the end of the first successful execution of task T_{i-1} and the end of the first successful execution of task T_i. The expected execution time of the DAG is $\mathbb{E}[\sum_{i=1}^{n} X_i]$. Let $F(X_i)$ be the event "There was a fault during X_i." Let Z_k^i be the event "There was a fault during X_k and no fault during X_{k+1} to X_{i-1}, given that T_{i-1} was successfully executed." We have:

$$Z_k^i = \bigcap_{j=k+1}^{i-1} \overline{F(X_j)} \bigcap F(X_k) \tag{4}$$

(for the limit cases, $Z_{i-1}^i = F(X_{i-1})$ and $Z_0^i = \bigcap_{j=1}^{i-1} \overline{F(X_j)}$). The set of events Z_k^i for $0 \leq k \leq i-1$ partitions the set of possibilities for X_i. Hence we can write

$$\mathbb{E}[X_i] = \sum_{k=0}^{i-1} \mathbb{P}(Z_k^i)\mathbb{E}[X_i|Z_k^i]. \tag{5}$$

We now need to show how to compute the $\mathbb{P}(Z_k^i)$ and $\mathbb{E}[X_i|Z_k^i]$.

Definition 1 ($T_i^{\downarrow k}$)**.** Given a schedule, let $j < k \leq i$, then we say that $T_j \in T_i^{\downarrow k}$, if for all $k \leq l < i$, $T_j \notin T_l^{\downarrow k}$, and

(i) either T_j is a direct predecessor of T_i,

(ii) or there exists $T_l \in T_i^{\downarrow k}$ such that T_l not checkpointed and T_j is a direct predecessor of T_l.

Less formally, the set $T_i^{\downarrow k}$ corresponds to all the predecessors of T_i (in the DAG), whose output is lost if the event Z_k^i occurs and needed for the computation of T_i. For instance, it is not lost if it has been recomputed for another task executed after the last fault (that occurred during the computation of T_k) but still before T_i. Furthermore, it is not needed if for all paths between T_j

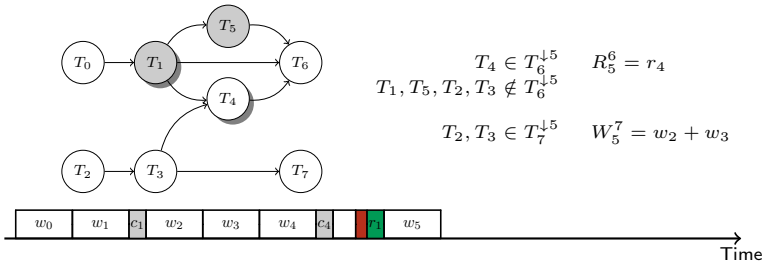


Figure 4: Consider the DAG in Figure 1. For clarity tasks are renumbered according to their execution order in a given linearization for the DAG. The output of T_1 and T_4 are checkpointed. Assume that a single fault occurs, and that it occurs during the execution of T_5. Then when the execution of T_6 begins, the output of T_1 and T_5 are already in memory, due to the re-execution of T_5, and do not need to be recomputed. However the output of T_4 needs to be recovered. Hence $T_4 \in T_6^{\downarrow 5}$. However, because it was checkpointed, we do not need to recover T_2 and T_3. Then, when computing T_7, the output of T_3 is needed, hence $T_3 \in T_7^{\downarrow 5}$. To compute T_3, we further need T_2. Hence $T_2 \in T_7^{\downarrow 5}$.

and T_i, there is a task whose output is not lost. If $T_j \in T_i^{\downarrow k}$ was not checkpointed, then we need to execute its work w_j again, otherwise we need to execute the recovery r_j. We give an illustrative example in Figure 4

Computing all sets $T_i^{\downarrow k}$ is the key to evaluating the schedule makespan.

Let W_k^i be the sum of the w_j such that (i) T_j is a non-checkpointed task and (ii) $T_j \in T_i^{\downarrow k}$. Similarly, let R_k^i be the sum of the r_j such that (i) T_j is a checkpointed task and (ii) $T_j \in T_i^{\downarrow k}$. We now show the following three properties:

A. $\forall k, 0 \leq k < i-1$,
$$\mathbb{P}(Z_k^i) = e^{-\lambda \sum_{j=k+1}^{i-1} \left(W_k^j + R_k^j + w_j + \delta_j c_j\right)} \cdot \mathbb{P}(Z_k^{k+1}),$$
where δ_j is 0 if T_j is not checkpointed, 1 otherwise.

B. $\forall i \geq 1, \mathbb{P}(Z_{i-1}^i) = 1 - \sum_{k=0}^{i-2} \mathbb{P}(Z_k^i)$.

C. $\forall k, 0 \leq k < i$,
$$\mathbb{E}[X_i | Z_k^i] = \mathbb{E}[t\left(W_k^i + R_k^i + w_i; \delta_i c_i; W_i^i + R_i^i - \left(W_k^i + R_k^i\right)\right)],$$
where δ_i is 0 if T_i is not checkpointed, 1 otherwise.

Computing $\mathbb{P}(Z_k^i)$ for $0 \leq k < i-1$. Let Y_k^i be the event "There is no fault during X_{k+1} to X_{i-1} given that there was a fault during X_k." We have:

$$Y_k^i = \{ \bigcap_{j=k+1}^{i-1} \overline{F(X_j)} | F(X_k) \}.$$

Then by definition, $\mathbb{P}(Z_k^i) = \mathbb{P}(Y_k^i) \cdot \mathbb{P}(F(X_k) | T_{i-1}$ is successfully executed). Then we derive $\mathbb{P}(Y_k^i) = e^{-\lambda \sum_{j=k+1}^{i-1} \left(W_k^j + R_k^j + w_j + \delta_j c_j\right)}$. This is because we need to execute $\sum_{j=k+1}^{i-1} \left(W_k^j + R_k^j + w_j + \delta_j c_j\right)$ consecutive units of work without fault by definition of the W_k^i and R_k^i. Also, $\mathbb{P}(F(X_k) | T_{i-1}$ is successfully executed) $= \mathbb{P}(F(X_k))$, as the probability of a fault during X_k is independent of the execution of T_{i-1} since $i - 1 > k$. Finally, one can see that $\mathbb{P}(F(X_k)) = \mathbb{P}(Z_k^{k+1})$ by definition of Z_k^{k+1}.

