Checkpointing Workflows for Fail-Stop Errors

Li Han, Louis-Claude Canon, Henri Casanova, Yves Robert, Frédéric Vivien

Checkpointing Workflows for Fail-Stop Errors

Li Han*†, Louis-Claude Canon*‡, Henri Casanova§, Yves Robert*¶ and Frédéric Vivien*
*LIP, École Normale Supérieure de Lyon, CNRS & Inria, France, email: {li.han|yves.robert|frederic.vivien}@inria.fr
†East China Normal University, China
‡FEMTO-ST, Université de Bourgogne Franche-Comté, France, email: louis-claude.canon@univ-fcomte.fr
§University of Hawai'i at Manoa, USA, email:henric@hawaii.edu
¶University of Tennessee Knoxville, USA

***Abstract*—We consider the problem of orchestrating the execution of workflow applications structured as Directed Acyclic Graphs (DAGs) on parallel computing platforms that are subject to fail-stop failures. The objective is to minimize expected overall execution time, or makespan. A solution to this problem consists of a schedule of the workflow tasks on the available processors and of a decision of which application data to checkpoint to stable storage, so as to mitigate the impact of processor failures. For general DAGs this problem is hopelessly intractable. In fact, given a solution, computing its expected makespan is still a difficult problem. To address this challenge, we consider a restricted class of graphs, Minimal Series-Parallel Graphs (M-SPGs). It turns out that many real-world workflow applications are naturally structured as M-SPGs. For this class of graphs, we propose a recursive list-scheduling algorithm that exploits the M-SPG structure to assign sub-graphs to individual processors, and uses dynamic programming to decide which tasks in these sub-gaphs should be checkpointed. Furthermore, it is possible to efficiently compute the expected makespan for the solution produced by this algorithm, using a first-order approximation of task weights and existing evaluation algorithms for 2-state probabilistic DAGs. We assess the performance of our algorithm for production workflow configurations, comparing it to (i) an approach in which all application data is checkpointed, which corresponds to the standard way in which most production workflows are executed today; and (ii) an approach in which no application data is checkpointed. Our results demonstrate that our algorithm strikes a good compromise between these two approaches, leading to lower checkpointing overhead than the former and to better resilience to failure than the latter.**

I. Introduction

This paper proposes a new algorithm to execute workflows on parallel computing platforms subject to fail-stop processor failures, e.g., a large-scale cluster. The de-facto approach to handle fail-stop failures is Checkpoint/Restart (C/R), by which application state is saved to stable storage, such as a shared file system, throughout execution. Workflows are structured as Directed Acyclic Graphs (DAGs) of tasks. Workflow tasks can be checkpointed individually and asynchronously. Also, rather than checkpointing the entire memory footprint of a task, it is typically only necessary to checkpoint its output data. Therefore, workflows are good candidates for a C/R approach.

The common strategy used in practice is *checkpoint everything*, or CkptAll: the output data of each task is saved onto stable storage (in which case we say "the task is checkpointed"). For instance, in production Workflow Management Systems (WMSs) [1], [2], [3], [4], [5], [6], the default behavior is that all output data is saved to files and all input data is read from files, which is exactly the CkptAll strategy. While this strategy leads to fast restarts in case of failures, its downside is that it maximizes checkpointing overhead. At the other end of the spectrum would be a *checkpoint nothing* strategy, or CkptNone, by which all output data is kept in memory (up to memory capacity constraints) and no task is ever checkpointed, which falls under the "in-situ" workflow executions paradigm [7]. While in a failure-free execution the checkpointing overhead is zero, the downside of this approach is that in case of a failure, a large number of tasks may have to be re-executed. The objective of this work is to achieve a desirable trade-off between these two extremes.

Consider the problem of scheduling a workflow execution and deciding which tasks should checkpoint their output data. The objective is to minimize the expectation of the execution time, or makespan, which is a random variable due to task failures and re-executions. The complexity of this problem is steep. Indeed, consider the CkptAll strategy and assume a given schedule in which each task is assigned to a different processor. Consider now the problem of computing the expected makespan, which amounts to computing the expected longest path in the schedule. Because of failures, task durations are non-deterministic. Computing the expected length of the longest path in a DAG with probabilistic task durations is a known difficult problem [8], [9]. Even in the simplified case in which each task is re-executed at most once, i.e., when task durations are random variables that can take only two discrete values, the problem is #P-complete [8].[1]

In this work, we consider strategies by which some tasks are checkpointed and others are not. When some tasks are not checkpointed, computing the expected makespan becomes more combinatorial due to the complexity of failure recoveries. To understand this intuitively, consider a workflow for which there is a given schedule, i.e., each processor is assigned a sequence of tasks to execute. Furthermore, assume that for each task it has already been decided whether to checkpoint it or not. Consider a non-checkpointed task T_1 assigned to processor P_1 that sends output data to an immediate successor T_2, which is scheduled on another processor, P_2. In this case, we say that T_1 and T_2 have a "crossover dependency". For simplicity, assume that all predecessors of T_1 are checkpointed,

[1]Recall that #P is the class of counting problems that correspond to NP decision problems [10], [11], [12], and that #P-complete problems are at least as hard as NP-complete problems.

meaning that T_1 can always be restarted immediately after a failure of P_1. After a successful execution of T_1, a datum d is sent to P_2, perhaps immediately or delayed until T_2 begins execution. Regardless, d is stored in memory. If P_1 crashes before d has been sent, then T_1 must be re-executed on P_1 (after a reboot) or on a spare processor. If P_2 crashes before T_2 completes, then d must be retrieved from P_1, assuming P_1 has not crashed and has kept d in memory (which may not be the case due to memory space constraints), or T_1 must be re-executed if P_1 has crashed. A series of alternating failures on P_1 and P_2, albeit unlikely, causes many re-executions and data transfers. In general, each processor is scheduled to execute many tasks. Due to the presence of crossover dependencies, a few crashes can thus lead to many task re-executions and data re-transfers, during which other crashes can occur. Computing the expected makespan in this case seems, if anything, more difficult than in the CKPTALL strategy which, as seen above, is already #P-complete. Finally, consider the other extreme strategy, CKPTNONE. To the best of our knowledge, the complexity of computing, or even approximating, the expected makespan for this strategy remained an open problem. In this work, we prove that it is #P-complete.

The above shows that merely computing the expected makespan of a workflow execution in the presence of fail-stop failures, when all scheduling and checkpointing decisions are given, is computationally difficult. Therefore, hoping to compute good scheduling and checkpointing decisions, the effectiveness of which cannot be tractably quantified, seems out of reach. We address this challenge by restricting the problem to Minimal Series Parallel Graphs (M-SPGS) [13]. Despite its name, an M-SPG is essentially an extension of classical Series Parallel Graph (SPG) [14], because source and sink nodes are not merged in series composition (see Section II-A for details). It turns out that most production workflows, e.g., those enabled by production WMSs [1], [2], [3], [4], [5], [6], are M-SPGS. The structure of these graphs makes it possible to orchestrate the execution in fork-join fashion, by which processors compute independent task sets, before joining and exchanging data with other processors. We call these independent task sets *superchains*, because tasks in these sets are linearized into a chain (as they are executed by a single processor) but have forward dependencies that can "skip over" immediate successors. We decide which tasks in a superchain should be checkpointed via a new algorithm, which extends the dynamic programming algorithm of Toueg and Babaoğlu [15] for regular chains. Our solution thus checkpoints fewer tasks than the standard CKPTALL strategy. Furthermore, we always checkpoint the exit tasks of each superchain, which removes all crossover dependencies. As a result, we can tractably compute the expected makespan. More specifically, the contributions of this work are:

- A method to efficiently compute the expected makespan of a checkpointed M-SPG (Section II-B);
- A scheduling/checkpointing strategy CKPTSOME for M-SPGS that improves upon the de-facto standard CKPTALL strategy and avoids all crossover dependencies, and that relies on the two algorithms below (Section II-C);
- A list-scheduling algorithm for scheduling M-SPG workflows as sets of superchains (Section III);
- An algorithm to checkpoint an optimal subset of tasks in a superchain (Section IV);
- The #P-completeness of the problem of computing the expected makespan for the CKPTNONE strategy (Section V);
- Extensive evaluation with real-world Pegasus [1] workflows to evaluate the performance gain afforded by our proposed approach in practice (Section VI).

