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}@ens-lyon.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, Generalized Series-Parallel Graphs (G-SPGs). It turns out that many real-world workflow applications are naturally structured as G-SPGs. For this class of graphs, we propose a recursive list-scheduling algorithm that exploits the G-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 compute a first-order approximation of the expected makespan for the solution produced by this algorithm. We evaluate 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 extremes, leading to lower checkpointing overhead than the former and to better resilience to failure than the latter. To the best of our knowledge, this is the first scheduling/checkpointing algorithm for workflow applications with fail-stop failures that considers workflow structures more general than mere linear chains of tasks.

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, and 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 execution 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, leading to slow restarts. The objective of this work is to achieve a desirable compromise between CkptAll and CkptNone.

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 probabilistic, and 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

[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.

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 assigned to processor P that sends output data to an immediate successor T', which is scheduled on another processor, P'. In this case we say that T and T' have a "crossover dependency". For simplicity, assume that all predecessors of T are checkpointed, meaning that T can always be restarted immediately after a failure of P. After a successful execution of T, a datum d is sent to P', perhaps immediately or delayed until T' begins execution. Regardless, d is stored in memory. If P crashes before d has been sent, then T must be re-executed on P (after a reboot) or on a spare processor. If P' crashes before T' completes, then d must be retrieved from P, assuming P has not crashed and has kept d in memory (which may not be the case due to memory space constraints), or T must be re-executed if P has crashed. A series of alternating failures on P and P', 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 Generalized Series Parallel Graphs (G-SPGS). A G-SPG is essentially a Series Parallel Graph (SPG), but with a slightly modified definition so as to avoid the proliferation of dummy nodes when converting a general DAG into an SPG. It turns out that most production workflows, e.g., those enabled by production WMSs [1], [2], [3], [4], [5], [6], are G-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 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 [13] 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 a first-order approximation of the expected makespan. More specifically, the contributions of this work are:

- The introduction of G-SPGS, a variant of classical SPGS that is better suited to modeling real-world workflows (Section II-A);
- A method to approximate the expected makespan of a checkpointed G-SPG (Section II-B);
- A scheduling/checkpointing strategy CKPTSOME for G-SPGS that improves upon the de-facto standard CKPTALL strategy and avoids all crossover dependencies, relying on the two algorithms below (Section II-C);
- A list-scheduling algorithm for scheduling G-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);
- Experimental evaluation with real-world Pegasus [1] workflows to quantify 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.

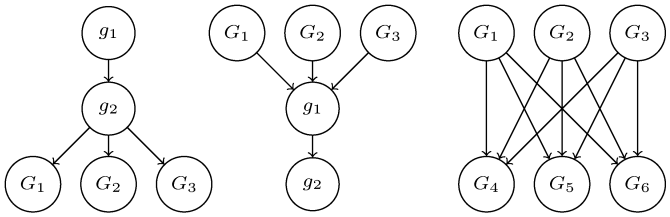
II. PRELIMINARIES AND PROPOSED APPROACH

In this section we define G-SPGS, the class of workflow DAGs that we consider in this work. We then review results on how to compute the makespan of a 2-state probabilistic G-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. General Series Parallel Graphs (G-SPG)

We consider computational workflows structured as Generalized Series Parallel Graphs (G-SPGS), which are generalizations of standards SPGS (see [14] for a definition of series-parallel graphs). A G-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. A G-SPG is defined recursively based on two operators, $\vec{;}$ and $||$, defined as follows:

- The *serial composition* operator, $\vec{;}$, 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 $G_2 = (V_2, E_2)$, $G_1 \vec{;} G_2 = (V_1 \cup V_2, E_1 \cup E_2 \cup (sk_1 \times sc_2))$, 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



(a) (b) (c)

Figure 1: Example G-SPG structures: (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: $(G_1||G_2||G_3) \overrightarrow{;} (G_4||G_5||G_6)$.

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 $G_1||\ldots||G_n$. This is in contrast to the binary parallel composition of SPGs, by which composing n parallel SPGs requires adding $\Theta(n)$ dummy vertices.

Given the above operators, a G-SPG is then defined recursively as follows:

- A *chain*, i.e., $g_1 \overrightarrow{;} \ldots \overrightarrow{;} g_n$, where each g_i is an atomic task;
- A serial composition, i.e., $G_1 \overrightarrow{;} \ldots \overrightarrow{;} G_n$, where each G_i is a G-SPG; or
- A parallel composition, i.e., $G_1||\ldots||G_n$, where each G_i is a G-SPG.

Figure 1 shows example G-SPG structures. Due to the above definition supporting multiple sources and sinks, and not merging sources and sinks, G-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 [15], [1], which comprises workflows from 20 real-world applications that span various fields of physics, biology, and engineering, are G-SPGs. Overall, G-SPGs exhibit the recursive structure of SPGs (which is key to developing tractable scheduling/checkpointing solutions), but are more convenient since they allow vertices with arbitrary arity (which avoids the proliferation of dummy nodes, which simplifies both modeling and solving the scheduling/checkpointing problem), and as a result maps directly to most production workflow applications. Moreover, G-SPGs can also model communication patterns that cannot be modeled with SPGs (this is the case of the bipartite structure shown in Figure 1.(c) and are thus more general.

B. First-Order Makespan Approximation for 2-State Probabilistic G-SPGs

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 [16], [17]. 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 [16]. These results are directly generalizable to G-SPGs.

In this work we consider failure-prone processors. Consider a single task T, with weight w, scheduled on such a processor. Consider that this task has a recovery cost of r in case of a failure. T's execution time, W, 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 $e^{-\lambda w} = 1 - \lambda w + \Theta(\lambda^2)$ (no failure), W is equal to w. With probability $(1-e^{-\lambda w})e^{-\lambda w} = \lambda w + \Theta(\lambda^2)$ 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 w seconds), is $1/\lambda - w/(e^{\lambda w} - 1)$ [18], which converges to $w/2$ 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 $r + \frac{3}{2}w + \Theta(\lambda)$ (one failure after $w/2$ seconds, a recovery that takes r seconds, and one successful execution that takes w seconds). As a first order approximation, we can ignore the cases in which more than one failure occurs (which occurs with probability $\Theta(\lambda^2)$) leading to:

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

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 compare four well-known algorithms to solve this latter problem in Section VI-B. Therefore, if all tasks are checkpointed, we can compute a first-order approximation of the overall expected makespan. This observation is the key driver for our proposed approach, which is outlined in the next section.

C. Proposed Approach

Thanks to the results in the previous section, given a scheduled G-SPG we can compute a first-order approximation of 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. The approach proposed in this work achieves this objective, while retaining the property that a first-order approximation of the expected makespan can be computed.

Consider a G-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 G-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 G-SPGs we use the following polynomial time list-scheduling approach, inspired by the "proportional mapping" heuristic [19]. 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. In other terms, we allocate more processors to more costly graphs. We apply this process recursively, each time scheduling a sub-G-SPG on some number of processors. Eventually, each sub-G-SPG is scheduled on a single processor, either because it is a chain or because it is allocated a single processor. In this case, all atomic tasks in the G-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-G-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 a G-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 a "G-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.