Computing $\mathbb{P}(Z_{i-1}^i)$ for $i \geq 1$. We have seen that the Z_k^i for $0 \leq k \leq i-1$ partition the set of possibilities. Hence, by definition, $\sum_{k=0}^{i-1} \mathbb{P}(Z_k^i) = 1$. We derive the value of $\mathbb{P}(Z_{i-1}^i)$ from the $i-2$ other values.

Computing $\mathbb{E}[X_i|Z_k^i]$ for $0 \leq k < i$. To compute $\mathbb{E}[X_i|Z_k^i]$, it suffices to see that we need to execute a work of $W_k^i + R_k^i + w_i$ with a checkpoint $\delta_i c_i$. Then, if there is a fault, the recovery cost is $W_i^i + R_i^i$ for a work of w_i, which is identical to having a recovery cost of $W_i^i + R_i^i - \left(W_k^i + R_k^i\right)$ for a work of $W_k^i + R_k^i + w_i$. Hence, using the notation of Equation (1), we obtain that:

$$\mathbb{E}[X_i|Z_k^i] = \mathbb{E}[t\left(W_k^i + R_k^i + w_i; \delta_i c_i; W_i^i + R_i^i - \left(W_k^i + R_k^i\right)\right)]$$

To conclude the proof, we need to show that we can compute the W_k^i and R_k^i values.

Lemma 3. FindWikRik *(Algorithm 2) computes W_k^i and R_k^i in polynomial time for all $i \geq k$.*

Proof. We consider the following invariant H_k^i for FindWikRik:

(H_k^i): At the end of the iteration i of the "for" loop (line 4), for all (j, i') such that $j < k \leq i' < i + 1$, then

- *if $T_j \in T_{i'}^{\downarrow k}$, then*
  - *$tab_k.(i').(j) \in \{1, 2\}$ (1 if T_j is not checkpointed, 2 otherwise),*
  - *for $i'' > i', tab_k.(i'').(j) = 0$ (0 means $T_j \notin T_{i''}^{\downarrow k}$ because $T_j \in T_{i'}^{\downarrow k}$),*
- *else,*
  - *if there exists $l < i'$, and $T_j \in T_l^{\downarrow k}$, then $tab_k.(i').(j) = 0$,*
  - *else $tab_k.(i').(j) = -1$.*

For all (j, l) such that $l > i > j$, and $T_j \in T_l^{\downarrow k}$, then $tab_k.(l).(j) = -1$.

To establish the invariant, we first introduce the following definition:

Definition 2 (path of T_j in $T_i^{\downarrow k}$)**.** Let $T_j \in T_i^{\downarrow k}$, then $T_j = T_{p_0}, T_{p_1}, \ldots, T_{p_l} = T_i$ is a path of T_j in $T_i^{\downarrow k}$ of length l, if

(i) $l = 1$, or

(ii) $T_{p_1} \in T_i^{\downarrow k}$, T_{p_1} is not checkpointed and $T_{p_1}, \ldots, T_{p_l} = T_i$ is a path of T_{p_1} in $T_i^{\downarrow k}$ of length $l - 1$.

We define the distance $l_j^{(i,k)}$ of T_j in $T_i^{\downarrow k}$ as the minimal length of a path of T_j in $T_i^{\downarrow k}$.

Here are some preliminary remarks before starting the proof:

- Once a value of tab_k is set, it is never modified by Traverse (the switch on line 19).
- If $tab_k.(i').(j) \in \{1, 2\}$, then for all $i'' > i', tab_k.(i'').(j) = 0$. Indeed, $tab_k.(i').(j)$ is only set to 1 or 2 in the switch line 19, and when it is the first step of this switch (line 25) is to set $tab_k.(i'').(j)$ to 0 for all $i'' > i'$.
- The only calls Traverse (j, i, k, tab_k) are for $j = i$ or $T_j \in T_i^{\downarrow k}$ and T_j not checkpointed. Hence for $T_{j'} \in$Pred(T_j), either $T_{j'} \in T_i^{\downarrow k}$ or $\exists l < i, T_{j'} \in T_l^{\downarrow k}$. This shows that for all (j, l) such that $l > i > j$, and $T_j \in T_l^{\downarrow k}$, then $tab_k.(l).(j) = -1$ since we will never visit such a node during iteration i of the "for" loop.

We are now ready to prove the invariant by induction. Let us show that H_k^i holds for $i \geq k$.

Let us show H_k^k. At the beginning of the "for" iteration (line 4), for $i = k$, $tab_k.(k).(j) = -1$. We show that H_k^k holds for all tasks in $T_k^{\downarrow k}$ (the case for tasks not in $T_k^{\downarrow k}$ is trivial), and do this by induction on their distance (as defined in Definition 2) in $T_k^{\downarrow k}$.

First, we verify that for all predecessors T_j of T_k whose distance is 1 in $T_k^{\downarrow k}$, the call Traverse (k, k, k, tab_k) checks whether $T_j \in T_k^{\downarrow k}$ (answer, yes) and has not been studied (the switch on line 19). If it is the case, then it assigns 1 or 2 to $tab_k.(k).(j)$, and then calls Traverse (j, k, k, tab_k) if and only if T_j is not checkpointed. Then there is a call Traverse (j, k, k, tab_k) for all not-checkpointed elements of $T_k^{\downarrow k}$ whose distance is 1 in $T_k^{\downarrow k}$.

Let us now assume H_k^k holds for all $T_j \in T_k^{\downarrow k}$ such that $l_j^{(i,k)} = l$. Let us show the result for all $T_{j'} \in T_k^{\downarrow k}$ such that $l_{j'}^{(i,k)} = l + 1$. Let $T_{j'}, T_{p_1}, \ldots, T_{p_l} = T_k$ path of $T_{j'}$ in $T_k^{\downarrow k}$ of length $l + 1$. Then when T_{p_1} was studied, by hypothesis because it is not checkpointed, there was a call Traverse

(p_1, k, k, tab_k). Because $T_{j'}$ is a direct predecessor of T_{p_1}, then either its value in tab_k was already set to 1 or 2 through another path or it was set to -1 and this call has set it up to 1 or 2. By induction we obtain H_k^k.