In addition to the above sections, Section VII reviews relevant related work, and Section VIII provides concluding remarks and highlights directions for future work.

II. PRELIMINARIES AND PROPOSED APPROACH

In this section, we first define M-SPGS. We then review results on how to compute the makespan of a 2-state probabilistic M-SPG, and how to approximate the probability distribution of the execution time of a checkpointed task. Finally, we provide an overview of our proposed approach, including how we compute a schedule and how we determine which tasks should be checkpointed.

A. Minimal Series Parallel Graphs (M-SPG)

We consider computational workflows structured as Minimal Series Parallel Graphs (M-SPGS) [13], which (despite their name) are generalizations of standards SPGS [14]. An M-SPG is a graph $G = (V, E)$, where V is a set of vertices (representing workflow tasks) and E is a set of edges (representing task dependencies). Each task has a *weight*, i.e., its execution time in a no-failure scenario. Each edge between two tasks T_i and T_j is also weighted by the size of the output data produced by T_i that is needed as input to T_j. An M-SPG is defined recursively based on two operators $\overrightarrow{;}$ and $||$ defined as follows:

- The *serial composition* operator $\overrightarrow{;}$ takes two graphs as input and adds dependencies from all sinks of the first graph to all sources of the second graph. Formally, given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, $G_1 \overrightarrow{;} G_2 = (V_1 \cup V_2, E_1 \cup E_2 \cup (sk_1 \times sc_2))$, where sk_1 is the set of sinks of G_1 and sc_2 the set of sources of G_2. This is similar to the serial composition of SPGS, but without merging the sink of the first graph to the source of the second, and extending the construct to multiple sources and sinks.
- The *parallel composition* operator $||$ simply makes the union of two graphs. Formally, given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, $G_1||G_2 = (V_1 \cup V_2, E_1 \cup E_2)$. This is similar to the parallel composition of SPGS, but without merging sources and sinks. Also, we extend the parallel composition to arbitrary numbers of graphs, say $G_1|| \ldots ||G_n$. An M-SPG is then defined recursively as follows:
- A *chain* $g_1 \overrightarrow{;} \ldots \overrightarrow{;} g_n$, where each g_i is an atomic task;
- A serial composition $G_1 \overrightarrow{;} \ldots \overrightarrow{;} G_n$, where each G_i is an M-SPG; or
- A parallel composition $G_1|| \ldots ||G_n$, where each G_i is an M-SPG.

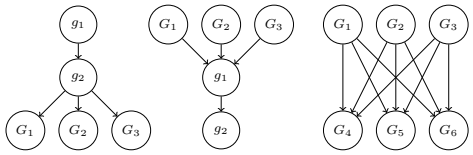

Figure 1: Example M-SPG structures (g_1 and g_2 are atomic tasks whereas G_1 to G_6 are M-SPGs): (a) fork: $(g_1 \overrightarrow{;} g_2) \overrightarrow{;} (G_1||G_2||G_3)$; (b) join: $(G_1||G_2||G_3) \overrightarrow{;} (g_1 \overrightarrow{;} g_2)$; (c) bipartite: $(G_1||G_2||G_3) \overrightarrow{;} (G_4||G_5||G_6)$.

Figure 1 shows example M-SPG structures. Due to the above definition supporting multiple sources and sinks, and not merging sources and sinks, M-SPGs naturally support fork, join (and therefore fork-join), and bipartite structures. It turns out that these structures are common in production workflow applications. For instance, most workflows from the Pegasus benchmark suite [16], [1], which comprises workflows from 20 real-world applications that span various fields of physics, biology, and engineering, are M-SPGs. Overall, M-SPGs exhibit the recursive structure of SPGs (which is key to developing tractable scheduling/checkpointing solutions), but are more general, and as a result maps directly to most production workflow applications. In particular, M-SPGs can model communication patterns that cannot be modeled with SPGs (as the bipartite structure shown in Figure 1.c).

B. First-Order Task Weight Approximation

As discussed in Section I, a key question is the estimation of the expected makespan of a workflow execution for a given schedule and a set of checkpointed tasks. This is because without this estimation, it is not possible to make any claim regarding the effectiveness of scheduling/checkpointing strategies. Computing the expected makespan is #P-complete, even if one considers that the execution time of a task is a discrete random variable that can take only 2 values, i.e., the application is a *2-state probabilistic DAG* [8]. However, basic probability theory tells us how to compute the probability distribution of the sum of two independent random variables (by a convolution) and of the maximum of two independent random variables (by taking the product of their cumulative density functions). As a result, one can compute the makespan distribution and its expected value if the DAG is an SPG, due to its recursive structure [17], [18]. However, the makespan may take an exponential number of values, which makes its direct evaluation inefficient. In fact, the problem of computing the expected makespan remains NP-complete, but in the weak sense, and admits a pseudo-polynomial solution [17]. These results are directly generalizable to M-SPGs.

In this work, we consider failure-prone processors. Consider a single task T, with weight w, scheduled on such a processor. It takes a time r to read the input data of T from stable storage, either for its first execution or after a failure. The total execution time W of T is a random variable, because several execution attempts may be needed before the task succeeds. Let $\lambda \ll 1$ be the exponential failure rate of the processor. With probability $e^{-\lambda(r+w)} = 1 - \lambda(r+w) + \Theta(\lambda^2)$, there i s no failure and W is equal to $r+w$. With probability $(1 - e^{-\lambda(r+w)})e^{-\lambda(r+w)} = \lambda(r+w) + \Theta(\lambda^2)$ a single failure has occurred. For exponentially distributed failures, the expected time to failure knowing that a failure occurs during the task execution (i.e., in the next $r+w$ seconds), is $1/\lambda - (r+w)/(e^{\lambda(r+w)} - 1)$ [19], which converges to $(r+w)/2$ as λ tends to 0. Therefore, when one failure occurs during the first execution of T, and the second execution is successful, W is equal to $\frac{3}{2}(r+w) + \Theta(\lambda)$ (one failure after $(r+w)/2$ seconds in average, a recovery of r seconds, and one successful execution of w seconds). As a first order approximation, we ignore the cases where more than one failure occurs (whose probability is $\Theta(\lambda^2)$), leading to:

$$W = \begin{cases} r+w & \text{with probability } 1-\lambda(r+w)\ , \\ 3/2(r+w) & \text{with probability } \lambda(r+w)\ . \end{cases} \quad (1)$$

Consider now a workflow application with a given schedule and with all tasks checkpointed, so that each task has a known deterministic recovery cost (that of loading from stable storage the output of its predecessors, which are all checkpointed). Then, with the first-order approximation above, computing the expected makespan of the application is the same problem as that of computing the expected makespan of a 2-state probabilistic DAG. We use and compare four existing algorithms to solve this latter problem:

- MONTECARLO – This is the classical Monte Carlo approach [20], [21];
- DODIN (approximation by series-parallel graphs) – See [17], [18] for a detailed description of Dodin's method;
- NORMAL (approximation via a normality assumption) – See [18] for a full description of Sculli's method [22];
- PATHAPPROX (approximation via longest paths) – See [23] for a description of this method.