A natural strategy is then simply to checkpoint all superchains, which avoids all crossover dependencies (see Section I). More precisely, given a superchain, a systematic checkpoint is taken after the execution of the last task of that superchain. This checkpoint saves the output files of all the exit tasks in the superchainr. This strategy is detailed in Section IV-A. Figure 3 shows an example of a schedule obtained on two processors for the G-SPG in Figure 2. A set of tasks is linearized on each processor (additional dependencies are added to enforce a sequential execution). Four checkpoints are taken: after the executions of T_1, T_{10}, T_{12}, and T_{13}. The checkpoint after T_{12} also saves the output of T_{11}, because T_{11} is an exit task. 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).

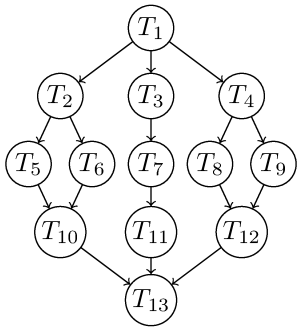


Figure 2: Example G-SPG.

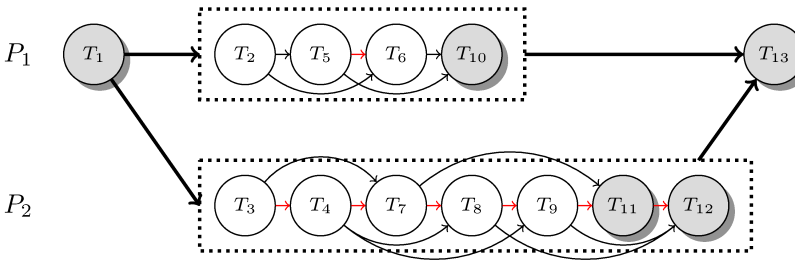


Figure 3: Mapping the G-SPG of Figure 2 onto two processors. The two superchains are shown inside boxes, with all internal and external dependencies from the original graph (red edges result from the linearization). T_{10} is the only exit task of the top superchain while T_{11} and T_{12} are the two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

For the makespan evaluation, a naive solution would be to coalesce all the tasks in any superchain into a single checkpointed task, leading to a G-SPG in which all tasks are checkpointed. In the example, the four tasks of the top superchain would be coalesced into one checkpointed task, and so would the seven tasks of the bottom superchain. Thanks to the results in the previous section, 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 G-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 detail in Section IV-B. Once the checkpoints are decided, thereby creating task segments ended by a checkpoint, we coalesce each task segment into a single task. Again, this is so that we can reuse algorithms for computing the expected makespan of 2-state probabilistic DAGs.

III. Scheduling G-SPGs

In this section, we describe the list-scheduling algorithm by which we assign sub-graphs to processors. Consider a G-SPG workflow, G, which comprises sequential atomic tasks, to be executed on a finite set of processors P. Our algorithm decides how many processors should be allocated to parallel sub-graphs. Furthermore, the algorithm is recursive, thus following the recursive G-SPG structure and producing a schedule of superchains, as explained in Section II-C.

The pseudo-code of our approach is given in Algorithm 1. Procedure ALLOCATE schedules a G-SPG G on a set 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, $G_1||\dots||G_n$, and a G-SPG, G_{n+1} (Line 3). To ensure termination, either C is non-empty, or two G-SPGs are non-empty among $\{G_1, \dots, G_{n+1}\}$. It then schedules these three components in sequence. To do so it relies on two helper procedures.

The ONONEPROCESSOR procedure (Lines 39-43) takes as input a G-SPG and a processor, performs a random topological sort of the G-SPG's atomic tasks, and then schedules these tasks in sequence onto the processor. (It also decides which tasks to checkpoint by calling the CHECKPOINT procedure, which is described in Section IV). ALLOCATE calls ONONEPROCESSOR to schedule C (Line 4) and to schedule $G_1||\dots||G_n$ if a single processor is available (Line 6). If $|P| > 1$, then ALLOCATE calls the second helper procedure, PROPMAP (Line 8). This procedure takes in a set of n G-SPGs and a number of processors, p, and returns a list of G-SPGs and a list of processor counts. ALLOCATE then simply recursively schedules the i-th returned G-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 $k = \min(p, n)$ be the number of returned G-SPGs and processor counts (Line 17). Initially, the k G-SPGs are set to empty graphs (Line 18), and the k processor counts are set to 1 (Line 19). Array W contains the weight of each returned G-SPGs, initially all zeros (Line 20). Then, input G-SPGs are sorted by non-increasing weight, the weight of a G-SPG being the sum of the weights of all its atomic tasks (Line 21). Two cases are then handled. If $n \geq p$, PROPMAP iteratively merges each G_i with the output G-SPG that has the lowest weight so as to obtain a total of p non-empty output G-SPGs (Lines 23-26). The processor counts remain set to 1 for each output G-SPG. If instead $n < p$, then there is a surplus of processors. PROPMAP first assigns each input G_i to one output G-SPG (Lines 28-30). The $p - n$ extra processors are then allocated iteratively to the output G-SPG with the largest weight (Lines 31-36). Finally, PROPMAP returns the lists of output G-SPGs and of processor counts.

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

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 [13] 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

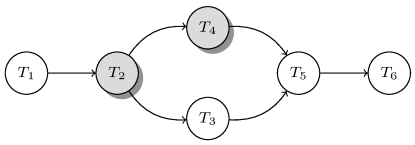


Figure 4: Example G-SPG. Tasks that are followed by a checkpoint (T_1 and T_3) are shadowed.

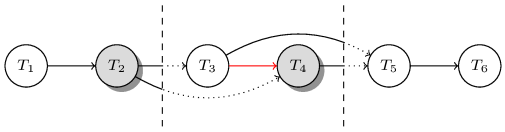


Figure 5: Linearization of the example G-SPG in Figure 4. Vertical dashed lines correspond to checkpoints (after T_1 and T_3). Dotted lines correspond to dependencies from tasks that have been checkpointed. The dependency from T_2 to T_3, in red, results from the linearization.

to recover from the latest checkpoint and re-execute all non-checkpointed ancestors of T.

In this work we target G-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 find the latest checkpoint. One must then recover by re-executing all non-checkpointed ancestors of T along all reverse paths. As an example consider the G-SPG in Figure 4, and its linearization on a single processor in Figure 5. Let us assume that tasks T_1 and T_3 are checkpointed (shadowed in the figures). According to the standard definition of checkpoints, the checkpoint of T_1 includes both its output for T_2 and its output for T_3, while the checkpoint of T_3 includes only its output for T_4.

HC: The task numbers in the 2 paragraphs below are very wrong... was the intent to change the figure(s)? Or should just fix them?

Let us now consider a single failure that occurs during the execution of T_4. To re-execute T_4, one needs to recover from the checkpointed output of T_3. But one also needs to re-execute T_2, which was not checkpointed, since the output of T_2 is needed for executing T_4. To re-execute T_2, one needs to recover from the checkpoint of T_1. This sequence of recoveries and re-executions must be re-attempted until T_4 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 [13].