Assuming $\forall k \leq i' < i, H_k^{i'}$, *let us show* H_k^i. First note that H_k^{i-1} gives us (i) if there exists $l < i'$, and $T_j \in T_l^{\downarrow k}$, then $tab_k.(i').(j) = 0$, and (ii) $\forall j, T_j \in T_i^{\downarrow k}$, then at the beginning of iteration i, $tab_k.(i).(j) = -1$. Furthermore, with the preliminary remark, to show H_k^i, we simply need to show that for all $j < k$,

- if $T_j \in T_i^{\downarrow k}$, then $tab_k.(i).(j) \in \{1, 2\}$ (1 if T_j is not checkpointed, 2 otherwise),
- else, if for all $l < i$, $T_j \notin T_l^{\downarrow k}$, then $tab_k.(i).(j) = -1$.

The proof can be done by induction and is similar to H_k^k. The first call TRAVERSE (i, i, k, tab_k) makes sure that this is true for all predecessors T_j of T_i whose distance is 1 in $T_i^{\downarrow k}$ (the only reason why a predecessor T_j of T_i would not be in $T_i^{\downarrow k}$ is if $\exists l < i, T_j \in T_l^{\downarrow k}$, and in that case by induction hypothesis, $tab_k.(i).(j) = 0$). Then there is a call to TRAVERSE only for the predecessor tasks $T_j \in T_i^{\downarrow k}$ that are not checkpointed.

Finally, H_k^n gives the correctness of Algorithm 2, whose complexity is $O(n^3)$. □

Altogether, Algorithm 2 is invoked for each task, and the complexity of the whole evaluation method is $O(n^4)$. □

Because we can compute the expected makespan of a schedule, a schedule of a DAG is a sufficient certificate to verify whether the expected makespan is below a certain threshold. Hence we have derived the following result:

Corollary 2. *The decision problem associated to* DAG-CHKPTSCHED *is in NP for general DAGs (and is NP-complete by Theorem 2).*

6 Heuristics for general DAGs

In this section, we develop polynomial-time heuristics in the case of general DAGs. A heuristic that computes a schedule for a given instance of DAG-CHKPTSCHED must answer two questions: (i) how should the DAG be linearized? and (ii) which tasks should be checkpointed? To answer the first question, we consider three possible linearization strategies: Depth First (DF), Breadth First (BF), and Random First (RF). For DF and BF, we prioritize the tasks by decreasing outweight (i.e., the sum of the weights of the task's successors). The rationale is that tasks that have "heavy" subtrees should be executed first.

To answer the second question, we propose four checkpointing strategies. The first and second strategies are baseline comparators, and correspond to either never checkpointing (CKPTNVR) or always checkpointing (CKPTALWS). For both these strategies, we only consider the DF linearization. A DF linearization makes sense when no checkpoints are taken because one should make progress toward exit tasks aggressively rather than pursuing multiple exit tasks simultaneously (which is risky in the presence of failures). The choice of the DAG linearization is inconsequential when all tasks are checkpointed.

The third and fourth strategies fix the total number of checkpoints taken throughout application execution, say N, and checkpoint N tasks based on some criteria. Then they do an exhaustive search for the N value, $N = 1, \ldots, n-1$ (recall that n is the number of tasks), that achieves the lowest expected makespan, which is computed in polynomial time as explained in Section 5.

In the third strategy, tasks are sorted by decreasing w_i (checkpoint first the tasks whose computations are the longest), by increasing c_i (checkpoint first the tasks whose checkpointing overheads are the shortest), or by decreasing d_i, the sum of the weights of the successors (checkpoint first the tasks whose successors are more likely to fail). The top N tasks taken in these orders are checkpointed. We name the three versions of this strategy CKPTW, CKPTC, CKPTD.

Algorithm 2 FINDWIKRIK

```
1: procedure FindWikRik(k)
2:   tab_k: n × n array initialized with -1
3:   W_k, R_k: n arrays initialized with 0
4:   for i = k ... n do
5:     tab_k =Traverse (i, i, k, tab_k)
6:     for j = 1 ... k − 1 do
7:       switch tab_k.(i).(j) do
8:         case 1
9:           W_k.(i) ← W_k.(i) + w_j
10:        case 2
11:          R_k.(i) ← R_k.(i) + r_j
12:    end for
13:  end for
14:  Return W_k,R_k
15: end procedure
16:
17: procedure Traverse(l, i, k, tab_k)
18:   for T_j ∈ Pred(T_l) do
19:     switch tab_k.(i).(j) do
20:       case 0                                   ▷ ∃i' < i, T_j ∈ T_{i'}^{↓k}
21:         Do nothing
22:       case 1,2                                 ▷ T_j ∈ T_i^{↓k}, already studied
23:         Do nothing
24:       case -1                                  ▷ T_j ∈ T_i^{↓k}, not yet studied
25:         for r = i + 1 ... n do
26:           tab_k.(r).(j)←0                      ▷ T_j ∈ T_i^{↓k} ⟹ T_j ∉ T_r^{↓k}
27:         end for
28:         if j < k then
29:           if T_j is checkpointed then
30:             tab_k.(i).(j) ← 2
31:           else
32:             tab_k.(i).(j) ← 1
33:             tab_k =Traverse (j, i, k, tab_k)
34:           end if
35:         else
36:           tab_k.(i).(j) ← 0
37:         end if
38:   end for
39:   Return tab_k
40: end procedure
```

The fourth strategy, CkptPer, relies on the idea of periodic checkpointing [2, 3]. Given a linearization of the DAG, consider a failure-free execution. If W is the sum of the w_i values over all tasks, CkptPer checkpoints the task that completes the earliest after time $x \times W/N$ for $x = 1, \ldots, N-1$. While periodic checkpointing is a typical approach for data-parallel computation, it does not account for the structure of the DAG.

Heuristic names are concatenations of the name of the linearization strategy and of the checkpointing strategy (e.g., RF-CkptC). Combining the three linearization strategies (DF, BF, RF) and the checkpointing strategies, we have a total of 14 heuristics.

Unfortunately, there are no heuristics in the literature to which we can compare the above heuristics. This is because no method to evaluate the expected makespan of a schedule was available before this work, thus precluding the design (and the straightforward evaluation) of reasonable heuristics.

7 Experimental evaluation

In this section, we present experimental results that quantify the performance of the heuristics in Section 6. The source-code (implemented in OCaml) and all input and output data are publicly available at [22].

