Scheduling computational workflows on failure-prone platforms

Guillaume Aupy*, Anne Benoit*, Henri Casanova† and Yves Robert*‡
*École Normale Supérieure de Lyon, France
Email: {guillaume.aupy|anne.benoit|yves.robert}@ens-lyon.fr
†University of Hawai‘i, USA. Email: henric@hawaii.edu
‡University of Knoxville, USA

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 whether to checkpoint or not checkpoint a task after it completes. We give a polynomial-time algorithm for fork graphs and show that the problem is NP-complete with join graphs. Our main result is a polynomial-time algorithm to compute the execution time of a workflow with specified to-be-checkpointed tasks. Using this algorithm as a basis, we propose efficient heuristics for solving the scheduling problem. We evaluate these heuristics for representative workflow configurations.

1. Introduction

Resilience has become a key concern when computing at large scales [1]. Enrolling more processors in an application execution leads to more frequent application failures. (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.) First, 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 use commercial-off-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. Second, when enrolling p processors to execute a tightly-coupled parallel application, a failure on any of the processor 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 implemented in a way that is 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 large-scale platforms, i.e., subject to processor failures during application execution. An application is structured as 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 graphs of parallel tasks, or applications with "mixed parallelism", without considering processor failures, has long been recognized [10]. The difficulty 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 both tasks do not necessarily 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 use checkpoint-roll-back recovery within each task, it would require either saving large checkpoints (application-agnostic) or to modify 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 given failure rate on which we want to execute the application. In which order should the tasks be executed? Which tasks should be checkpointed? We call an answer to these two questions 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 states of all its predecessors, which depend upon when the last failure has occurred. In this context, we make the following contributions:

- We provide a polynomial-time algorithm for computing the expected makespan of a schedule. This algorithm is the fundamental basis for designing and comparing heuristics that find efficient schedules for arbitrary DAGs.
- We propose a set of heuristics for solving DAG-CHKPTSCHED for general DAGs. 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]).
- We show that although DAG-CHKPTSCHED can be solved in polynomial time for fork DAGs, its associated decision problem is NP-complete for join DAGs. 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.

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 framework and all model parameters. Section 4 gives our main theoretical contributions. Section 5 presents a set of heuristics for solving the problem with DAGs. These heuristics are evaluated experimentally in Section 6. Finally, Section 7 outlines 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 and 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 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 the checkpoints 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. 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 hard to determine which tasks to re-execute and which tasks to recover from after one or more failures have occurred during the execution.

3. Framework

We consider a (subset of a) parallel platform with p processors, where each processor is a processing element that is

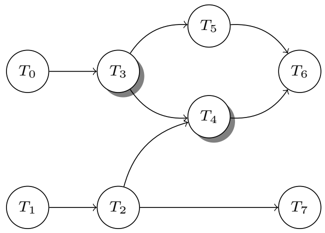


Figure 1: Example DAG. Tasks whose output is checkpointed (T_3 and T_4) are shadowed.

subject to its own individual failures. When a failure occurs at a processor, this 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.

If a failure happens 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. This hints 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 (*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 (*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 accounted for 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 computation 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:

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

We make extensive use of this notation 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. Theoretical results

In this section, we present several theoretical results. First, in Section 4.1, we establish the NP-hardness of DAG-CHKPTSCHED. Then, Section 4.2 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.

4.1. Complexity

Fork DAGs are simple DAGs made of a source task with independent children. Formally $G = (V, E)$ where $V = \{T_0, T_1, \dots, T_n\}$ and $E = \{(T_0, T_i), 1 \le i \le n\}$. Join DAGs are fork DAGs where edges are reversed, hence they are made of independent source tasks and a common sink task.

Theorem 1.

- DAG-CHKPTSCHED *for fork DAGs can be solved in linear time.*
- DAG-CHKPTSCHED *for join DAGs is NP-complete.*

The proof for fork DAGs is simple, because we can show that the ordering of the sink tasks does not matter. The only decision to make is whether to checkpoint the source task. One can compute the expected makespan of both cases and simply pick the case that achieves the lowest expected makespan.

