Checkpointing Workflows à la Young/Daly Is Not Good Enough

ANNE BENOIT, LUCA PEROTIN, and YVES ROBERT, Laboratoire LIP, ENS Lyon, France
HONGYANG SUN, University of Kansas, USA

This article revisits checkpointing strategies when workflows composed of multiple tasks execute on a parallel platform. The objective is to minimize the expectation of the total execution time. For a single task, the Young/Daly formula provides the optimal checkpointing period. However, when many tasks execute simultaneously, the risk that one of them is severely delayed increases with the number of tasks. To mitigate this risk, a possibility is to checkpoint each task more often than with the Young/Daly strategy. But is it worth slowing each task down with extra checkpoints? Does the extra checkpointing make a difference globally? This article answers these questions. On the theoretical side, we prove several negative results for keeping the Young/Daly period when many tasks execute concurrently, and we design novel checkpointing strategies that guarantee an efficient execution with high probability. On the practical side, we report comprehensive experiments that demonstrate the need to go beyond the Young/Daly period and to checkpoint more often for a wide range of application/platform settings.

CCS Concepts: • **Software and its engineering** → *Checkpoint/restart*; • **Computer systems organization** → *Reliability*; • **Hardware** → *System-level fault tolerance;*

Additional Key Words and Phrases: Checkpoint, workflow, concurrent tasks, Young/Daly formula

ACM Reference format:
Anne Benoit, Luca Perotin, Yves Robert, and Hongyang Sun. 2022. Checkpointing Workflows à la Young/Daly Is Not Good Enough. *ACM Trans. Parallel Comput.* 9, 4, Article 14 (December 2022), 25 pages.
https://doi.org/10.1145/3548607

1 INTRODUCTION

Checkpointing is the standard technique to protect applications running on **High Performance Computing (HPC)** platforms. Every day, the platform will experience a few fail-stop errors (or failures; we use both terms interchangeably). After each failure, the application executing on the faulty processor (and likely on many other processors for a large parallel application) is interrupted and must be restarted. Without checkpointing, all the work executed for the application is lost. With checkpointing, the execution can resume from the last checkpoint, after some downtime (enroll a spare to replace the faulty processor) and a recovery (read the checkpoint).

Yves Robert also with University of Tennessee, Knoxville, TN, USA.
Authors' addresses: Y. Robert (corresponding author), A. Benoit, and L. Perotin, Laboratoire LIP, ENS Lyon, 69364 Lyon Cedex 07, France; emails: {yves.robert, anne.benoit, lucas.perotin}@ens-lyon.fr; H. Sun, Department of Electrical Engineering and Computer Science at the University of Kansas, 1520 W 15th St Lawrence, KS 66045, USA; email: hongyang.sun@ku.edu.

2329-4949/2022/12-ART14 $15.00
https://doi.org/10.1145/3548607

Consider an application, composed of a unique task, executing on a platform whose nodes are subject to fail-stop errors. Say the application executes for $T_{base} = 10$ hours, can checkpoint in $C = 6$ minutes, and experiences failures whose inter-arrival times follow an Exponential distribution with mean $\mu = 4$ hours. This means that a failure strikes the application every 4 hours in expectation (see Section 2.1 for details). Assume a short downtime $D = 1$ minute and a recovery time $R = C$. How frequently should the task be checkpointed so that its expected execution time $\mathbb{E}(T_{1\text{-}task})$ is minimized? There is a well-known tradeoff: Taking too many checkpoints leads to a high overhead, especially when there are few failures, while taking too few checkpoints leads to a large re-execution time after each failure. Here is an illustration of this tradeoff:

- If we take no checkpoint at all, then the expected execution time is $\mathbb{E}(T_{1\text{-}task}) \approx 46$ hours (see Equation (1) in Section 2.5 to derive this value);
- If we take a checkpoint at the end of the execution, e.g., to save final results on stable storage,[1] then $\mathbb{E}(T_{1\text{-}task})$ increases by about 76 minutes, which is surprising given the short checkpoint time; but keep in mind that if a failure strikes during the checkpoint, which happens with a low probability of 2.5%, then the 10 hours of execution are wasted;
- If we checkpoint every hour (an application-agnostic approach that has been implemented for several HPC platforms [25]), then we have 10 equal-length segments, each of duration of 1 hour and followed by a checkpoint. We obtain $\mathbb{E}(T_{1\text{-}task}) \approx 13$ hours. Checkpointing every hour brings a huge benefit!
- Finally, if we checkpoint every 20 minutes, then we obtain $\mathbb{E}(T_{1\text{-}task}) \approx 14$ hours. Checkpointing too frequently becomes an overkill.

The optimal checkpointing period is given by the Young/Daly formula as $W_{YD} = \sqrt{2\mu C}$ [12, 38], where μ is the application **Mean Time Between Failures (MTBF)** and C the checkpoint duration. In the example above with $\mu = 4$ hours and $C = 6$ minutes, we obtain $W_{YD} \approx 54$ minutes. This value would be the optimal checkpointing period for a task of infinite length. For a task of length $T_{base} = 10$ hours, the optimal solution (see Section 2.5) is to use either $\max(1, \lfloor \frac{T_{base}}{W_{YD}} \rfloor) = 11$ or $\lceil \frac{T_{base}}{W_{YD}} \rceil = 12$ equal-length segments, whichever leads to the smaller $\mathbb{E}(T_{1\text{-}task})$. We find that the best value is $\mathbb{E}(T_{1\text{-}task}) \approx 13$ hours with 12 segments of length 50 minutes. The best value is smaller by only 1 minute than the value with 11 segments, and by only 3 minutes than the value with 10 segments, which shows the robustness of the approach.