7.1 Experimental methodology

To evaluate the heuristics with representative DAGs, we use the Pegasus Workflow Generator (PWG) [9, 23]. PWG uses the information gathered from actual executions of scientific workflows as well as domain-specific knowledge of these workflows to generate representative and realistic synthetic workflows. We consider four different workflows generated by PWG (information on the corresponding scientific applications is available in [23, 24]):

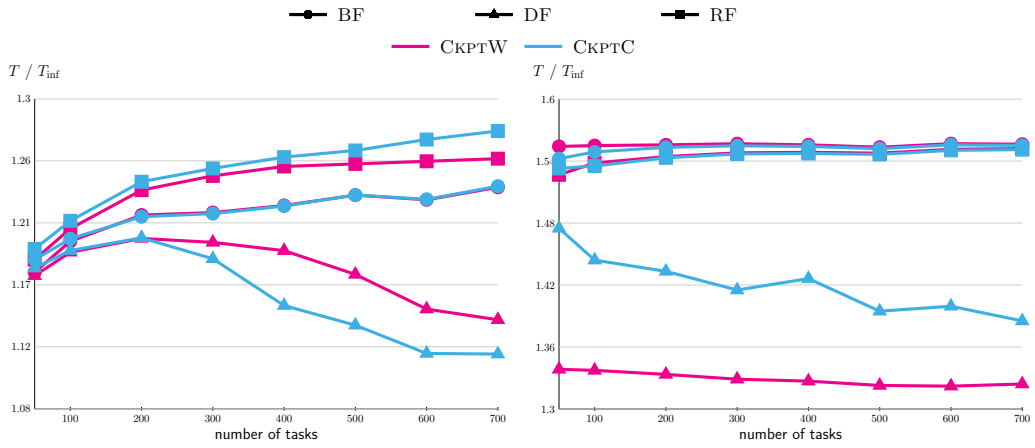
- Montage: The NASA/IPAC Montage application stitches together multiple input images to create custom mosaics of the sky. The average weight of a Montage task is 10s.
  Structurally, Montage is a three-level graph [25]. The first level (reprojection of input image) consists of a bipartite directed graph. The second level (background rectification) is a bottleneck that consists in a join followed by a fork. Then the third level (co addition to form the final mosaic) is simply a join.
- Ligo: LIGO's Inspiral Analysis workflow is used to generate and analyze gravitational waveforms from data collected during the coalescing of compact binary systems. The average weight of a Ligo task is 220s.
  Structurally, Ligo can be seen as a succession of Fork-Joins super tasks, that contain themselves either fork-join graphs or bipartite graphs (see the LIGO IHOPE workflow on [23]).
- CyberShake: The CyberShake workflow is used by the Southern California Earthquake Center to characterize regional earthquake hazards. The average weight of a CyberShake task is 25s.
  Structurally, CyberShake is harder to categorize than the other graphs. It's first part is less structured, but end with many parellel linear chains (see the CyberShake workflow on [23]).
- Genome: The epigenomics workflow created by the USC Epigenome Center and the Pegasus team automates various operations in genome sequence processing. The average weight of a Genome task depends on the number of tasks and is greater than 1000s.
  Structurally, Genome starts with many parallel fork-join graphs, whose exit tasks are then both joined into an new exit task and generate new fork graphs (see the Epigenomics workflow on [23]).

In all experiments, $c_i = r_i$ (checkpoint and recovery costs are identical for a task) and $D = 0$ (downtime is zero seconds). We focus mainly on the particular case where $c_i = 0.1w_i$, and for a MTBF of 10^3s (except for Genome where the average weight of each task is significantly longer than for other DAGs, in which case we consider a MTBF of 10^4s). We vary the number of tasks in each workflow from 50 to 700. Unless stated otherwise, the figures show the number of tasks on the horizontal axis and the ratio of the expected execution time (T) over the execution time of

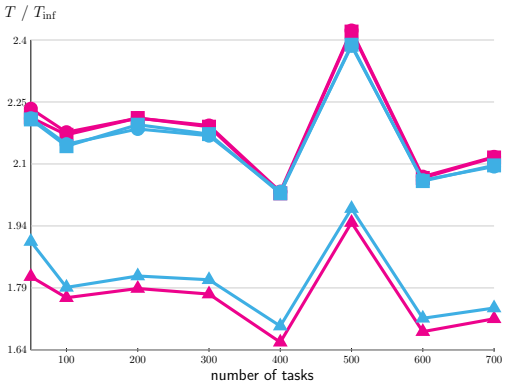
a failure-free, checkpoint-free execution (T_{inf}) on the vertical axis (lower values are better). The expected execution time T is computed using the method described in Section 5.

7.2 Results

We find that our results strongly depend on the structure of the DAG, meaning that the relative performance of the heuristics vary between each workflow type. Consequently, we do not show results aggregated over all workflows. The goal of our experiments is to determine for each workflow (i) which DAG linearization strategy is best, and (ii) which checkpointing strategy is best, hoping to identify strategies that work well across different workflows.



(a) CyberShake: $\lambda = 0.001$, $c_i = 0.1w_i$. (b) Ligo: $\lambda = 0.001$, $c_i = 0.1w_i$.