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 on the example in Figure 5, where checkpoint times are depicted as vertical dashed lines, after each execution of a checkpointed task (in this case T_1 and T_3). 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_3 now includes the output data of T_2 for T_4, in addition to the output of T_3 for T_4.

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 $T_1, \ldots, T_n$, and without loss of generality let us assume that T_j executes immediately before T_{j+1}, $j = 1, \ldots, n-1$. Our approach always takes a checkpoint after T_n 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_n completes ensures that all output data from all exit 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 $T_1, \ldots, T_j$, when a checkpoint is taken immediately after T_j completes (with possibly earlier checkpoints). Our goal is to minimize $\mathcal{E}Time(n)$.

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

$$\mathcal{E}Time(j) = \min_{0 \le i < j} \left\{ \mathcal{E}Time(i) + T(i+1, j) + C_j^{(i)} \right\},$$

where $T(i+1, j)$ is the expected time to successfully execute tasks T_{i+1} to T_j, provided that T_i and T_j are both checkpointed, while no other task in between is checkpointed; and $C_j^{(i)}$ is the time taken to perform the checkpoint after task T_j completes, given that the previous checkpoint occurred after the completion of task T_i. Note that $C_j^{(i)}$ is the time to checkpoint the input data of $T_{j+1}, \ldots, T_n$, which is produced by $T_{i+1}, \ldots, T_j$ (i.e., the non-checkpointed predecessors of $T_{j+1}, \ldots, T_n$ in $T_{i+1}, \ldots, T_j$). Thus, $C_j^{(i)} \ge c_j$, where c_j is the time to checkpoint all output data of T_j.

To initialize the dynamic program, we define $\mathcal{E}Time(0) = 0$. 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

$$T(i,j) = (1 - \lambda W_i^j) \times W_i^j + \lambda W_i^j \times \left(\frac{3}{2} W_i^j + R_i^j \right) \quad (2)$$

where λ is the processor's exponential failure rate, $W_i^j = w_i + \ldots + w_j$ is the time to execute tasks T_i to T_j when no failures occur, and R_i^j is the (recovery) time necessary to read

from stable storage all data produced by tasks $T_1, \ldots, T_{i-1}$ and needed by tasks $T_i, \ldots, T_j$ for $i \geq 2$ and $R_1^j = 0$ for $1 \leq j \leq n$. The first term above corresponds to the "no failure" case (with probability $(1 - \lambda W_i^j)$ the execution takes time W_i^j). The second term corresponds to the "one failure" case (with probability λW_i^j there is one failed execution, which on average takes time $\frac{1}{2} W_i^j$, followed by a successful execution, which takes time W_i^j). As explained in Section II-B we neglect the λ^2 terms. The pseudo-code for this dynamic programming solution is given in Algorithm 2.

Algorithm 2 CHECKPOINT

```
1: procedure CHECKPOINT(T_1, ..., T_n)
2:     last_ckpt ← [0, ..., 0] (n elements)
3:     ETime(0) ← 0
4:     for j = 1 ... n do
5:         ETime(j) ← T(1, j) + C_j^(0)
6:         last_ckpt[j] ← 0
7:         for i = 1 ... j − 1 do
8:             temp ← ETime(i) + T(i + 1, j) + C_j^(i)
9:             if temp < ETime(j) then
10:                ETime(j) ← temp
11:                last_ckpt[j] ← i
12:    Ckpts ← ∅                          ▷ List of tasks to checkpoint
13:    while n ≠ 0 do                     ▷ Backtracking
14:        Ckpts ← Ckpts ∪ {T_n}          ▷ Checkpoint after task T_n
15:        n ← last_ckpt[n]
16:    return Ckpts
```

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 $O(n^2)$.

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, if T_{11} turns out not checkpointed, the algorithm guarantees that all its output files are saved when checkpointing T_{12}. In addition, the structure of G-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

In this section we establish the complexity of computing the expected makespan of a scheduled task graph when the CKPTNONE strategy is used. In Section V-A we construct a simple instance and show that it is already #P-Complete, thereby establishing the #P-completeness of the problem. Then in Section V-B we derive a simple formula to compute an approximation of the expected makespan.

A. #P-completeness

Let us first define the problem:

Definition 1 (DAG-MKS)**.** Consider a task graph with n tasks. Each task T_i is scheduled on its own processor P_i and has a unitary cost. Each task can thus start executing as soon as all its predecessors have completed (there are no resource constraints). There is a fixed probability, p_i, that processor P_i fails when it executes its allocated task T_i for $1 \leq i \leq n$. Once P_i has failed, it restarts at the next time-step and it cannot fail again. Hence, if P_i fails while executing T_i, it will successfully re-execute T_i during the next time-step. The problem is to compute the expected makespan of the schedule.

In this simplified problem, we have discrete times-steps, and failures hit processors only once, similarly to the approximated execution model given in Equation (1). Note that with this simple model, the makespan is bounded. Also, since tasks are unitary, there is no distinction between transient and fail-stop failures.

Theorem 1. DAG-MKS *is #P-complete.*

Proof. We show this result with a reduction from REL [8], a #P-complete problem. Consider a DAG with a source vertex, and let V_i be the set of vertices with a path of length $i-1$ from the source. In what follows, we consider layered graphs, and V_i is thus the set of nodes on layer i. A transportation DAG is a graph in which edges go only from the source $v_1 \in V_1$ to vertices in V_2, from vertices in V_2 to vertices in V_3 and from vertices in V_3 to the sink $v_n \in V_4$. In other words, this is a four-layer graph shaped as a directed bipartite graph with a source and a sink (see Figure 6).

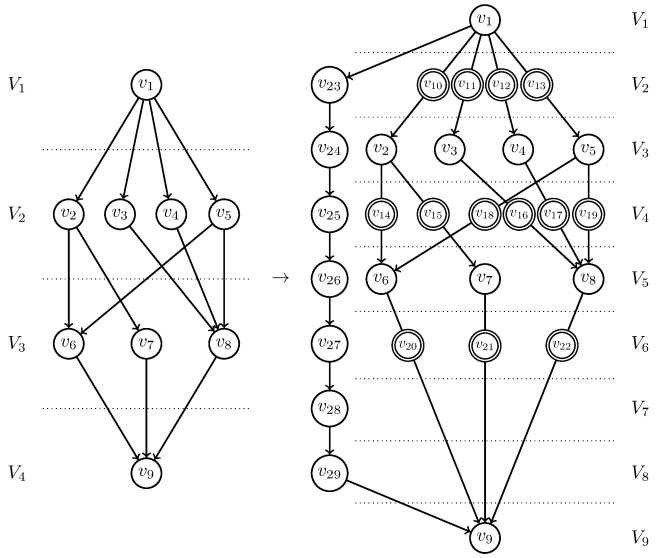


Figure 6: The transportation graph of the REL instance (left) and the corresponding DAG-MKS instance (right). In the REL instance, each edge may fail with probability p. In the DAG-MKS instance, tasks with a double circle (v_{10} to v_{22}) may fail with probability $1 - p$, while other tasks never fail.