Again, if each task were checkpointed, we could use these four algorithms to compute the expected makespan. This observation is the key driver for our proposed approach.

C. Proposed Approach

Thanks to the results in Section II-B, given a scheduled M-SPG we can compute the expected makespan for the CKPTALL strategy. However, as outlined in Section I, our objective is to not checkpoint all tasks, so as to save on checkpointing overhead and thus reduce the expected makespan. Our CKPTSOME approach achieves this objective, while retaining the property that the expected makespan can be computed via evaluation algorithms for 2-state probabilistic DAGs.

Consider an M-SPG, G. Without loss of generality, $G = C \overrightarrow{;} (G_1||\ldots||G_n) \overrightarrow{;} G_{n+1}$, where C is a chain and $G_1, \ldots, G_n, G_{n+1}$ are M-SPG graphs, with some of these graphs possibly empty graphs. The schedule for G is the temporal concatenation of the schedule for C, the schedule for $G_1||\ldots||G_n$, and the schedule for G_{n+1}. A chain is always scheduled on a single processor, with all its tasks executed

in sequence on that processor. When scheduling a parallel composition of M-SPGs, we use the following polynomial-time list-scheduling approach, inspired by the "proportional mapping" heuristic [24]. Given an available number of processors, we allocate to each parallel component G_i an integral fraction of the processors in proportion to the sum of the task weights in G_i (communications with stable storage are ignored in this phase). In other terms, we allocate more processors to more costly graphs. We apply this process recursively, each time scheduling a sub-M-SPG on some number of processors. Eventually, each sub-M-SPG is scheduled on a single processor, either because it is a chain or because it is allocated to a single processor. In this case, all atomic tasks in the sub-M-SPG are linearized based on a topological order induced by task dependencies and scheduled sequentially on the processor. This algorithm is described in Section III.

Each time a sub-M-SPG is scheduled on a single processor, we call the set of its atomic tasks a *superchain*, because the tasks are executed sequentially even though the graph may not be a chain. We call the *entry tasks*, resp. *exit tasks*, of a superchain the tasks in the superchain that have predecessors, resp. successors, outside the superchain. Due to the recursive structure of an M-SPG, all predecessors of the entry tasks in a superchain are themselves exit tasks in other superchains. Similarly, all successors of the exit tasks in a superchain are themselves entry tasks in other superchains. This has two important consequences:

- The workflow is an "M-SPG of superchains"; and
- Checkpointing the exit tasks of a superchain means that this superchain never needs to be re-executed. In this case, we say that the superchain is checkpointed.

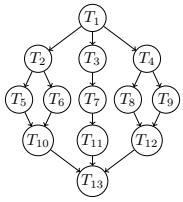

Figure 2: Example M-SPG.

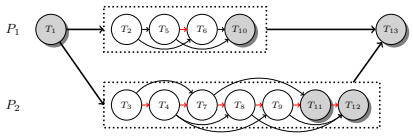

Figure 3: Mapping the M-SPG of Figure 2 onto two processors. The two superchains are shown inside boxes, with all internal and external dependencies from the original graph (red edges result from the linearization). T_{10} is the only exit task of the top superchain while T_{11} and T_{12} are the two exit tasks of the bottom superchain. A checkpoint is performed to save the output of each shadowed task.

A natural strategy is then simply to checkpoint all superchains, which avoids all crossover dependencies (see Section I). In the proposed mechanism, a systematic checkpoint that saves the output files of all exit tasks is done after the execution of the last task of any superchain. This checkpoint strategy is detailed in Section IV-A. Figure 3 shows an example of a schedule obtained on two processors for the M-SPG in Figure 2. A set of tasks is linearized on each processor (additional dependencies are added to enforce a sequential execution). Five checkpoints are taken, after T_1, T_{10}, T_{11}, T_{12}, and T_{13}. This guarantees that once T_{13} starts its execution, any failure on P_2 will have no effect (if P_1 fails, T_{13} will be immediately restarted, otherwise the execution will succeed).

For the makespan evaluation, a naive solution would be to coalesce all the tasks in any superchain into a single checkpointed task, leading to an M-SPG in which all tasks are checkpointed. In the example, the four tasks of the top superchain would be coalesced into one checkpointed task, just as the seven tasks of the bottom superchain. Thanks to the results in Section II-B, one could then compute the expected makespan using the algorithms for 2-state probabilistic DAGs. This naive solution meets our objective, but it may not lead to enough checkpoints. Depending on the parallelism of the M-SPG and the total number of available processors, superchains may contain large numbers of tasks. If only the exit tasks are checkpointed, then the expected execution time of the superchain can be large due to many re-executions from scratch. The solution is to checkpoint other tasks in the superchain in addition to the exit tasks. To this end, we propose a polynomial-time dynamic programming algorithm that determines the optimal set of tasks to checkpoint in each superchain. This algorithm is described in Section IV-B. Once the checkpoints are located, thereby creating task segments ended by a checkpoint, we coalesce each task segment into a single task: again, this is to be able to use the algorithms for 2-state probabilistic DAGs to evaluate the expected makespan.

III. Scheduling M-SPGs

In this section, we describe the list-scheduling algorithm of our CkptSome approach, by which we assign sub-graphs to processors. Consider an M-SPG workflow, G, which comprises sequential atomic tasks, to be executed on a finite set of processors $\mathcal{P}$. Our algorithm decides how many processors should be allocated to each parallel sub-graph. Furthermore, the algorithm is recursive, thus following the recursive M-SPG structure and producing a schedule of superchains, as explained in Section II-C. The pseudo-code is given in Algorithm 1. Procedure Allocate schedules an M-SPG G on a set $\mathcal{P}$ of processors. It does nothing if $G = \emptyset$ (Line 2), otherwise it decomposes G into the sequential composition of a chain, C, a parallel composition, $G_1||\ldots||G_n$, and an M-SPG, G_{n+1} (Line 3). Several such decompositions exist and some of them lead to infinite recursions (when the chain is empty and a single graph is non-empty among $\{G_1, \ldots, G_{n+1}\}$). Our algorithm avoids these decompositions and make sure that C is the longest possible chain. It then schedules the three components in sequence. To do so, it relies on two helper procedures: the OnOneProcessor

procedure, which schedules tasks on a single processor, and the PROPMAP procedure, when more processors are available. ALLOCATE calls ONONEPROCESSOR to schedule C (Line 4) and to schedule $G_1||\ldots||G_n$ if a single processor is available (Line 6). If $|\mathcal{P}| > 1$, then ALLOCATE calls the second helper procedure, PROPMAP (Line 8). This procedure takes in a set of n M-SPGs and a number of processors, p, and returns a list of M-SPGs and a list of processor counts. ALLOCATE then simply recursively schedules the i-th returned M-SPG onto a partition of the platform that contains the i-th processor count (Lines 9-12). Finally, ALLOCATE is called recursively to schedule G_{n+1} (Line 13).