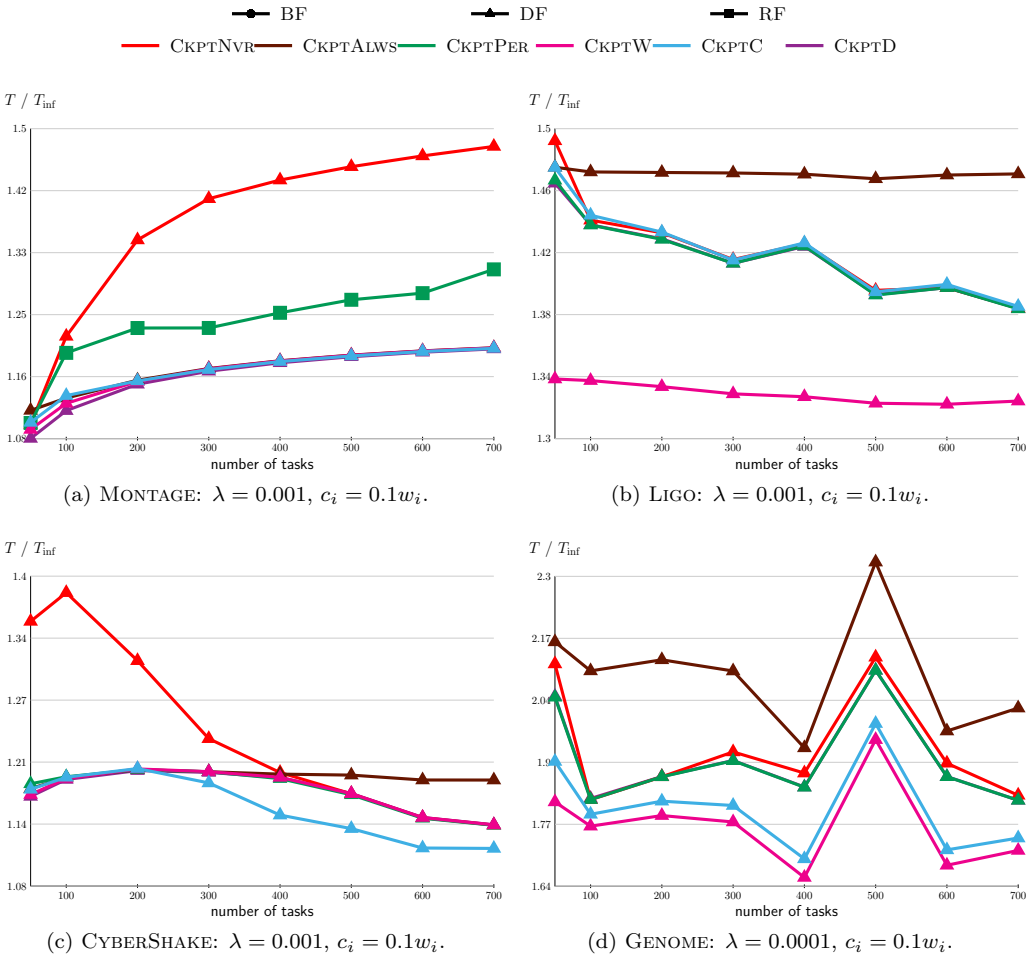


(c) Genome: $\lambda = 0.0001$, $c_i = 0.1w_i$.

Figure 5: Impact of the linearization strategy.

Linearization strategies – Figure 5 shows results for the CyberShake, Ligo, and Genome workflows for two checkpointing strategies, CkptW and CkptC, and for all three linearization strategies. CkptW and CkptC are the best checkpointing strategies in our results (see the discussion of the results in Figure 6 hereafter). Figure 5 does not show results for the Montage workflow. For this workload, the choice of the linearization strategy has almost no impact on the results (at most a 1% relative difference). Overall, the DF linearization is almost always the best. This makes sense as this strategy stipulates that if some work that depends on the most recently completed work can be done, then it should be done. Otherwise, by following a different branch of the workflow, one risks losing that recent work and having to do it again (or recover it). The only case where DF is not the best linearization approach is for the Montage DAG and the CkptPer heuristic (see Figure 6a). We have no explanation but since CkptPer is the worst checkpointing strategy for that workflow,

this result is not particularly relevant. Finally, it is interesting to see in Figure 5b that, for the LIGO workflow, RF performs better than BF. This is because RF sometimes can be close to a DF strategy.



(a) MONTAGE: $\lambda = 0.001$, $c_i = 0.1w_i$.

(b) LIGO: $\lambda = 0.001$, $c_i = 0.1w_i$.

(c) CYBERSHAKE: $\lambda = 0.001$, $c_i = 0.1w_i$.

(d) GENOME: $\lambda = 0.0001$, $c_i = 0.1w_i$.

Figure 6: Impact of the checkpointing strategy. For each checkpointing strategy, we plot the best linearization strategy.

Checkpointing strategies – Figure 6 shows results for all four workflows. For each checkpointing strategy, we only show results for the linearization strategy that leads to the best results (the line symbols indicate which linearization strategy is used). First, we note that our checkpointing heuristics always perform better than the two baseline comparators, CKPTNVR and CKPTALWS. Second, an interesting (but expected) result is that CKPTPER does not behave well, and sometimes even worse than CKPTNVR or CKPTALWS. The CKPTPER approach was specifically designed in the literature for divisible applications. As such, it does not account for the structure of the DAG. This causes it to make poor checkpointing choices. For instance, consider the example workload in Figure 1 with the linearization T_0, T_3, T_1, T_2, etc. It makes sense to checkpoint T_3 before executing T_1, which is an entry task. But CKPTPER may checkpoint T_1 instead because $w_0 + w_3 + w_1$ happens to correspond to the chosen checkpointing period. The main result from Figure 6 is that two checkpointing strategies outperform the other strategies: CKPTW (for MONTAGE, LIGO and GENOME) and CKPTC (for CYBERSHAKE). These two heuristics behave very differently because we have $c_i = 0.1w_i$. CKPTW checkpoints the tasks by decreasing weight (hence by decreasing checkpointing time since it is proportional to the weight of the tasks), while CKPTC checkpoints the tasks by increasing checkpointing time (hence increasing weight). The good performance of both

heuristics in different scenarios can be explained intuitively. After finishing a long/large task, it is useful to checkpoint it as quickly as possible in case a failure occurs soon (which is what CkptW does). Conversely, checkpointing a short/small task (which may be the successor of a long task) is also useful because its checkpointing time is low (which is what CkptC does).

Examining the structure of the DAGs helps understanding the results. Indeed, Montage, Ligo and Genome are all very parallel task graphs composed of successive fork and joins. Intuitively, while this is not necessarily the optimal strategy, it makes sense to checkpoint the large tasks first when considering a join: indeed, when executed, the entry tasks of a join are put on hold while the remaining are executed, which may increase the likeliness that a failure occurs. Hence checkpointing the large tasks first may be a good strategy. On the contrary, Montage is the only DAG that does not consist in successive fork-joins, but instead comprises linear chains. When executing a linear chain, it can make sense to save only the smallest output data. In particular, when the checkpointing time depends on the size of the task, if a big task is followed by a smaller task, then it might be cheaper to execute and checkpoint the smallest task than to checkpoint the bigger task. This could explain the fact that CkptC performs well for Montage. In summary, a good strategy should be workflow-dependent and take into account the structural shape of the DAG (fork joins, linear chains).

To verify the impact of the checkpointing overhead model on the above results, Figure 7 presents similar results but with a lower $c_i = 0.01w_i$. A noticeable and expected effect of the lowered checkpointing costs is that CkptAlws performs slightly better. Otherwise, these results confirm the same trends and conclusions as that with the results shown in Figure 6.