Definition 2 (REL)**.** We consider a transportation DAG where each edge may fail with probability p. The objective is to

determine the probability that there is path between the source and the sink.

We first transform an instance of REL into an instance of a related problem in which the vertices fail instead of the edges. Each initial vertex remains unchanged and cannot fail. We replace each edge by a vertex that can fail with probability p and connect this vertex to the predecessor and the successor of the edge. This leads to a transformed graph with 7 layers of vertices. Vertices in even layers fail with probability p, whereas vertices in odd layers do not fail. The probability that there is a path between the source and the sink is the same as with the initial REL instance.

We now build an instance of DAG-MKS with the same graph structure, and we let $p_i = 1 - p$ for all vertices of even layers and $p_i = 0$ otherwise. We will prove that determining the probability that the makespan of this DAG is equal to 10 solves the REL instance.

We introduce some notations for the REL instance. Let E_{ij} be the event that occurs when the edge from vertex v_i to v_j succeeds ($\Pr[E_{ij}] = 1 - p$). All E_{ij} are independent. Let F_i^j be the event that occurs when there is a path from the source to a vertex $v_i \in V_j$ in the REL instance. Then, F_1 always occurs, $F_i = E_{1i}$ for $v_i \in V_2$, $F_i = \bigcup_{j \in \text{Pred}(v_i)} F_j \cap E_{ji}$ for $v_i \in V_3$ and $F_n = \bigcup_{j \in \text{Pred}(v_n)} F_j \cap E_{jn} = \bigcup_{j \in \text{Pred}(v_n)} \bigcup_{k \in \text{Pred}(v_j)} E_{1k} \cap E_{kj} \cap E_{jn}$. Solving REL consists in determining $\Pr[F_n]$.

We now focus on the DAG-MKS instance. Let G_i be the event that occurs when vertex v_i in layer V_j fails at step j and is re-executed, for $j \in \{2, 4, 6\}$ (recall that vertices in odd layers never fail). We have $\Pr[G_i] = 1 - p$, which is equivalent to the event $E_{\text{pred}(v_i)\text{succ}(v_i)}$. All G_i are independent. Let C_i be the completion time of vertex v_i. Consider the first three layers. The event $\{C_1 = 1\}$ always occurs, because the source vertex never fails. For $v_i \in V_2$, either no fault occurs ($\overline{G_i}$) and $C_i = 2$, or a fault occurs and it takes one more time-step to execute task v_i, i.e., we derive that $G_i = \{C_i = 3\}$. Finally, $\{C_i = 4\} = \{C_{\text{pred}(v_i)} = 3\} = G_{\text{pred}(v_i)}$ for $v_i \in V_3$. Analogously, for the two next layers, we have: $\{C_i = 6\} = \{C_{\text{pred}(v_i)} = 4\} \cap G_i$ for $v_i \in V_4$ and $\{C_i = 7\} = \bigcup_{j \in \text{Pred}(v_i)} \{C_j = 6\} = \bigcup_{j \in \text{Pred}(v_i)} \{C_{\text{pred}(v_j)} = 4\} \cap G_j$ for $v_i \in V_5$. For the last two layers, we have: $\{C_i = 9\} = \{C_{\text{pred}(v_i)} = 7\} \cap G_i$ for $v_i \in V_6$ and $\{C_n = 10\} = \bigcup_{j \in \text{Pred}(v_i)} \{C_j = 9\} = \bigcup_{j \in \text{Pred}(v_n)} \{C_{\text{pred}(v_j)} = 7\} \cap G_j$. After simplification, we have $\{C_n = 10\} = \bigcup_{j \in \text{Pred}(v_n)} \bigcup_{k \in \text{Pred}(\text{pred}(v_j))} G_{\text{pred}(\text{pred}(v_k))} \cap G_k \cap G_j$. We see that $\Pr[\{C_n = 10\}] = \Pr[F_n]$ because the graph structure of the DAG-MKS instance is the same as REL.

It remains to prove that determining the probability that the makespan is 10 (i.e., $\Pr[\{C_n = 10\}]$) can be done by determining the expected makespan. We use a technique similar to the one used in [8]. We simply add a series of 7 never-failing vertices between the source and the sink, in parallel of the previous graph (see Figure 6). Then, the expected makespan of this new DAG is $\Pr[\{C_n = 10\}] + 9$. □

The general problem (i.e., when task costs are not unitary, when several tasks may be allocated to a given processor, when there is a probability of failure during re-execution, when there are recovery costs, etc.) is thus also #P-completed, and likely even more challenging than DAG-MKS.

B. Lower bound on makespan

Section V-A shows the difficulty of computing the makespan of a schedule where no task is checkpointed. Still, for the sake of comparing the CKPTNONE strategy with our new algorithm CKPTSOME, we derive the following (lower bound) approximation:

Theorem 2. *Consider a schedule for a* G-SPG G *with* P *processors, in which no task is checkpointed. 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*

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

Proof. The idea is to consider a single task of weight W_{par} and to compute its expected execution time as in Equation (2). The only differences are that: (i) we use the platform's exponential failure rate $P\lambda$; and (ii) we neglect the recovery cost. □

In Section VI, we use $EM(G)$ to evaluate the expected makespan of the CKPTNONE strategy. 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) [20], [21], [15]. 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, which are G-SPG (information on the corresponding scientific applications is available in [15], [22]):

- MONTAGE: The NASA/IPAC Montage application stitches together multiple input images to create custom mosaics of the sky. The average weight of a MONTAGE task is 10s. Structurally, MONTAGE is a three-level graph [23]. The first level (reprojection of input image) consists of a bipartite directed graph. The second level (background rectification) is a bottleneck that consists in a join followed by a fork. Then the third level (co-addition to form the final mosaic) is simply a join.
- LIGO: LIGO's Inspiral Analysis workflow is used to generate and analyze gravitational waveforms from data

collected during the coalescing of compact binary systems. The average weight of a LIGO task is 220s. Structurally, LIGO can be seen as a succession of Fork-Joins meta-tasks, that each contains either fork-join graphs or bipartite graphs (see the LIGO IHOPE workflow in [15]). Depending on the number of tasks required, PWG may not output a G-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).

- 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 total number of tasks and is greater than 1000s. Structurally, GENOME starts with many parallel fork-join graphs, whose exit tasks are then both joined into a new exit task, which is the root of fork graphs (see the Epigenomics workflow in [15]).

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 checkpoint it as $c_i = Of_i/BW$ and its recovery time as $r_i = If_i/BW$, where Of_i, resp. If_i, represents the sum of T_i's output, resp. input, file sizes in bytes, and BW is the file system bandwidth in byte/sec. Of_i and If_i are generated by PWG.

In the experiments we consider different exponential processor failure rates. To allow for consistent comparisons of results across different G-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 G-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 (i.e., Of_i and If_i) by a factor. This makes it possible to study the performance impact of I/O operations in a coherent manner across experiments and workflow classes and configurations.

The experiments compare our proposed approach (CKPTSOME) to the two extreme approaches, CKPTALL and CKPTNONE. Recall from Section V-B 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 http://graal.ens-lyon.fr/~yrobert/code.zip.