The PROPMAP procedure is the core of our scheduling algorithm. Let $k = \min(n, p)$ be the number of returned M-SPGs and processor counts (Line 16). Initially, the k M-SPGs are set to empty graphs (Line 17), and the k processor counts are set to 1 (Line 18). Array W contains the weight of each returned M-SPGs, initially all zeros (Line 19). Then, input M-SPGs are sorted by non-increasing weight, the weight of an M-SPG being the sum of the weights of all its atomic tasks (Line 20). Two cases are then handled. If $n \geq p$, PROPMAP iteratively merges each G_i with the output M-SPG that has the lowest weight so as to obtain a total of p non-empty output M-SPGs (Lines 22-25). The processor counts remain set to 1 for each output M-SPG. If instead $n < p$, then there is a surplus of processors. PROPMAP first assigns each input G_i to one output M-SPG (Lines 27-29). The $p - n$ extra processors are then allocated iteratively to the output M-SPG with the largest weight (Lines 30-35). Finally, PROPMAP returns the lists of output M-SPGs and of processor counts.

The ONONEPROCESSOR procedure (Lines 38-41) takes as input an M-SPG and a processor, performs a random topological sort of the M-SPG's atomic tasks, and then schedules these tasks in sequence onto the processor.

After assigning all sub-graphs of G onto processors, we complete our CKPTSOME approach by calling the CHECKPOINT procedure to decide which tasks to checkpoint (Lines 43-46), which is described in Section IV.

Algorithm 1 Algorithm CKPTSOME

```
1: procedure ALLOCATE(G, P)
2:     if G = ∅ then return
3:     C →; (G_1||...||G_n) →; G_{n+1} ← G
4:     L ← ONONEPROCESSOR (C, P[0])
5:     if (|P| = 1) then
6:         L ← L ∪ ONONEPROCESSOR (G_1||...||G_n, P[0])
7:     else
8:         (Graphs, Counts) ← PROPMAP (G_1, ..., G_n, |P|)
9:         i ← 0
10:        for each graph, count in Graphs, Counts do
11:            ALLOCATE (graph, {P[i], ..., P[i + count − 1]})
12:            i ← i + count
13:    return L ∪ ALLOCATE (G_{n+1}, P)
14:
15: procedure PROPMAP(G_1, ..., G_n, p)
16:    k ← min(n, p)
17:    Graphs ← [∅, ..., ∅] (k elements)
18:    procNums ← [1, ..., 1] (k elements)
19:    W ← [0, ..., 0] (k elements)
20:    Sort [G_1, ..., G_n] by non-increasing total weight
21:    if n ≥ p then
22:        for i = 1...n do
23:            j ← arg min_{1≤q≤p} (W[q])
24:            W[j] ← W[j] + weight(G_i)
25:            Graphs[j] ← Graphs[j] || G_i
26:    else
27:        for i = 1...n in G_i do
28:            Graphs[i] ← G_i
29:            W[i] ← weight(G_i)
30:        ρ ← p − n
31:        while ρ ≠ 0 do
32:            j ← arg max_{1≤q≤n} (W[q])
33:            procNums[j] ← procNums[j] + 1
34:            W[j] ← W[j] × (1 − 1/procNums[j])
35:            ρ ← ρ − 1
36:    return Graphs, procNums
37:
38: procedure ONONEPROCESSOR(G, proc)
39:    L ← topological_sort(G)
40:    MAP (L, proc)  ▷ Schedule tasks serially on one processor
41:    return {L}
42:
43: procedure CKPTSOME(G, P)
44:    L ← ALLOCATE (G, P)
45:    for L ∈ L do
46:        CHECKPOINT (L)  ▷ Decide which tasks to checkpoint
```

IV. PLACING CHECKPOINTS IN SUPER CHAINS

In this section, we describe our approach for deciding which tasks in a superchain should be checkpointed. We first describe existing results for simple chains and explain how the problem is more difficult in the case of superchains. We then describe an optimal dynamic programming algorithm for superchains.

A. From chains to superchains

Toueg and Babaoğlu [15] have proposed an optimal dynamic programming algorithm to decide which tasks to checkpoint in a linear chain of tasks. For a linear chain, when a failure occurs during the execution of a task T, one has to recover from the latest checkpoint and re-execute all non-checkpointed ancestors of T. In this work, we target M-SPG (sub-)graphs that are linearized on a single processor. As a result, recovery from failure is more complex than in the case of a linear chain. Consider a failure during the execution of a task T. For T to be re-executed, all its input data must be available in memory. Therefore, for each reverse path in the graph from T back to entry tasks of the superchain, one must recover from the latest checkpoint, and then recover by re-executing all non-checkpointed ancestors of T along all reverse paths. Consider the M-SPG in Figure 4(a), and its linearization on a single processor in Figure 4(b). Let us assume that tasks T_2 and T_4 are checkpointed (shadowed in the figures). According to the standard definition, the checkpoint of T_2 includes both its output for T_3 and its output for T_4, while the checkpoint of T_4 includes only its output for T_5.

Let us now consider a single failure that occurs during the execution of T_5. To re-execute T_5, one needs to recover from the checkpointed output of T_4. But one also needs to

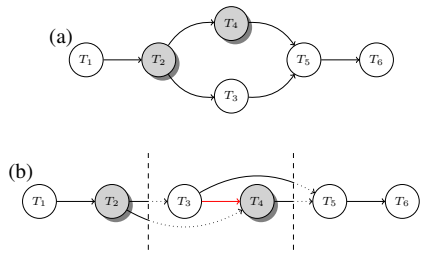

Figure 4: (a) Example of M-SPG. Tasks that are followed by a checkpoint (T_2 and T_4) are shadowed. (b) Linearization of the M-SPG. The dependency from T_3 to T_4, in red, results from the linearization. Vertical dashed lines correspond to checkpoints (after T_2 and T_4). Dotted lines correspond to dependencies from tasks that have been checkpointed.

re-execute T_3, which was not checkpointed, since the output of T_3 is needed for executing T_5. To re-execute T_3, one needs to recover from the checkpoint of T_2. This sequence of recoveries and re-executions must be re-attempted until T_5 executes successfully. As a result, the problem of deciding which tasks to checkpoint to minimize expected makespan cannot be solved by the simple linear chain algorithm in [15].

We thus propose an alternative approach by which a checkpoint, which takes place after the execution of a task, saves not only the output from that task, but also the output of all non-checkpointed tasks with at least one yet-to-be-executed successor. This is shown in Figure 4, where checkpoint times are depicted as vertical dashed lines, after each execution of a checkpointed task (in this case T_2 and T_4). Graphically, "taking a checkpoint" means saving to stable storage all output data of previously executed but un-checkpointed tasks, which corresponds to solid dependence edges that cross the checkpoint time. With this extended definition of checkpoints, the checkpoint of T_4 now includes the output data of T_3 for T_5, in addition to the output of T_4 for T_5.