The proof for join DAGs is much more elaborate and involves technical derivations. Due to lack of space, we give here a sketch of the proof, and we refer the reader to the companion research report [21] for the full proof.

To prove the result for join DAGs, we start by denoting by I_{CKPT}, resp. I_{NCKPT}, the subset of $\{T_1, \dots, T_n\}$ composed of the tasks that are checkpointed, resp. not checkpointed. We then prove that in the optimal solution, the tasks from I_{CKPT} should be executed before the tasks from I_{NCKPT}. Furthermore, while the execution order of the tasks from I_{NCKPT} does not matter the tasks from I_{CKPT} must be executed in a specific order (namely in non-increasing values of $g(i)$, where $g(i) = e^{-\lambda(w_i+c_i+r_i)} + e^{-\lambda r_i} - e^{-\lambda(w_i+c_i)}$). Given the two sets $(I_{\text{CKPT}}, I_{\text{NCKPT}})$, we can construct the optimal solution in polynomial time, hence the problem belongs to NP. Finally, we reduce the problem of finding the sets $(I_{\text{CKPT}}, I_{\text{NCKPT}})$ for a given upper bound on the expected makespan to finding a solution to the SUBSET-SUM problem [22].

While this problem is hard in general for join DAGs, there are instances in which it can be solved in polynomial time.

Proposition 1. DAG-CHKPTSCHED *for a join DAG where $c_i = c$ and $r_i = r$ for all i can be solved in quadratic time.*

This result is a corollary from the optimal scheduling orders for I_{CKPT} and I_{NCKPT}. We can show that if there are two tasks T_i and T_j such that $w_i > w_j$ and T_j is checkpointed in the optimal solution, then T_i should be checkpointed. This gives us only n possible sets I_{CKPT}. Evaluating the expected execution time for each set gives us the optimal solution.

4.2. Evaluating a schedule for a general DAG

In this section, we consider a general DAG and a given schedule that specifies a linearization of the DAG and which tasks are checkpointed. For simplicity, we renumber the tasks so that task T_i is the i^{th} task executed in the linearization.

Theorem 2. *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:

$$Z_k^i = \bigcap_{j=k+1}^{i-1} \overline{F(X_j)} \bigcap F(X_k) \tag{2}$$

(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 $0 \le k \le i-1$ partitions the set of possibilities for X_i. Hence we can write

$$\mathbb{E}[X_i] = \sum_{k=0}^{i-1} \mathbb{P}(Z_k^i)\mathbb{E}[X_i | Z_k^i]. \tag{3}$$

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 $j < k \le i$, then we say that $T_j \in T_i^{\downarrow k}$, if for all $k \le l < i$, $T_j \notin T_l^{\downarrow k}$, 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 is 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 is 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 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. 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.

[A] Let us compute $\mathbb{P}(Z_k^i)$ *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$) = \mathbb{P}(F(X_k))$, indeed, 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}.

[B] Let us compute $\mathbb{P}(Z_{i-1}^i)$ *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.

[C] Let us compute $\mathbb{E}[X_i|Z_k^i]$ *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 1. FINDWIKRIK *(Algorithm 1) 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 1, whose complexity is $O(n^3)$. □

Altogether, Algorithm 1 is called 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 1. *The decision problem associate to* DAG-CHKPTSCHED *is in NP for general DAGs (and is NP-complete from Theorem 1).*

5. 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 out-weight (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 sink tasks aggressively rather than pursuing multiple sink tasks simultaneously (which is risky in the presence of failures). The choice of the DAG linearization is inconsequential when all tasks are checkpointed.