B. Accuracy of makespan evaluation

In this section, we evaluate the accuracy of the approximation of the expected makespan, using four methods to compute the expected longest path in 2-state probabilistic DAGs. To this end, we follow the methodology in [24], and also reuse the simulator developed by the authors. Essentially, we compare a brute-force Monte Carlo approach [25], [26] with three previously proposed approximation techniques [27], [28], [24], as described hereafter.

MONTECARLO – A task in the DAG succeeds or fails as determined by sampling its 2-state distribution. After sampling, the DAG is deterministic and its makespan can be computed as a longest path. This operation is repeated for a large number of trials, each of which produced a sample makespan. These samples approach the actual makespan distribution as the number of trials increases. Following [24], 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. In all results hereafter we report on the relative error between the three other approximations and this ground truth.

DODIN (approximation by series-parallel graphs) – Basic probability theory tells us how to compute the probability distribution of the sum of two random variables (by a convolution) and of the maximum of two random variables (by taking the product of their cumulative density functions). This simple consideration leads to an exact method to compute the expected makespan when the DAG is series-parallel. When the DAG is not series-parallel, one approach is to approximate it by a series-parallel graph, which is constructed iteratively, first by a sequence of reductions and then by duplicating some vertices. Dodin's method [27] constructs such an approximated series-parallel graph, whose expected makespan is used to estimate that of the original DAG. See [16], [17] for a detailed description of Dodin's method.

NORMAL (approximation via a normality assumption) – The central-limit theorem states that the sum of independent random variables tends to be normally distributed as the number of variables increases. Since the expected makespan of the DAG is a combination of sums and maxima of the original task weights, a popular approach proposed by Sculli [28]

is based on the *normality assumption*: (i) Approximate the distribution of each task by a normal distribution of same mean and variance. This step has constant cost per task for probabilistic 2-state DAGs; (ii) Use Clarke's formulas in [29] to compute the mean and variance of the sum and maximum of two (correlated) normal distributions, and then assuming that they also follow normal distributions; and (iii) Traverse the original DAG and compute the mean and variance of the makespan. See [17] for a full description of Sculli's method.

PATHAPPROX (approximation via longest paths) – A fourth approach is to provide an approximation of the expected makespan based upon non-overlapping failures [24]. Consider a workflow $G = (V, E)$ whose tasks have been assigned to processors, and are all checkpointed. Without failures, the expected makespan $\mathcal{E}(G)$ is the longest path $L(G)$ when weights are chosen according to the first case in Equation (1). With failures, $\mathcal{E}(G)$ can be expressed as an infinite series in λ_{ind}, the common failure rate of individual processors, as follows:

$$\mathcal{E}(G) = L(G) + a_1\lambda_{ind} + a_2\lambda_{ind}^2 + a_3\lambda_{ind}^3 + \dots$$

Intuitively, the term $a_j\lambda_{ind}^j$ corresponds to the average makespan increase when the workflow if struck by j failures. The PATHAPPROX approximation method uses

$$\mathcal{E}(G) = L(G) + a_1\lambda_{ind}$$

and is accurate whenever λ_{ind} is close to zero, which allows us to neglect $O(\lambda_{ind}^2)$ terms. The coefficient a_1 is then computed using a series of traversals, at total cost $O(|V|^2 + |V|.|E|)$ (see [24] for details). Each traversal computes the longest path when one task fails, hence the approach is accurate unless we have more than one failure in a given dependence path of the DAG (hence the name PATHAPPROX). PATHAPPROX has proven both faster and more accurate than the previous three methods for dense LU and QR factorization workflows subject to silent errors [24].

Figure 7 show the relative error (compared to MONTECARLO) of PATHAPPROX, DODIN, and NORMAL vs. the number of tasks for Genome workflows, for CCR=1.6×10^{-2} and the three different p_{fail} values. The results in Figure 7 are representative. Full results for other workflow classes and other CCR values are provided in a companion research report [30]. Although CCR values and workflow classes lead to different curve shapes for a given p_{fail} value, the range of the relative error is consistent (in some cases DODIN is omitted due to taking a prohibitively long time for large workflows). For low failure rates ($p_{\text{fail}} = 0.0001$), PATHAPPROX achieves relative errors well below 0.1% and in most cases orders of magnitude lower that that achieve by DODIN and NORMAL. For moderate failure rates ($p_{\text{fail}} = 0.001$), it achieves relative errors below 10%, which is comparable to that achieved by DODIN and NORMAL, with DODIN typically leading to higher error. For large failure rates ($p_{\text{fail}} = 0.01$), the relative error of PATHAPPROX becomes larger, and is often larger than that of DODIN and NORMAL although their relative errors also increase. This is not surprising: when there are 1,000 tasks, there will be on average 10 failures per execution but PATHAPPROX assumes at most one failure on each path from an entry task to an exit task [24]. This leads to high error when failures are more frequent than this assumption.

The above puts in question the usefulness of PATHAPPROX for (possibly not very relevant to practice) high failure rate scenarios. However, we find that PATHAPPROX remains useful even in such scenarios in terms of comparing checkpoint strategies. As an example, Figure 8 presents results obtained with PATHAPPROX and with MONTECARLO for quantifying the expected makespan of CKPTALL relative to that of CKPTSOME vs. the CCR. Results are shown for each of the three workflow classes, for workflows with 1,000 tasks, and for various numbers of processors P. PATHAPPROX delivers quantitatively accurate (i.e., similar to MONTECARLO) results in all cases except for Montage at very high CCR. However, even in that case, the qualitative prediction of PATHAPPROX is the same as that of MONTECARLO: CKPTALL achieves significantly worse performance than CKPTSOME. Overall, we conclude that PATHAPPROX provides a sound basis for comparing the expected makespan achieved by different scheduling/checkpointing policies in relevant practical configurations.

To assess the scalability of the three methods, we run them (using one core of a 2.3GHz AMD Opteron Processor 8356), with 3000 tasks for GENOME, and $p_{\text{fail}} = 0.0001$. For this large graph, we still ran the Monte Carlo for 300,000 times, so as to ensure the accuracy of the ground truth. Error (normalized difference with Monte Carlo) and execution times are shown in Table I. We see DODIN exhibits very large error. PATHAPPROX is roughly one order of magnitude more accurate than NORMAL. We also see that PATHAPPROX can be computed in under a 0.1 second, while NORMAL and DODIN requires about 1 minute, i.e., about three orders of magnitude slower. We conclude that not only PATHAPPROX achieves a good accuracy, but is also significantly faster than DODIN and NORMAL, which makes it 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 9, 10, and 11 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 [30].

A clear observation is that CKPTSOME always outper-

Table I: Results for GENOME with 3000 tasks and $p_{\text{fail}} = 0.0001$.