B. Checkpointing algorithm

To answer the question of when to take checkpoints throughout the execution of a superchain on a processor, we propose an $O(n^2)$ algorithm. Let us consider a superchain that contains tasks $T_a, \ldots, T_b$ (we assume that tasks $T_1, \ldots, T_n$ are numbered according to a topological sort in such a way that tasks from any superchain have contiguous indices). Without loss of generality let us assume that T_j executes immediately before T_{j+1}, $j = a, \ldots, b-1$ and that T_a starts as soon as the necessary input data is read from stable storage. Our approach always takes a checkpoint after T_b completes. This is to avoid crossover dependencies. Recall from Section I that a crossover dependency occurs when a processor failure during the execution of a superchain would require the re-execution of a previously executed superchain. With the checkpointing approach described in the previous section, taking a checkpoint after T_b completes ensures that all output data from all exist tasks of the superchain are checkpointed. As a result, crossover dependencies are prevented. Let $\mathcal{E}Time(j)$ be the optimal expected time to successfully execute tasks $T_a, \ldots, T_j$, when a checkpoint is taken immediately after T_j completes (with possibly earlier checkpoints). Our goal is to minimize $\mathcal{E}Time(b)$.

To compute $\mathcal{E}Time(j)$, we formulate the following dynamic program by trying all possible locations for the last checkpoint before T_j:

$$\mathcal{E}Time(j) = \min\left(T(a,j), \min_{a \le i < j}\{\mathcal{E}Time(i) + T(i+1,j)\}\right),$$

where $T(i+1,j)$ is the expected time to successfully execute tasks T_{i+1} to T_j, provided that a checkpoint occurs after task T_j completes and the previous checkpoint occurred before task T_{i+1} starts. This account for the time to read the input data, execute the tasks and checkpoint them. As there is no checkpoint between tasks T_{i+1} and T_j, all intermediate data are kept in memory and retrieved instantly. This limits the checkpoint overhead. A first-order approximation of the expected time needed to execute tasks T_i to T_j for each (i,j) pair with $i \le j$ is given by

$$\begin{aligned} T(i,j) =& (1 - \lambda(R_i^j + W_i^j + C_i^j)) \times (R_i^j + W_i^j + C_i^j) + \\ & \lambda(R_i^j + W_i^j + C_i^j) \times \frac{3}{2}(R_i^j + W_i^j + C_i^j) \end{aligned} \tag{2}$$

where λ is the processor's exponential failure rate, R_i^j is the time necessary to read from stable storage all data produced by tasks $T_1, \ldots, T_{i-1}$ and needed by tasks $T_i, \ldots, T_j$, $W_i^j = w_i + \ldots + w_j$ is the time to execute tasks T_i to T_j when no failure occurs, and C_i^j is the time taken to checkpoint the input data of $T_{j+1}, \ldots, T_n$ that is produced by $T_i, \ldots, T_j$ (i.e., the non-checkpointed predecessors of $T_{j+1}, \ldots, T_n$ in $T_i, \ldots, T_j$). Formally, $R_i^j = \sum_{k=i}^{j} \sum_{T_l \in \text{Pred}(T_k) \setminus \{T_i, \ldots, T_j\}} c_{lk}$ and $C_i^j = \sum_{k=i}^{j} \sum_{T_l \in \text{Succ}(T_k) \setminus \{T_i, \ldots, T_j\}} c_{kl}$ where c_{kl} is the cost to read or write the data produced by T_k and needed by T_l, $\text{Pred}(T_k)$ is the set of predecessors of T_k and $\text{Succ}(T_k)$ is the set of successors of T_k. Note that the data that is read (during R_i^j) may be produced by exit tasks of previous superchains and that the data that is saved (during C_i^j) may be needed by entry tasks in next superchains. In particular, C_i^j is greater than or equal to the time to checkpoint all output data of T_j.

The first term in Equation (2) corresponds to the "no failure" case (with probability $(1 - \lambda(R_i^j + W_i^j + C_i^j))$ the execution takes time $R_i^j + W_i^j + C_i^j$). The second term corresponds to the "one failure" case (with probability $\lambda(R_i^j + W_i^j + C_i^j)$ there is one failed execution, which on average takes time $\frac{1}{2}(R_i^j + W_i^j + C_i^j)$, followed by a successful execution, which takes time $R_i^j + W_i^j + C_i^j$). As explained in Section II-B, this first-order approximation neglects the λ^2 terms, which correspond to scenarios when multiple failures occur. However, in case of multiple successive failures, $T(i,j)$ is underestimated.

The pseudo-code for this dynamic programming solution is given in Algorithm 2. The computation of $\mathcal{E}Time(j)$ takes $O(n)$ time, as it depends on at most j other entries. The

computation of $T(i,j)$ for all (i,j) pairs with $i \leq j$ takes $O(n^2)$ time. Therefore, the overall complexity is $O(n^2)$.

Algorithm 2 CHECKPOINT

```
1: procedure CHECKPOINT(T_a, ..., T_b)
2:     last_ckpt ← [0, ..., 0] (b − a + 1 elements)
3:     for j = a ... b do
4:         ETime(j) ← T(a, j)
5:         last_ckpt[j] ← 0
6:         for i = a ... j − 1 do
7:             temp ← ETime(i) + T(i + 1, j)
8:             if temp < ETime(j) then
9:                 ETime(j) ← temp
10:                last_ckpt[j] ← i
11:    Ckpts ← ∅                          ▷ List of tasks to checkpoint
12:    while b ≠ a do                                  ▷ Backtracking
13:        Ckpts ← Ckpts ∪ {T_b}          ▷ Checkpoint after task T_b
14:        b ← last_ckpt[b]
15:    return Ckpts
```

We conclude this section with a technical remark. We said a superchain is checkpointed when all its exit tasks are checkpointed. The exact definition should be: a superchain is checkpointed when all the output data of all its exit tasks are saved onto stable storage. Consider the superchain in the example of Figure 3 with two exit tasks T_{11} and T_{12}. Algorithm 2 systematically checkpoints the last task T_{12} but not necessarily T_{11}. However, whenever T_{11} is not checkpointed, the algorithm guarantees that all its output files are saved when checkpointing T_{12}. In addition, the structure of M-SPGs ensures that T_{11} and T_{12} have the same successors outside the superchain, and the recovery is straightforward to implement.

V. THE CKPTNONE STRATEGY

A major contribution of this work is to show the #P-Completeness of the CKPTNONE strategy. Due to lack of space, we refer to the extended version [25] for a precise statement of this result and its proof. Hereafter we simply state a simple formula to evaluate the expected makespan [25]:

Theorem 1. *Consider a schedule for an* M-SPG G *with* p *processors, with all tasks assigned to processors and no checkpoint. Let* W_{par} *be the parallel time of the schedule with no failure, and let* λ *be the processor's exponential failure rate. A formula to estimate the expected makespan* $EM(G)$ *is*

$$EM(G) = (1 - p\lambda W_{par}) \times W_{par} + p\lambda W_{par} \times \left(\frac{3}{2} W_{par}\right)$$

In Section VI, we use $EM(G)$ to evaluate the expected makespan of CKPTNONE. While this formula is likely to be inaccurate, we are not aware of any better approximation.

VI. EXPERIMENTS

In this section, we present experimental results that quantify the effectiveness of the proposed CKPTSOME algorithm.

A. Experimental methodology

Our experiments are for representative workflow applications generated by the Pegasus Workflow Generator (PWG) [26], [27], [16]. 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 (the parameters of which, e.g., total number of tasks, can be chosen). We consider three different classes of workflows generated by PWG, namely MONTAGE, LIGO and GENOME, which are all M-SPGs[2] (information on the corresponding scientific applications is available in [16], [28]). We generate MONTAGE, LIGO, and GENOME workflows with various number of tasks. For each task T_i in the workflow, its weight w_i is generated by PWG. We compute the time required to read or save the data produced by task T_i and needed by task T_j, c_{ij}, by dividing the size of the file in bytes by the stable storage bandwidth in byte/sec. The file sizes are generated by PWG. In some instances, a task may generate the same file for two successors. When this happens, a checkpoint will save the file only once.