Algorithm 1 FINDWIKRIK

```
procedure FINDWIKRIK($k$)
    $tab_k$: $n \times n$ array initialized with -1
    $W_k$, $R_k$: $n$ arrays initialized with 0
    for $i = k \ldots n$ do
        $tab_k$ =TRAVERSE $(i, i, k, tab_k)$
        for $j = 1 \ldots k-1$ do
            switch $tab_k.(i).(j)$ do
                case 1
                    $W_k.(i) \leftarrow W_k.(i) + w_j$
                case 2
                    $R_k.(i) \leftarrow R_k.(i) + r_j$
        end for
    end for
    Return $W_k$,$R_k$
end procedure

procedure TRAVERSE($l, i, k, tab_k$)
    for $T_j \in$ PRED$(T_l)$ do
        switch $tab_k.(i).(j)$ do
            case 0                          ▷ $\exists i' < i, T_j \in T_{i'}^{\downarrow k}$
                Do nothing
            case 1,2                        ▷ $T_j \in T_i^{\downarrow k}$, already studied
                Do nothing
            case -1                         ▷ $T_j \in T_i^{\downarrow k}$, not yet studied
                for $r = i+1 \ldots n$ do
                    $tab_k.(r).(j) \leftarrow 0$      ▷ $T_j \in T_i^{\downarrow k} \Longrightarrow T_j \notin T_r^{\downarrow k}$
                end for
                if $j < k$ then
                    if $T_j$ ckpted then
                        $tab_k.(i).(j) \leftarrow 2$
                    else
                        $tab_k.(i).(j) \leftarrow 1$
                        $tab_k$ =TRAVERSE $(j, i, k, tab_k)$
                    end if
                else
                    $tab_k.(i).(j) \leftarrow 0$
                end if
    end for
    Return $tab_k$
end procedure
```

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 4.2.

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.

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.

6. Experimental evaluation

In this section, we present experimental results that quantify the performance of the heuristics in Section 5. The source-code (implemented in OCaml) and all input and output data are publicly available at [23].

6.1. Experimental methodology

To evaluate our heuristics with representative DAGs, we use the Pegasus Workflow Generator (PWG) [9], [24]. 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 [24], [25]):

- 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.
- 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.
- 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.
- 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.

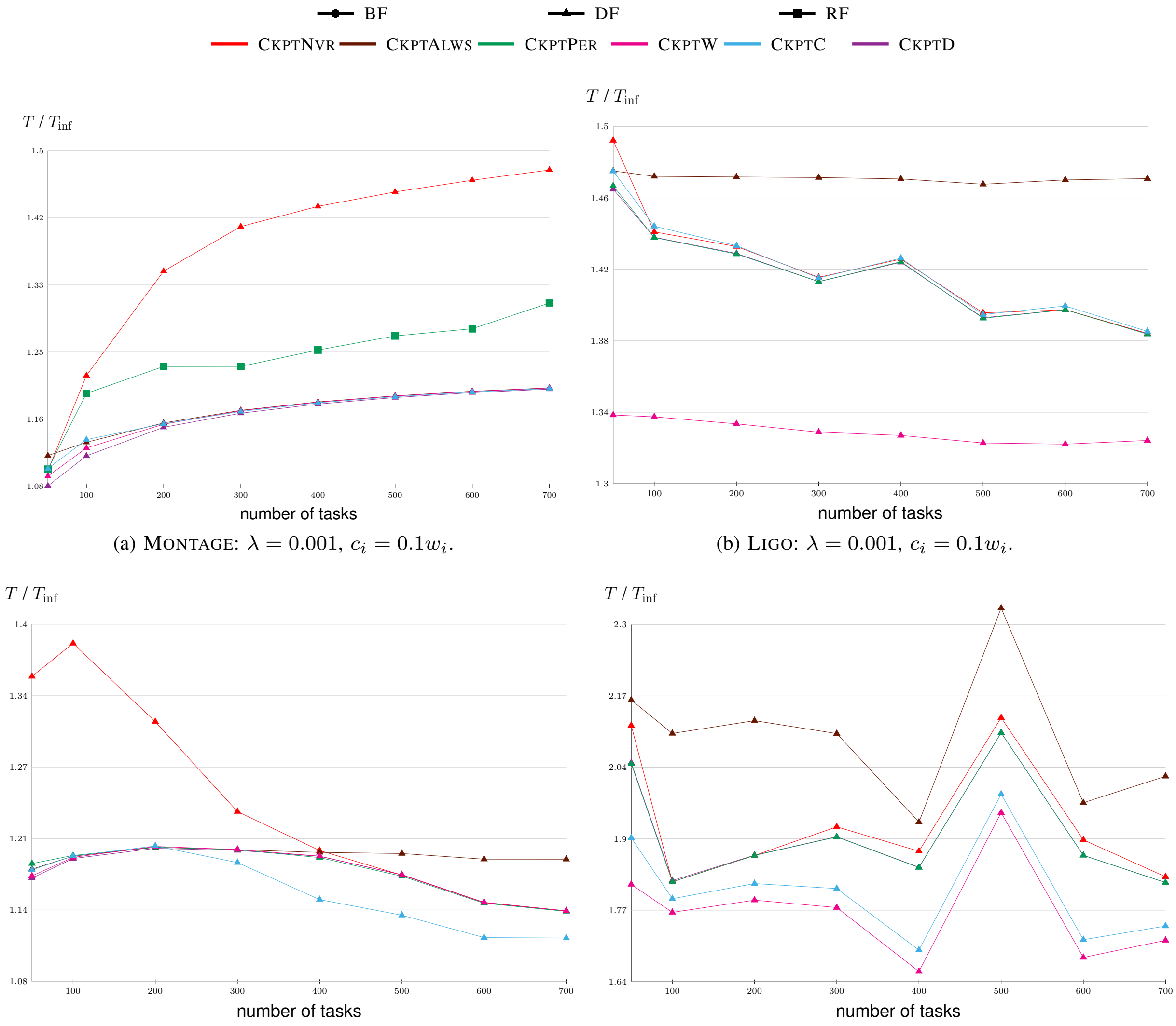
In all experiments, $c_i = r_i$ (checkpoint and recovery costs are identical for a task) and $D = 0$ (downtime is zero seconds). We present results for the different workflows in the particular case where $c_i = w_i/10$, and for a MTBF of 10^3s (except for GENOME where the average weight of each task is significantly longer than for other graphs, in which case we consider a MTBF of 10^4s). These results are very similar to the results that we obtained for MTBF values between 10^2 and 10^7 seconds, and for $c_i = w_i/100$ or $c_i = c$ (constant for all i). See [21] for all results. We vary the number of tasks in each workflow from 50 to 700. All figures in the next section 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 4.2.