	DODIN	NORMAL	PATHAPPROX
Normalized difference with Monte Carlo	-0.11	-2.5×10^{-3}	1.1×10^{-4}
Execution time	around 1 minute	around 1 minute	around 0.08 second

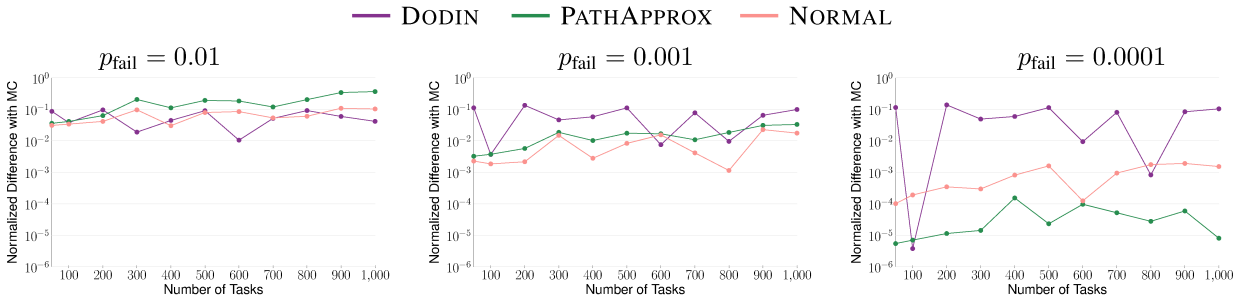


Figure 7: Relative error vs. workflow size for different approximations of the expected makespan for GENOME with 300 processors for CCR $= 1.6 \times 10^{-2}$.

forms CKPTALL.[2] 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), 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 particular 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 situation.

[2]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).

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. For completeness we first review related work devoted to soft errors (Section VII-A). We then review work devoted, like this work, to fail-stop errors (Section VII-B).

A. Soft and silent errors

Many authors have considered soft errors, by which a task execution fails but does not lead to completely losing the data present in the processor memory. Checkpointing, or more precisely making a copy of all task input/output data, is the most widely used technique to address soft errors. If a soft error occurs during its execution, a task can then be re-executed from scratch. This solution can be too costly, and it is possible to save a copy of task input/output data only periodically, at the price of more re-execution when an error is detected. This is the trade-off analyzed by Cao et al. [31] for Cholesky factorization. Several authors have suggested techniques that identify tasks on the critical path, and then making scheduling decisions that attempt to ensure the timely execution of these tasks [32], [33]. A widely used technique to cope with soft errors is task replication, the challenge being to avoid over-duplicating tasks so as strike a good balance between fast failure-free executions and resilient executions [34]. Two representative practical frameworks are the NARBIT system [35], which recovers from soft errors via task replication and work stealing, and Nanos [36], [37], a runtime system that supports the OpenMP programming model.

Silent errors represent a different challenge than soft errors, in that they do not interrupt the execution of the task but corrupt its output data. However, their net effect is the same, since a task must be re-executed whenever a silent error is detected. Since a silent detector is applied at the end of a

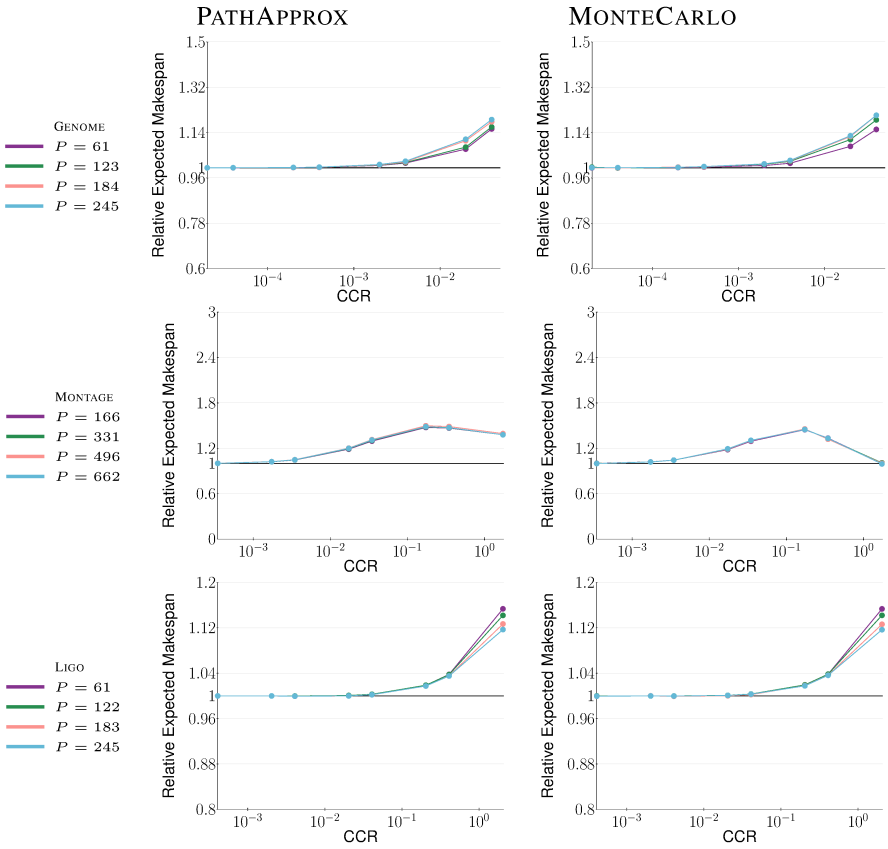


Figure 8: Comparison of the performance predicted by the PathApprox (left column) and by MonteCarlo (right column) for the ratio of the expected makespan of CkptAll with respect to that of CkptSome, when $p_{\text{fail}} = 0.01$, for the three workflows, for 1000 tasks, and for various values of the CCR.

task's execution, the task must be re-executed from scratch in case of an error. Checkpointing (making copies of input/output data) or replicating tasks and comparing outputs, are two common techniques to mitigate the impact of silent errors. With checkpointing, several application-specific detectors can be used to avoid replication and increase performance in failure-free executions. Two well-known examples are Algorithm-Based Fault Tolerance (ABFT) [38], [39], [40] and silent error detectors based on domain-specific data analytics [41], [42], [43].

B. Fail-stop failures

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 [**?**], 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 [**?**] was later extended in [44] 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 [45]. 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 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.