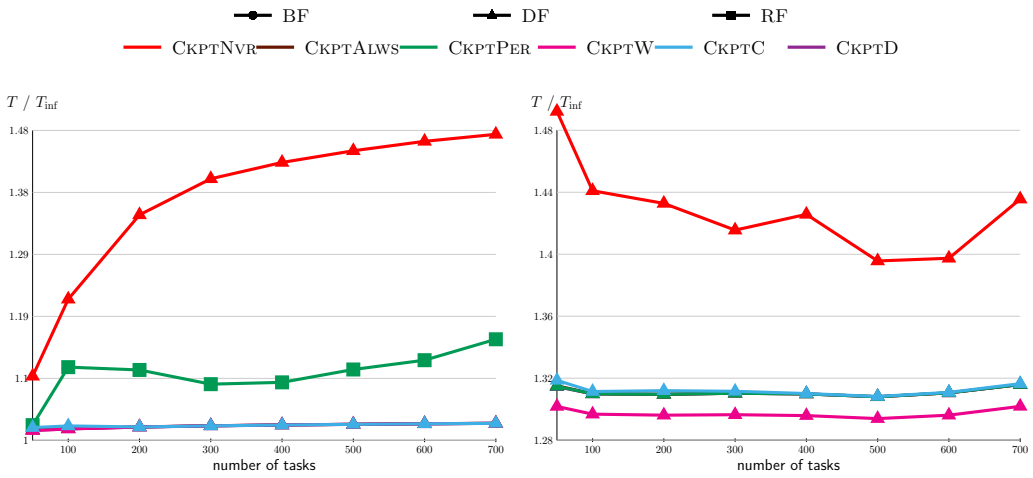
Constant checkpoint overhead – To better assess the impact of checkpointing costs, we discuss results with a constant checkpoint cost, i.e., checkpointing costs that are independent of task weights. Expectedly, when CkptW performs better with proportional checkpoint costs it also perform better with constant checkpoint costs. This is because the ratio of the amount of computation that risks being lost over the checkpointing time will be even more beneficial to large tasks. However, for workflows where CkptC performs better, the question of the impact of constant checkpointing costs is interesting. Figure 8 shows results for CyberShake that allow a comparison of CkptW and CkptC when the checkpointing cost is constant (for $c_i = 10$s and $c_i = 5$s). This plot can be compared to Figure 5a where the checkpoint is proportional to the computation. We can see that when the checkpointing cost is constant, CkptW tends to behave as well as CkptC on CyberShake workflows.

We also evaluate the impact of the checkpointing strategy with a constant checkpointing cost ($c_i = 5$s) in Figure 9 for all four workflow types. Again, the same conclusions hold as observed previously, although CkptAlws becomes more costly for the Montage and CyberShake workflows.

Impact of λ – Finally, we wish to assess the impact of the MTBF on our conclusions. Figure 10 shows results for $n = 200$ nodes vs. λ. We observe the same general trends as reported above, i.e., DF-CkptW is the winning strategy in most cases regardless of the value of λ.

Summary – We have compared our heuristics over a range of experimental scenarios. In general, DF-CkptW leads to the best results, which in practice would translate to shorter makespans. DF-CkptC performs well in some cases. These performance differences depend on the structure of the DAG (CkptW performs better for fork-joins while CkptC performs better for linear chains). These differences can be discovered empirically, as done in this section, or perhaps by analyzing the underlying shape of the DAG.

Overall, the heuristics that rely on the computation of the expected makespan given in Section 5 lead to significantly better results than the baseline CkptAlws and CkptNvr approaches. A significant, if expected finding, is that taking into account the structure of the DAG is important, as highlighted by the poor results of the CkptPer heuristic.



(a) Montage: $\lambda = 0.001$, $c_i = 0.01w_i$.

(b) Ligo: $\lambda = 0.001$, $c_i = 0.01w_i$.

(c) CyberShake: $\lambda = 0.001$, $c_i = 0.01w_i$.

(d) Genome: $\lambda = 0.0001$, $c_i = 0.01w_i$.

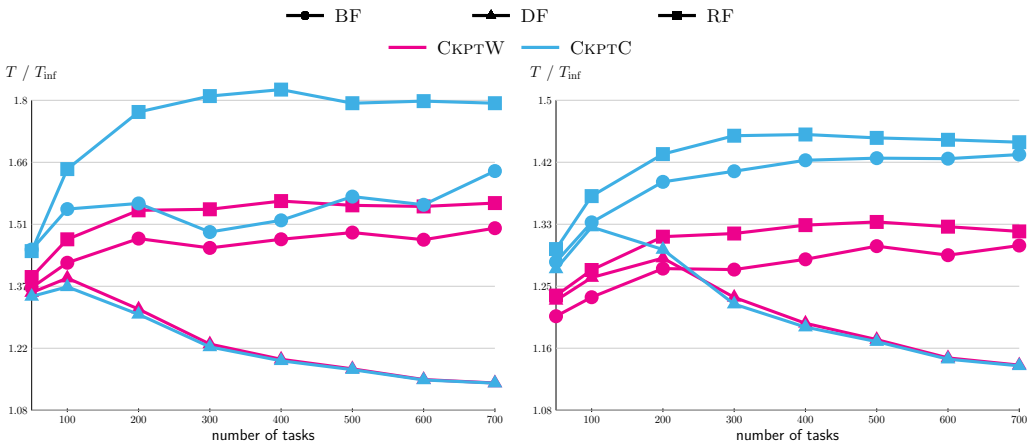
Figure 7: Impact of the checkpointing strategy when $c_i = 0.01w_i$. For each checkpointing strategy, we plot the best linearization strategy.

8 Conclusion

In this work, we have studied the problem of scheduling computational workflows on a failure-prone platform. We have used a framework where applications are scheduled on the full platform where processors are subject to i.i.d. exponentially distributed failures. Checkpoint-rollback-recovery is used to tolerate failures. Our main contribution over previous work [13, 19] is that we consider general Directed Acyclic Graphs instead of linear chains. Our theoretical results include polynomial-time algorithms for fork DAGs and for some join DAGs (when the checkpoint and recovery costs are constant) and the intractability of the problem for join DAGs in general. We have also discussed the complexity of the simple problem instance where no task is checkpointed.