6.2. Results

We find that our results strongly depend on the structure of the DAG, meaning that the relative performance of 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 are good across different workflows.

Linearization strategies – Figure 2 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 3 hereafter). Figure 2 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 can be done that depends on the most recently completed work 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 graph and the CKPTPER heuristic (see Figure 3a). We have no explanation but since CKPTPER is the worst checkpointing strategy for that workflow, this result is not particularly relevant. It is interesting to see in Figure 2b that, for the LIGO workflow, RF performs better than BF. This is because RF sometimes corresponds to a DF-like strategy.

Checkpointing strategies – Figure 3 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



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

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

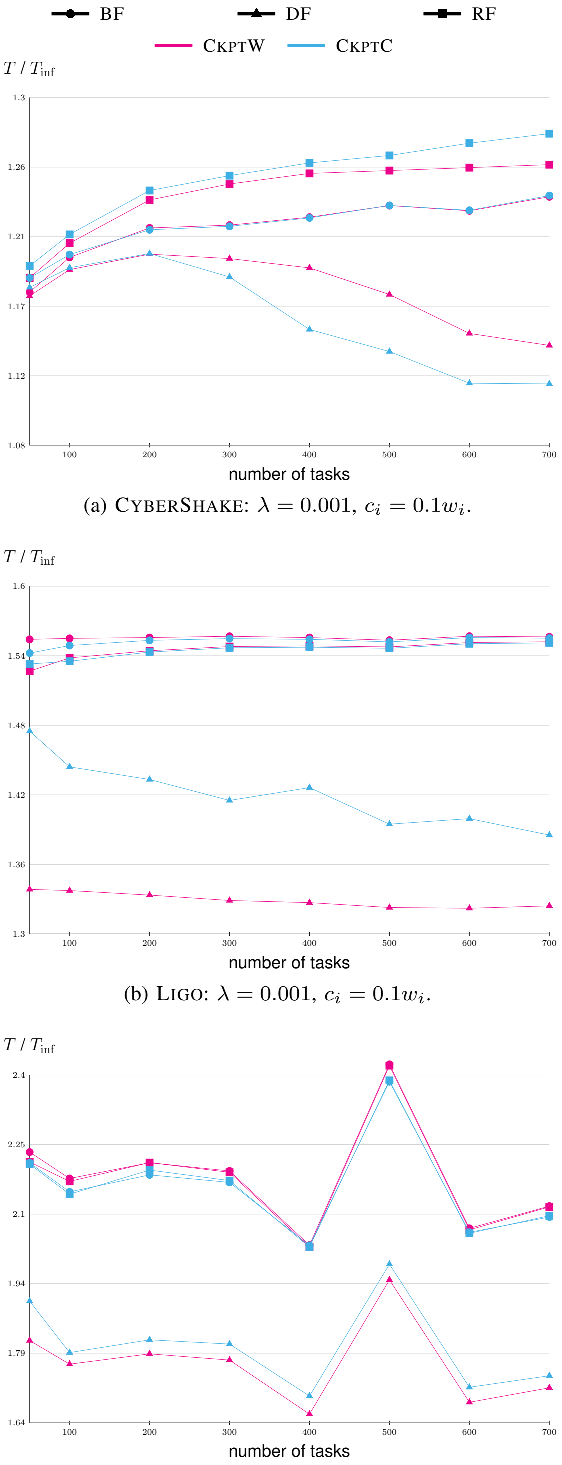
(c) CyberShake: $\lambda = 0.001$, $c_i = 0.1w_i$.