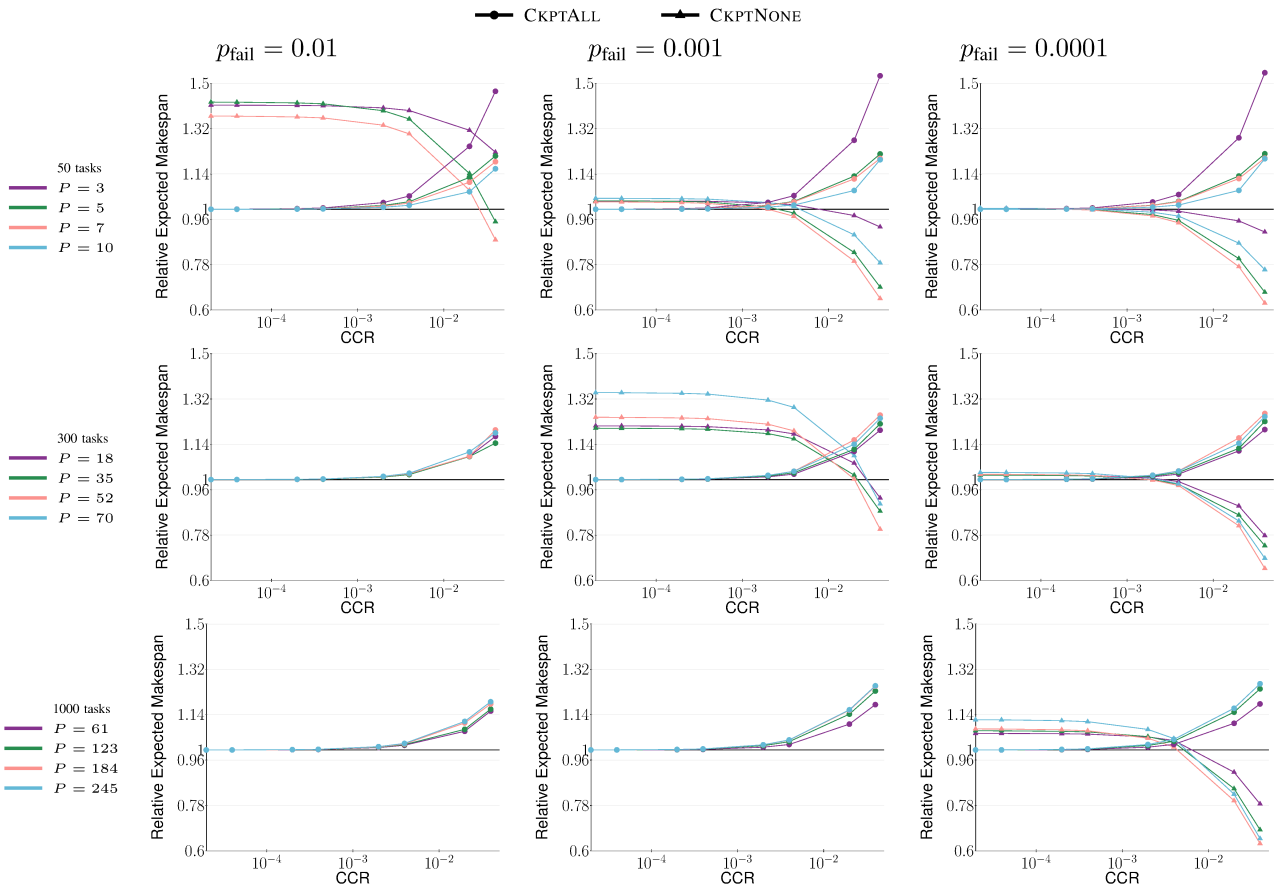


Figure 9: Relative expected makespan of CkptAll and of CkptNone with that of CkptSome for the Genome workflow, three different failure rates, three workflow sizes, and varying Communication-to-Computation Ratio (CCR).

VIII. Conclusion

In this work we have proposed a scheduling/checkpointing algorithm, called CkptSome, for executing workflow applications on 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, Generalized Series-Parallel Graphs (G-SPGs), which are broadly relevant to production workflow applications, we are able to design a sensible algorithm and are able to compute a first-order approximation of the expected makespan of the solutions it produces. A competing approach, CkptAll, sidesteps 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 [20], [21], [15]. 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),

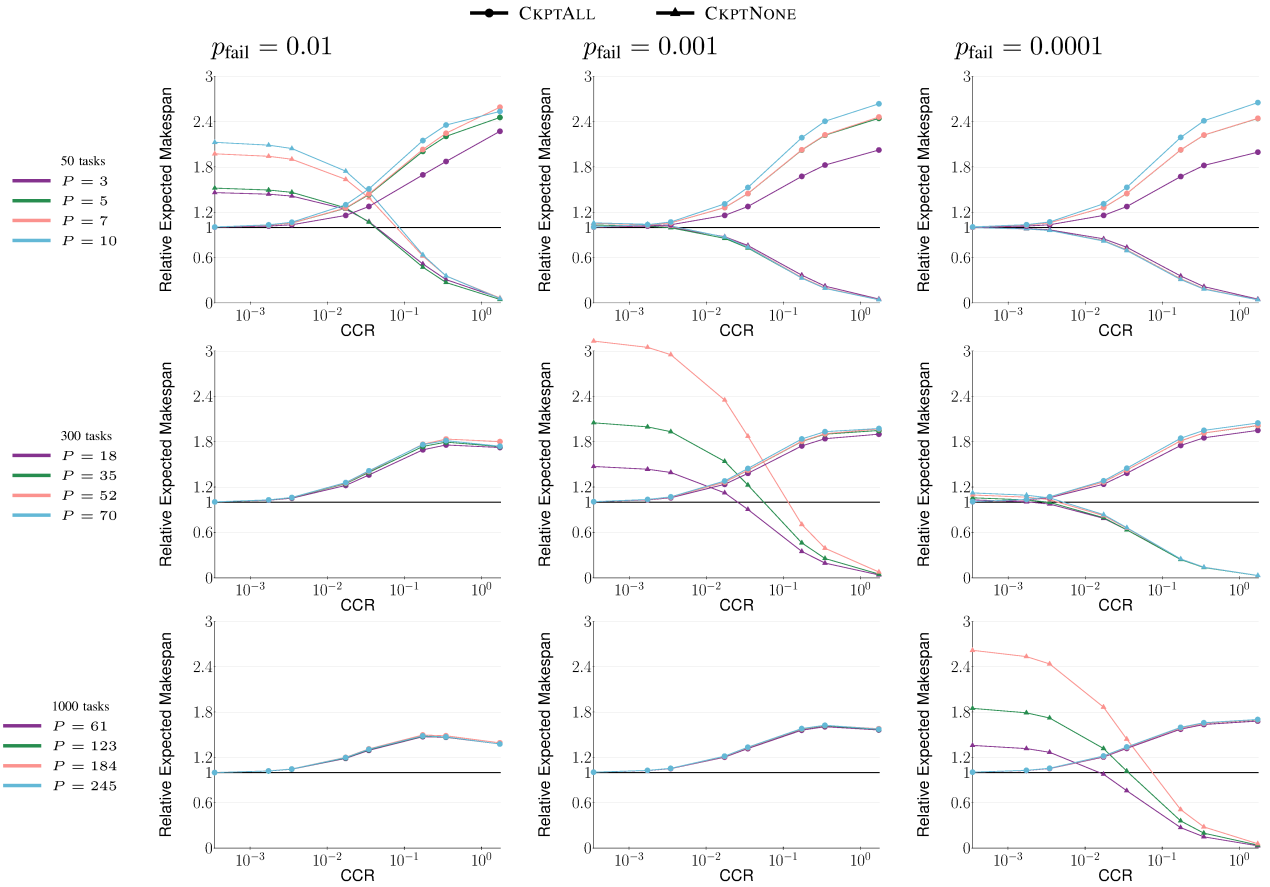


Figure 10: Relative expected makespan of CkptAll and of CkptNone with that of CkptSome for the Montage workflow, three different failure rates, three workflow sizes, and varying Communication-to-Computation Ratio (CCR).

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. 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 [46], which is NP-Complete for general DAGs, but may be amenable to efficient solutions for G-SPGs.