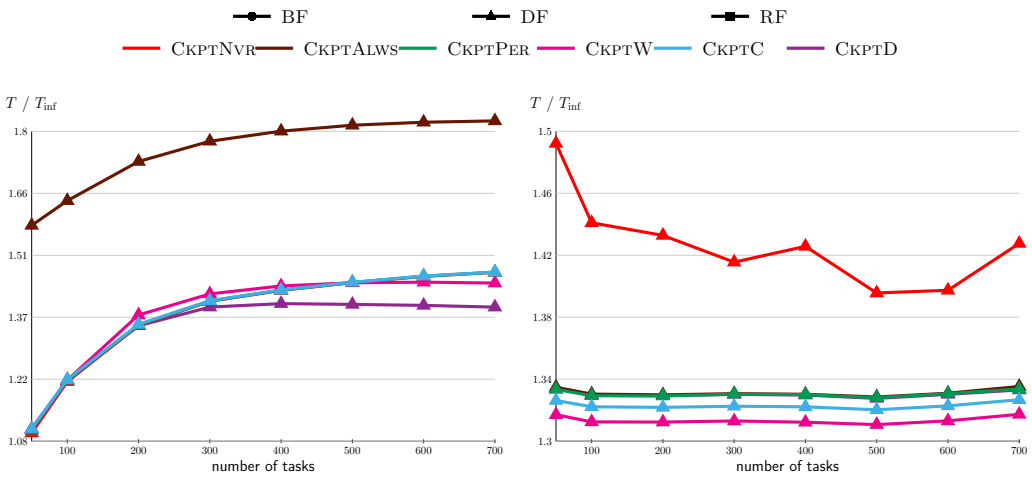
Our main theoretical result is a polynomial-time algorithm to evaluate the expected makespan of a schedule for general DAGs. This is a key result as it makes it possible to design heuristics for general DAGs, i.e., heuristics that can construct a schedule with a known objective. Without this result, the only way to attempt to find a good schedule would be to run numerous and likely prohibitively time-consuming Monte-Carlo experiments with a fault generator (either in simulation or on a real platform).

We have proposed several heuristics and have evaluated them for four representative scientific workflow configurations. Overall, we find that DAGs should be traversed depth-first (DF) and that checkpointing should be done by prioritizing tasks based on weight (CkptW) or checkpointing cost (CkptC). The two resulting heuristics, DF-CkptW and DF-CkptC perform differently on

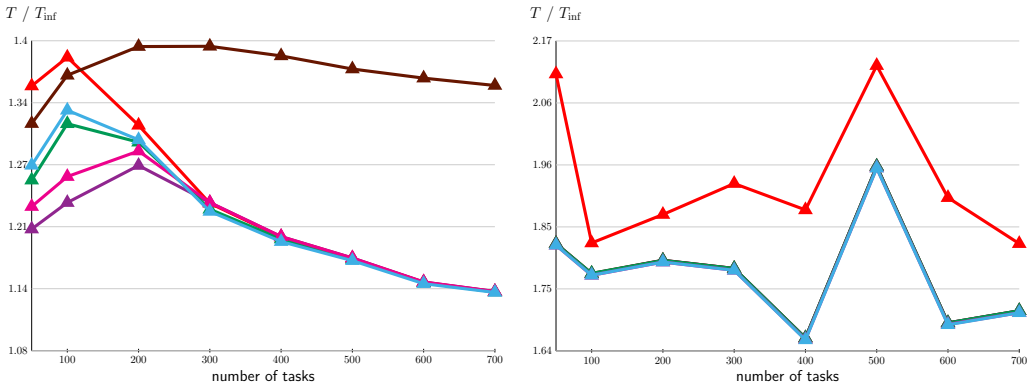


(a) CyberShake: $\lambda = 0.001$, $\forall i,\ c_i = 10s$. (b) CyberShake: $\lambda = 0.001$, $\forall i,\ c_i = 5s$.

Figure 8: Impact of the linearization strategy for a constant checkpoint.

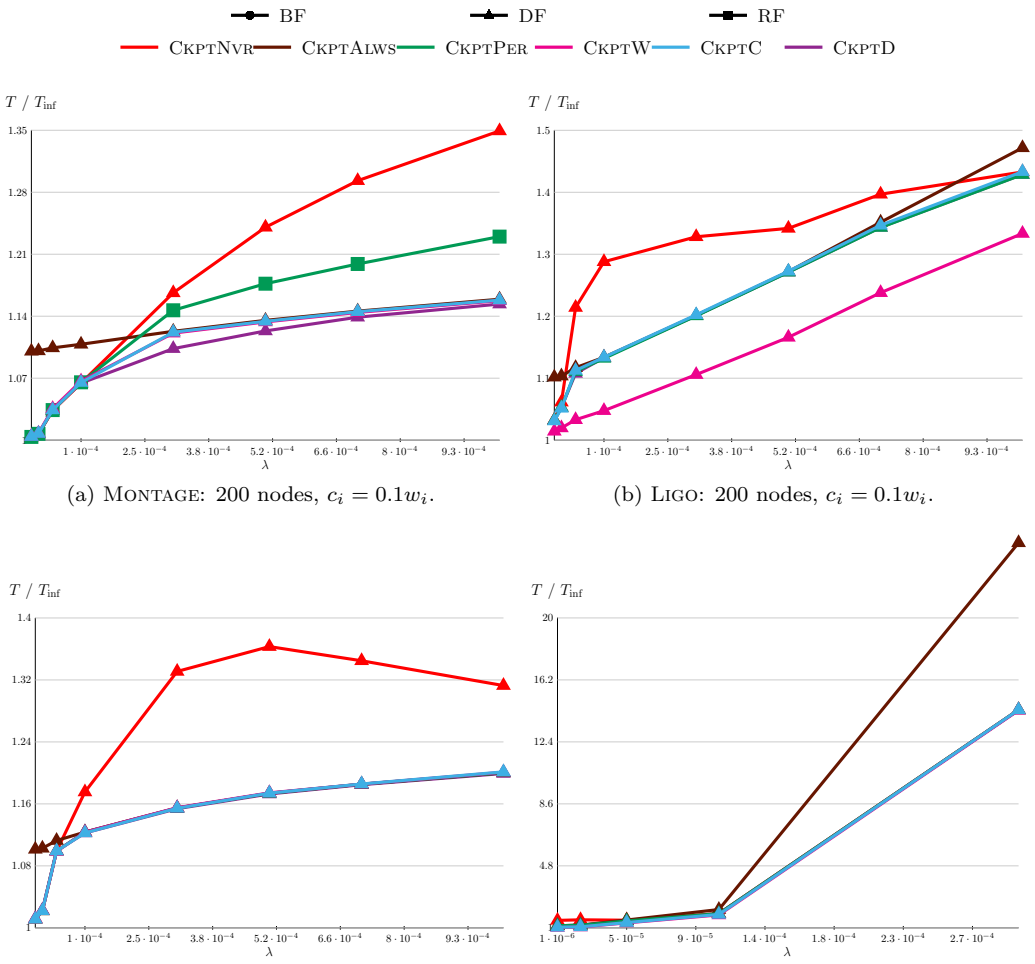


(a) Montage: $\lambda = 0.001$, $c_i = 5s$. (b) Ligo: $\lambda = 0.001$, $c_i = 5s$.