In the experiments we consider different exponential processor failure rates. To allow for consistent comparisons of results across different M-SPGs (with different numbers of tasks and different task weights), we simply fix the probability that a task fails, which we denote as p_{fail}, and then simulate the corresponding failure rate. Formally, for a given M-SPG $G = (V, E)$ and a given p_{fail} value, we compute the average task weight as $\bar{w} = \sum_{i \in V} w_i / |V|$, where w_i is the weight of the i-th task in V. We then pick the failure rate λ such that $p_{\text{fail}} = 1 - e^{-\lambda \bar{w}}$. We conduct experiments for three p_{fail} values: 0.01, 0.001, and 0.0001.

An important factor that influences the performance of checkpointing strategies, and more precisely of the checkpointing and recovery overheads, is the time spent computing relative to the time spent performing I/O. The workflows generated by PWG give task durations in seconds and file sizes in bytes. We thus define the Communication-to-Computation Ratio (CCR) as the time needed to store all the files handled by a workflow (input, output, and intermediate files) divided by the time needed to perform all the computations of that workflow on a single processor. The total store time is the total file size divided by the bandwidth to the stable storage. Instead of picking arbitrary bandwidth values, which would have different meanings for different workflows, we vary the CCR by scaling file data sizes by a factor. Decreasing the value of this factor will reduce the file store time (checkpointing cost) and CCR value. This makes it possible to study the performance impact of I/O operations in a coherent manner across experiments and workflow classes and configurations.

[2]Depending on the number of tasks required, PWG may not output an M-SPG Ligo workflow because of some incomplete bipartite graphs. In these cases, to ensure full fairness when comparing approaches, the baseline strategies process the original workflow while CKPTSOME processes a workflow where bipartite graphs have been extended with dummy dependencies carrying empty files (which adds synchronizations but no data transfers).

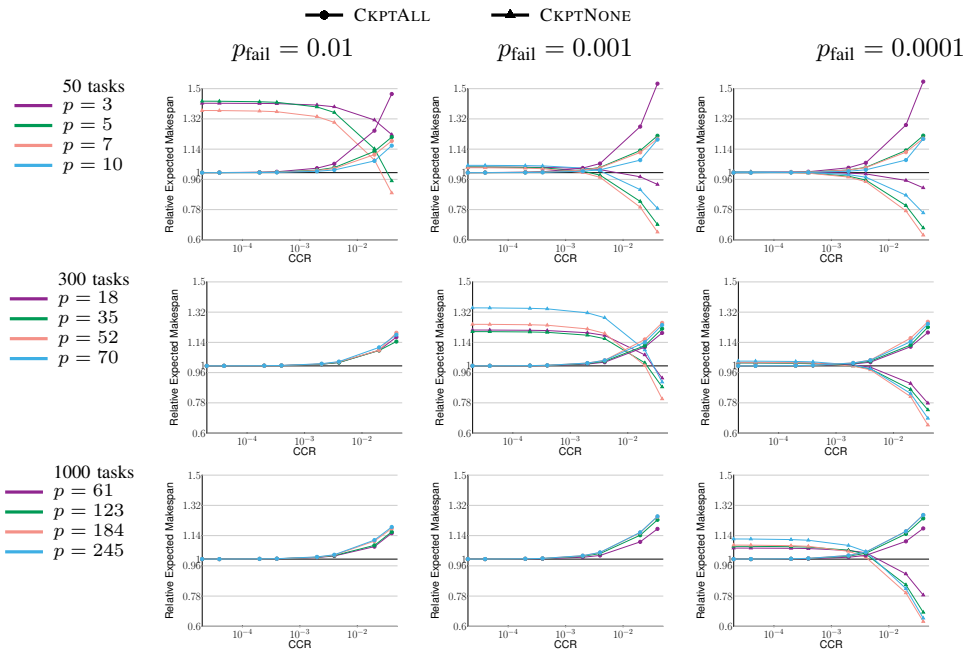

Figure 5: Relative expected makespan of CkptAll and of CkptNone with that of CkptSome for the Genome workflow, three different failure rates, three workflow sizes, and varying Communication-to-Computation Ratio (CCR).

The experiments compare CkptSome to the two extreme approaches, CkptAll and CkptNone. Recall from Section V that we have only an estimate for CkptNone. Expected makespan results are discussed in Section VI-C. But since the expected makespan in those results is computed using approximation algorithms, we first evaluate the accuracy of these algorithms in Section VI-B. The code is publicly available at [29].

B. Accuracy of makespan evaluation

In the extended version [25], we evaluate the accuracy of the evaluation of the expected makespan, using the four methods mentioned in Section II-B, namely MonteCarlo, Dodin, Normal and PathApprox, to compute the expected longest path in 2-state probabilistic DAGs. To this end, we follow the methodology in [23], and also reuse the simulator developed by the authors. For MonteCarlo, we use 300,000 trials and approximate the expected makespan as the average over the 300,000 makespan samples. This huge number of trials is prohibitively expensive in practice, but provides us with an accurate ground truth.

PathApprox had proven both faster and more accurate than Dodin and Normal for dense LU and QR factorization workflows subject to silent errors [23]. We reach the same conclusion for the workflows under study (see [25]), and make PathApprox the method of choice for our experiments.

C. Expected makespan

In this section, we compare the expected makespan of two baseline strategies (CkptAll and CkptNone) over that of our proposed strategy (CkptSome). Figures 5, 6 and 7 show these relative expected makespans vs. the Communication-to-Computation Ratio (CCR). Data points above the $y = 1$ line denote cases in which our strategy leads to better performance than a competitor (i.e., a lower expected makespan). Each figure shows results for workflows with 50, 300, and 1000 tasks, for various numbers of processors P, and for the three p_{fail} values (0.01, 0.001, and 0.0001). More comprehensive results are provided in a companion research report [25].

A clear observation is that CkptSome always outperforms CkptAll[3]. In each scenario, above some CCR value, which depends on the failure rate and the workflow size, CkptSome leads to significant improvement over CkptAll. As the CCR decreases, the relative expected makespan of CkptAll decreases and converges to 1. This is because when checkpointing becomes cheap enough CkptSome decides to checkpoint every task, and thus is equivalent to CkptAll.

Another common trend is that the relative expected makespan of CkptNone increases as the CCR decreases since as checkpoints become cheaper not checkpointing becomes a losing strategy (poorer resilience to failures, but little saving on checkpointing overhead). Overall, CkptNone becomes worse whenever there are more failing tasks, i.e., when the failure rate increases (going from the rightmost column to the leftmost one in the figures), and/or when the number of tasks increases (going from the topmost row to the bottom one in the figures). When the failure rate is high and the workflows are large (the bottom left corner of the figures),

[3]There are in fact a couple of CCR values for Ligo with 300 tasks for which this is not true. This is an artifact of our slight transformation of the Ligo workflow (see Section VI-A for details).