Acknowledgments

This work is supported in part by NSF award SI2-SSE:1642369, and by the PIA ELCI project. Yves Robert is with Institut Universitaire de France.

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.

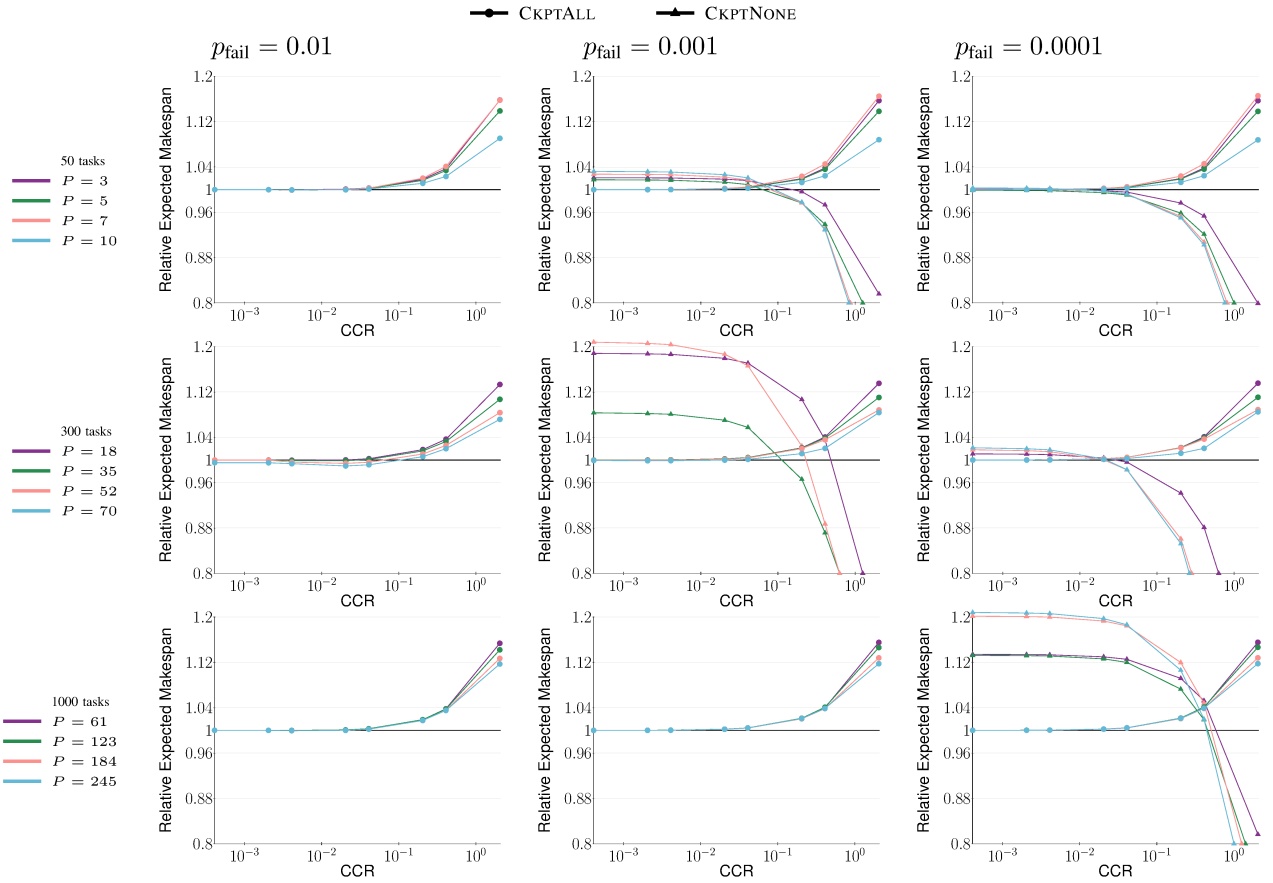


Figure 11: Relative expected makespan of CkptAll and of CkptNone with that of CkptSome for the Ligo workflow, three different failure rates, three workflow sizes, and varying Communication-to-Computation Ratio (CCR).

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

[14] 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.

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

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

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

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

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

[27] B. Dodin, "Bounding the project completion time distribution in PERT networks," *Operations Research*, vol. 33, no. 4, pp. 862–881, 1985.

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

[29] C. E. Clark, "The greatest of a finite set of random variables," *Operations Research*, vol. 9, no. 2, pp. 145–162, 1961.

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

[31] C. Cao, T. Herault, G. Bosilca, and J. Dongarra, "Design for a soft error resilient dynamic task-based runtime," in *IPDPS*. IEEE, 2015, pp. 765–774.

[32] H. Jin, X.-H. Sun, Z. Zheng, Z. Lan, and B. Xie, "Performance Under

Failures of DAG-based Parallel Computing," in *CCGRID '09*. IEEE Computer Society, 2009.

[33] E. Kail, P. fchtpen, and M. Kozlovszky, "A novel adaptive checkpointing method based on information obtained from workflow structure," *Computer Science*, vol. 17, no. 3, 2016.

[34] M. Chtepen, F. H. A. Claeys, B. Dhoedt, F. D. Turck, P. Demeester, and P. A. Vanrolleghem, "Adaptive task checkpointing and replication: Toward efficient fault-tolerant grids," *IEEE Trans. Parallel Distributed Systems*, vol. 20, no. 2, pp. 180–190, 2009.

[35] M. C. Kurt, S. Krishnamoorthy, K. Agrawal, and G. Agrawal, "Fault-tolerant dynamic task graph scheduling," in *SC '14*. IEEE Press, 2014, pp. 719–730.

[36] J. A. Moreno, O. S. Unsal, J. Labarta, and A. Cristal, "Nanocheckpoints: A task-based asynchronous dataflow framework for efficient and scalable checkpoint/restart," in *23rd Euromicro PDP*, 2015, pp. 99–102.

[37] O. Subasi, O. S. Ünsal, J. Labarta, G. Yalcin, and A. Cristal, "Crc-based memory reliability for task-parallel HPC applications," in *IPDPS*, 2016, pp. 1101–1112.

[38] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Comput.*, vol. 33, no. 6, pp. 518–528, 1984.

[39] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou, "Algorithm-based fault tolerance applied to high performance computing," *J. Parallel Distrib. Comput.*, vol. 69, no. 4, pp. 410–416, 2009.

[40] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, "Fault tolerant preconditioned conjugate gradient for sparse linear system solution," in *ICS*. ACM, 2012.

[41] E. Berrocal, L. Bautista-Gomez, S. Di, Z. Lan, and F. Cappello, "Lightweight silent data corruption detection based on runtime data analysis for HPC applications," in *HPDC*. ACM, 2015.

[42] L. Bautista Gomez and F. Cappello, "Detecting silent data corruption through data dynamic monitoring for scientific applications," *SIGPLAN Notices*, vol. 49, no. 8, pp. 381–382, 2014.

[43] ——, "Detecting and correcting data corruption in stencil applications through multivariate interpolation," in *FTS*. IEEE, 2015.

[44] 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.

[45] 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.

[46] 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.