(c) CyberShake: $\lambda = 0.001$, $c_i = 5s$. (d) Genome: $\lambda = 0.0001$, $c_i = 5s$.

Figure 9: Impact of the checkpointing strategy when $c_i = 5s$. For each checkpointing strategy, we plot the best linearization strategy.



(a) Montage: 200 nodes, $c_i = 0.1w_i$.

(b) Ligo: 200 nodes, $c_i = 0.1w_i$.

(c) CyberShake: 200 nodes, $c_i = 0.1w_i$.

(d) Genome: 200 nodes, $c_i = 0.1w_i$.

Figure 10: Impact of the checkpointing strategy with different values of λ. For each checkpointing strategy, we plot the best linearization strategy.

different workflows depending on the DAG structure. We found that a periodic checkpointing approach, although widely used for divisible applications, is not effective, precisely because it does not account for the structure of the DAG.

A future direction for this work is to consider non-blocking checkpointing operations, i.e., a processor can compute a task, perhaps at a reduced speed, while checkpointing a previously executed task. Overlapping of computation and checkpointing can improve performance, but changes the problem. In particular, it would be interesting to see how our theoretical results are impacted when considering non-blocking checkpointing. A broader future direction would be to remove the assumption that the DAG is linearized, i.e., that each task executes on the entire platform. The scheduling problem then becomes much more complex since one must decide how many processors are allocated to each task, and possibly account for data redistribution costs.

Acknowledgments This work was supported in part by the French Research Agency (ANR) through the Rescue project. Yves Robert is with Institut Universitaire de France.

References

[1] J. Dongarra *et al.*, "The International Exascale Software Project," *Int. J. High Performance Computing App.*, vol. 23, no. 4, pp. 309–322, 2009.

[2] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Communications of the ACM*, vol. 17, no. 9, pp. 530–531, 1974.

[3] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *FGCS*, vol. 22, no. 3, pp. 303–312, 2006.

[4] E. Gelenbe and D. Derochette, "Performance of rollback recovery systems under intermittent failures," *Communications of the ACM*, vol. 21, no. 6, pp. 493–499, 1978.

[5] A. Bouteiller, P. Lemarinier, K. Krawezik, and F. Capello, "Coordinated checkpoint versus message log for fault tolerant MPI," in *Cluster Computing*. IEEE Computer Society Press, 2003, pp. 242–250.

[6] K. M. Chandy and L. Lamport, "Distributed snapshots : Determining global states of distributed systems," in *Transactions on Computer Systems*, vol. 3(1). ACM, February 1985, pp. 63–75.

[7] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Survey*, vol. 34, pp. 375–408, 2002.

[8] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, "Adaptive incremental checkpointing for massively parallel systems," in *Proc. ICS '04*. ACM, 2004.

[9] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi, "Characterization of scientific workflows," in *Workflows in Support of Large-Scale Science (WORKS 2008)*. IEEE, 2008, pp. 1–10.

[10] S. Chakrabarti, J. Demmel, and K. Yelick, "Modeling the benefits of mixed data and task parallelism," in *Proc. SPAA'95*. ACM, 1995.

[11] P. Dutot, L. Eyraud, G. Mounié, and D. Trystram, "Scheduling on large scale distributed platforms: from models to implementations," *Int. J. Found. Comput. Sci.*, vol. 16, no. 2, pp. 217–237, 2005.

[12] F. Suter, "Scheduling delta-critical tasks in mixed-parallel applications on a national grid," in *Int. Conf. Grid Computing (GRID 2007)*. IEEE, 2007.

[13] S. Toueg and Ö. Babaoglu, "On the optimum checkpoint selection problem," *SIAM J. Comput.*, vol. 13, no. 3, pp. 630–649, 1984.

[14] M.-S. Bouguerra, T. Gautier, D. Trystram, and J.-M. Vincent, "A flexible checkpoint/restart model in distributed systems," in *PPAM*, vol. LNCS 6067, 2010.

[15] Y. Ling, J. Mi, and X. Lin, "A variational calculus approach to optimal checkpoint placement," *IEEE Trans. Computers*, pp. 699–708, 2001.

[16] T. Ozaki, T. Dohi, H. Okamura, and N. Kaio, "Distribution-free checkpoint placement algorithms based on min-max principle," *IEEE TDSC*, pp. 130–140, 2006.

[17] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien, "Checkpointing strategies for parallel jobs," in *Proc. SC'2011*. ACM, 2011.

[18] E. Gelenbe and M. Hernández, "Optimum checkpoints with age dependent failures," *Acta Informatica*, vol. 27, no. 6, pp. 519–531, 1990.

[19] M.-S. Bouguerra, D. Trystram, and F. Wagner, "Complexity Analysis of Checkpoint Scheduling with Variable Costs," *IEEE Trans. Computers*, vol. 62, no. 6, pp. 1269–1275, 2013.

[20] Y. Robert, F. Vivien, and D. Zaidouni, "On the complexity of scheduling checkpoints for computational workflows," in *Proc. of the Dependable Systems and Networks Workshop*, 2012, pp. 1–6.

[21] M. R. Garey and D. S. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[22] G. Aupy, "Source code and data." https://github.com/Gaupy/linear-workflows, 2014.

[23] Pegasus, "Pegasus workflow generator." https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator, 2014.

[24] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and profiling scientific workflows," *Future Generation Computer Systems*, vol. 29, no. 3, pp. 682–692, 2013.

[25] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good *et al.*, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.