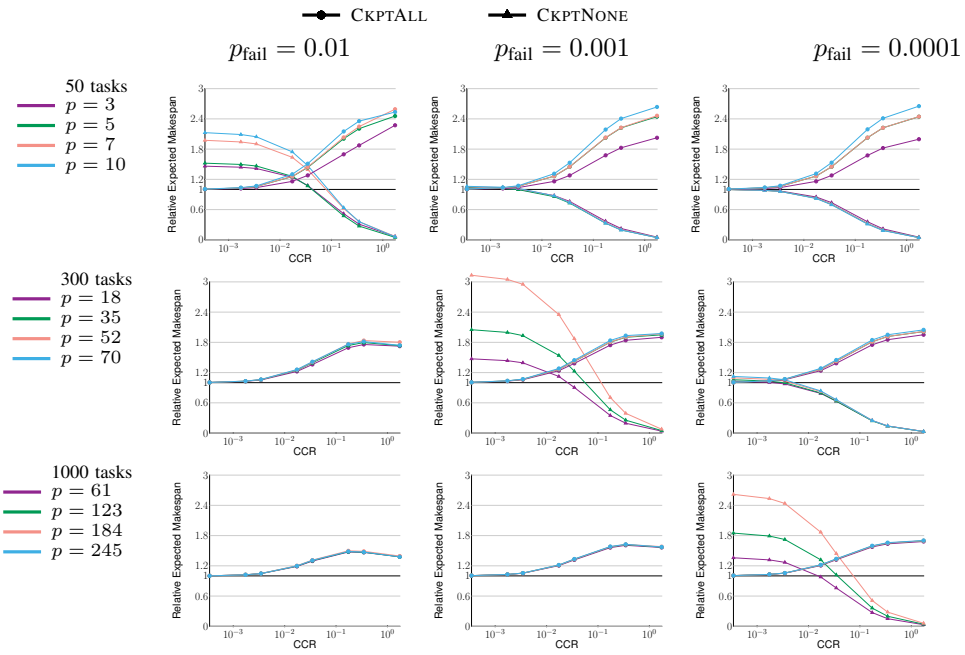

Figure 6: Relative expected makespan of CkptAll and of CkptNone with that of CkptSome for the Montage workflow, three different failure rates, three workflow sizes, and varying Communication-to-Computation Ratio (CCR).

the relative expected makespan of CkptNone is so high that it does not appear in the plots.

CkptSome achieves better results than CkptNone except when (i) checkpoints are expensive (high CCR) and/or (ii) failures are rare (low p_{fail}). In these cases, checkpointing is a losing proposition, and yet CkptSome always checkpoints some tasks (the exit tasks of superchains). In practice, in such cases, the optimal approach is to bet that no failure will happen and to restart the whole workflow execution from scratch upon the very rare occurrence of a failure. The results above for our benchmark workflows, and our experimental methodology in general, make it possible to identify these cases so as to select which approach to use in particular practical situations.

VII. Related work

Checkpointing workflows has received considerable attention in the recent years, but no satisfactory solution is available for fail-stop failures and general DAGs. Many works have been devoted to soft errors, by which a task execution fails but does not lead to completely losing the data present in the processor memory. See the extended version [25] for an overview.

By contrast with soft errors, relatively few published works have studied fail-stop failures in the context of workflow applications. In fact, to the best of our knowledge, existing work only considers linear chains of tasks, or considers workflows that are fully linearized before execution. Consider first a workflow that consists of a linear chain of tasks. The problem of finding the optimal checkpoint strategy, i.e., of determining which tasks to checkpoint, in order to minimize the expected execution time, has been solved by Toueg and Babaoglu [30] using a dynamic programming algorithm. Note that the tasks can themselves be parallel, but the execution flow is sequential, which dramatically limits the amount of re-execution in case of a failure. The algorithm of [30] was later extended in [31] to cope with both fail-stop and silent errors simultaneously.

Consider now a general workflow comprised of parallel tasks that each executes on the whole platform. Therefore, the workflow execution is linearized, and in essence executes as a chain of macro-tasks that execute on a single macro-processor whose speed is the aggregate speed of the available processors and whose failure rate is proportional to the number of available processors. Checkpoints can then be placed after some tasks. However, because the original workflow is not a chain, it is more complicated to keep track of live output data, and the problem of placing checkpoints is NP-complete for simple join graphs [32]. To circumvent this problem, when checkpointing a task, one can decide to checkpoint not only the task's own output data, but also all the live data that will be needed later on in the workflow. This is the main idea of the algorithm proposed in Section IV. To the best of our knowledge, this work is the first approach (beyond application-specific instances) that does not resort to linearizing the entire workflow as a chain of (macro-)tasks. As a result, we propose the first DAG scheduling/checkpointing algorithm that allows independent (sequential) tasks to execute concurrently on multiple processors in standard task-parallel fashion.

VIII. Conclusion

We have proposed a scheduling/checkpointing algorithm, called CkptSome, for executing workflow applications on

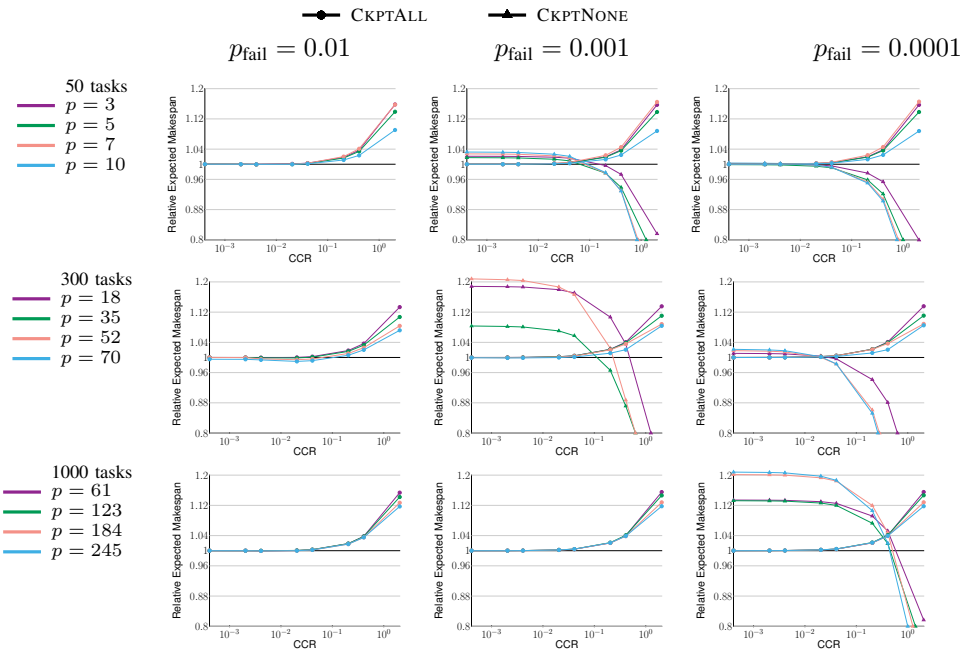

Figure 7: Relative expected makespan of CkptAll and of CkptNone with that of CkptSome for the Ligo workflow, three different failure rates, three workflow sizes, and varying Communication-to-Computation Ratio (CCR).

parallel computing platforms in which processors are subject to fail-stop failures. The objective function to be minimized is the expectation of the makespan, which is a random variable due to non-deterministic task re-executions when failures occur. For general Directed Acyclic Graphs (DAGs), this problem is intractable and even computing the objective function is itself a #P-complete problem. However, by restricting our work to a class of structured recursive DAGs, Minimal Series-Parallel Graphs (M-SPGs), which are broadly relevant to production workflow applications, we are able to design a sensible algorithm and to efficiently compute the expected makespan of the solutions it produces. A competing approach, CkptAll, side-steps part of the difficulty of solving the problem by saving all application data to stable storage so as to minimize the impact of failures, with the drawback of maximizing checkpointing overhead. This is the approach employed by default in most production workflow executions, in which each task is an executable that reads all its input from files and writes all its output to files. Another competing approach, CkptNone, is a risky zero-overhead approach in which the whole workflow is re-executed from scratch in case of a failure. The broad objective of our algorithm is to produce solutions that strike a good compromise between these two extremes. Note that for the CkptNone approach, when applied to general DAGs, we have established that the problem of computing the expected makespan is #P-complete, which to the best of our knowledge is a new result.