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

Figure 3: Impact of the checkpointing strategy. For each checkpointing strategy, we plot the best linearization strategy.

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 a source 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 3 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 $c_i = w_i/10$. 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).

Constant checkpoint overhead – To better assess the impact of checkpointing costs, we discuss results with a constant checkpoint cost, independent of task weights. First, when CkptW performs better with a proportional checkpoint, it also perform better in this case. Indeed, 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



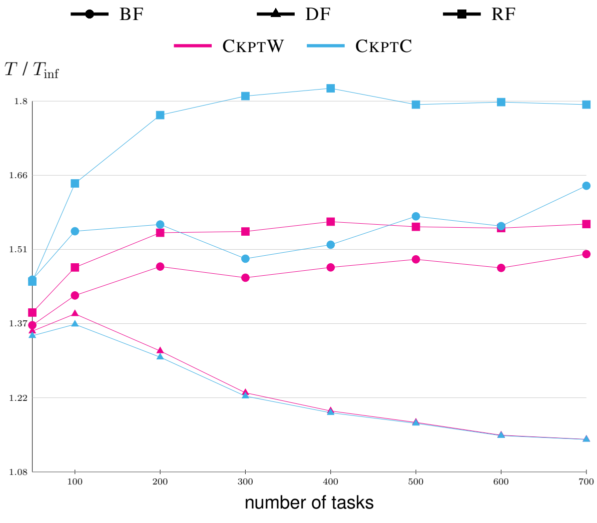
(a) CyberShake: $\lambda = 0.001$, $c_i = 0.1w_i$.

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

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

Figure 2: Impact of the linearization strategy.

is interesting. Figure 4 shows results for CyberShake that allow a comparison of CkptW and CkptC when the checkpointing cost is constant (using $c_i = 10$). This plot can be compared to Figure 2a 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.



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

Figure 4: Impact of the linearization strategy for a constant checkpoint.

Summary – We have compared our heuristics in different 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, and can likely only be discovered empirically as done in this section. Overall our best heuristics, which rely on the computation of the expected makespan given in Section 4.2, lead to significantly better results than the baseline CkptAlws and CkptNvr approaches. Taking into account the structure of the DAG is important, which is highlighted by the poor results of the CkptPer heuristic. Finally, the execution time of all heuristics on a low-end laptop is only a few seconds for graphs with $n = 200$ tasks: this shows that their high complexity $O(n^4)$ does not limit their applicability in practice.

7. 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, and 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 task 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. 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 stochastic 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 different workflows depending of their 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] G. Aupy, A. Benoit, H. Casanova, and Y. Robert, "Scheduling computational workflow on failure-prone platforms," INRIA, Research report RR-8609, Oct. 2014, available at http://graal.ens-lyon.fr/~abenoit.

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

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

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

[25] 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.