We have evaluated the effectiveness of our algorithm by considering realistic workflow configurations produced by a workflow generator from the Pegasus community [26], [27], [16]. We have first demonstrated that the PathApprox method for the expected makespan leads to accurate results, and in particular to results close to those obtained using a brute-force Monte Carlo method, while much faster than Dodin or Normal. Then, we have shown that our CkptSome algorithm does indeed provide an attractive compromise between the CkptAll and CkptNone approaches. More specifically, CkptSome always outperforms CkptAll, is only outperformed by CkptNone when checkpoints are expensive and/or failures are rare. Our experimental methodology provides the quantitative means to identify these cases (based on application CCR, platform scale, and failure rates), so as to select which approach to use in practice.

Future work will be devoted to extending the scheduling algorithms to parallel (moldable) tasks, and to derive graph transformation techniques to enable the approach to arbitrary workflows. A first step would be to deal with General Series Parallel Graphs, which are defined in [13] as graphs whose transitive reduction is an M-SPG.

Another promising direction is to refine the linearization algorithm for superchains (Algorithm 1). Instead of choosing the topological sort arbitrarily, one may try and reduce the total volume of output files, in the hope of reducing the total checkpointing cost when applying Algorithm 2 after the linearization. This problem is related to the sum cut problem [33], which is NP-Complete for general DAGs, but may be amenable to efficient solutions for M-SPGs.

Acknowledgments

This work was supported by the LABEX MILYON (ANR-10-LABX-0070) of Université de Lyon, within the program "Investissements d'Avenir" (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR).

REFERENCES

[1] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, no. 0, pp. 17–35, 2015.

[2] T. Fahringer, R. Prodan, R. Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong *et al.*, "Askalon: A development and grid computing environment for scientific workflows," in *Workflows for e-Science*. Springer, 2007, pp. 450–471.

[3] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633–652, 2011.

[4] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher *et al.*, "The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud," *Nucleic acids research*, p. gkt328, 2013.

[5] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, "Kepler: an extensible system for design and execution of scientific workflows," in *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*. IEEE, 2004, pp. 423–424.

[6] M. Albrecht, P. Donnelly, P. Bui, and D. Thain, "Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids," in *1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*. ACM, 2012, p. 1.

[7] F. Zhang, C. Docan, M. Parashar, S. Klasky, N. Podhorszki, and H. Abbasi, "Enabling In-situ Execution of Coupled Scientific Workflow on Multi-core Platform," in *Proc. of the 26th IEEE International Parallel and Distributed Processing Symposium*, 2012, pp. 1352–1363.

[8] J. N. Hagstrom, "Computational complexity of PERT problems," *Networks*, vol. 18, no. 2, pp. 139–147, 1988.

[9] M. L. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, 5th ed. Springer, 2016.

[10] L. G. Valiant, "The complexity of enumeration and reliability problems," *SIAM J. Comput.*, vol. 8, no. 3, pp. 410–421, 1979.

[11] J. S. Provan and M. O. Ball, "The complexity of counting cuts and of computing the probability that a graph is connected," *SIAM J. Comp.*, vol. 12, no. 4, pp. 777–788, 1983.

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

[13] J. Valdes, R. E. Tarjan, and E. L. Lawler, "The recognition of series parallel digraphs," in *Proc. 11th ACM Symp. Theory of Computing*, ser. STOC '79. ACM, 1979, pp. 1–12.

[14] H. L. Bodlaender and B. de Fluiter, *Parallel algorithms for series parallel graphs*. Springer, 1996, pp. 277–289.

[15] S. Toueg and O. Babaoğlu, "On the optimum checkpoint selection problem," *SIAM J. Comput.*, vol. 13, no. 3, 1984.

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

[17] R. H. Möhring, "Scheduling under uncertainty: Bounding the makespan distribution," in *Computational Discrete Mathematics: Advanced Lectures*, H. Alt, Ed. Springer, 2001, pp. 79–97.

[18] L. C. Canon and E. Jeannot, "Correlation-aware heuristics for evaluating the distribution of the longest path length of a DAG with random weights," *IEEE Trans. Parallel Distributed Systems*, 2016, available at http://doi.ieeecomputersociety.org/10.1109/TPDS.2016.2528983.

[19] T. Hérault and Y. Robert, Eds., *Fault-Tolerance Techniques for High-Performance Computing*, ser. Computer Communications and Networks. Springer Verlag, 2015.

[20] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[21] R. M. van Slyke, "Monte carlo methods and the pert problem," *Operations Research*, vol. 11, no. 5, pp. 839–860, 1963.

[22] D. Sculli, "The completion time of PERT networks," *The Journal of the Operational Research Society*, vol. 34, no. 2, pp. 155–158, 1983.

[23] H. Casanova, J. Herrmann, and Y. Robert, "Computing the expected makespan of task graphs in the presence of silent errors," in *P2S2'2016, the 9th Int. Workshop on Programming Models and Systems Software for High-End Computing*. IEEE Computer Society Press, 2016.

[24] A. Pothen and C. Sun, "A mapping algorithm for parallel sparse cholesky factorization," *SIAM Journal on Scientific Computing*, vol. 14, no. 5, pp. 1253–1257, 1993.

[25] L. Han, L.-C. Canon, H. Casanova, Y. Robert, and F. Vivien, "Checkpointing workflows for fail-stop errors," INRIA, Research Report 9068, May 2017.

[26] R. F. da Silva, W. Chen, G. Juve, K. Vahi, and E. Deelman, "Community resources for enabling research in distributed scientific workflows," in *e-Science (e-Science), 2014 IEEE 10th International Conference on*, vol. 1. IEEE, 2014, pp. 177–184.

[27] 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)*. IEEE, 2008, pp. 1–10.

[28] 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.

[29] L. Han, "Checkpointing Workflows for Fail-Stop Errors: Simulation Code," https://doi.org/10.6084/m9.figshare.5057650.v3, 2017.

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

[31] A. Benoit, A. Cavelan, Y. Robert, and H. Sun, "Assessing general-purpose algorithms to cope with fail-stop and silent errors," *ACM Trans. Parallel Computing*, vol. 3, no. 2, 2016.

[32] G. Aupy, A. Benoit, H. Casanova, and Y. Robert, "Scheduling computational workflows on failure-prone platforms," *Int. J. of Networking and Computing*, vol. 6, no. 1, pp. 2–26, 2016.

[33] J. Díaz, J. Petit, and M. Serna, "A survey of graph layout problems," *ACM Computing Surveys (CSUR)*, vol. 34, no. 3, pp. 313–356, 2002.