We now move to a more complicated example and assume that 300 independent applications have been launched concurrently on the platform. These 300 applications are identical to the application above: Each has a unique task of length $T_{base} = 10$ hours, checkpoint duration $C = 6$ minutes, recovery time $R = C$, and the downtime is $D = 1$ minute. For the example to be more realistic in terms of failure rate, we assume that each application executes with $p = 30$ processors. Hence, the platform has at least $m = 9{,}000$ processors. Each processor is subject to failures following an Exponential distribution $Exp(\frac{1}{\mu_{ind}})$, where μ_{ind} is the individual processor's MTBF. Since each task executes on $p = 30$ processors, its MTBF is $\mu = \frac{\mu_{ind}}{p}$. In other words, the MTBF of a task is inversely proportional to the number of processors enrolled, which is intuitive in terms of failure frequency (see Reference [25] for a formal proof). We now assume that each task has 0.5% chances to fail during execution; this setting corresponds to an individual MTBF μ_{ind} such that $1 - e^{-\frac{pT_{base}}{\mu_{ind}}} = 0.005$, i.e., $\mu_{ind} = 59{,}850$ hours (or 6.8 years). This is in accordance with MTBFs typically observed on

[1] We make this assumption throughout the article for simplicity. Section B of the Web Supplementary Material extends the analysis to the case where no checkpoint is taken at the end of the execution of a task. Changes are minimal, and results are quite similar.

large-scale platforms, which range from a few years to a few dozens of years [8]. For each task, the Young/Daly period is $W_{YD} = \sqrt{2\frac{\mu_{ind}}{p}C} \approx 20$ hours, and the expected execution time of a single task $\mathbb{E}(T_{1\text{-}task})$ is minimized either when no checkpoint is taken or if a single checkpoint is taken at the end of the execution (see Section 2.6). Recall that, as stated above, we assume that we always take a checkpoint at the end of the execution of a task. Then, we derive that $\mathbb{E}(T_{1\text{-}task}) \approx 10.4$ with a single checkpoint taken at the end of each task (see Section 2.6 for details of this computation).

Is it safe to checkpoint each task individually à la Young/Daly? The problem comes from the fact that the expectation $\mathbb{E}(T_{all\text{-}tasks})$ of the maximum execution time over all tasks, i.e., the expectation of the total time required to complete all tasks, is far larger than the maximum of the expectations (which in the example have all the same value $\mathbb{E}(T_{1\text{-}task})$). When a single checkpoint is taken at the end of each task, we compute that $\mathbb{E}(T_{all\text{-}tasks}) > 14$, while adding four intermediate checkpoints to each task reduces it down to $\mathbb{E}(T_{all\text{-}tasks}) < 12.75$ (see Section 2.6 for details of the computation of both numbers). Intuitively, this is because adding these intermediate checkpoints greatly reduces the chance of re-executing any single task from scratch when it is struck by a failure, and the probability of having at least one failed task increases with the number of tasks. Of course, there is a penalty from the user's point of view: Adding four checkpoints to each task augments their length by 24 minutes, while the majority of them will not be struck by a failure. In other words, users may feel that their response time has been unduly increased and state that it is not worth to add these extra checkpoints.

Going one step further, consider now a single application whose dependence graph is a simple fork-join graph, made of 302 tasks: an entry task, 300 parallel tasks identical to the tasks above (each task runs on $p = 30$ processors for $T_{base} = 10$ hours, and is checkpointed in $C = 6$ minutes), and an exit task. Such applications are typical of HPC applications that explore a wide range of parameters or launch subproblems in parallel. Now, the extra checkpoints make full sense, because the exit task cannot start before the last parallel task has completed. The expectation of the total execution time is $\mathbb{E}(T_{total}) = \mathbb{E}(T_{entry}) + \mathbb{E}(T_{all\text{-}tasks}) + \mathbb{E}(T_{exit})$, where $\mathbb{E}(T_{entry})$ and $\mathbb{E}(T_{exit})$ are the expected durations of the entry and exit tasks, and $\mathbb{E}(T_{total})$ is minimized when $\mathbb{E}(T_{all\text{-}tasks})$ is minimized. By diminishing $\mathbb{E}(T_{all\text{-}tasks})$, we save 1.25 hour, or 75 minutes (and in fact much more than that, because the lower and upper bounds for $\mathbb{E}(T_{all\text{-}tasks})$ are loosely computed).

This last example shows that the optimal execution of large workflows on failure-prone platforms requires to checkpoint each workflow task more frequently than prescribed by the Young/Daly formula. The main focus of this article is to explore various checkpointing strategies, and our main contributions are as follows:

- We provide approximation bounds for the performance of MinExp, a strategy à la Young/Daly that minimizes the expected execution time of each task and for a novel strategy CheckMore that performs more checkpoints than MinExp.
- Both bounds apply to workflows of arbitrary shape and whose tasks can be either rigid or moldable. In addition, we exhibit an example where the bounds are tight and where CheckMore can be an order of magnitude better than MinExp.
- The novel CheckMore strategy comes in two flavors, one that tunes the number of checkpoints as a function of the degree of parallelism in the failure-free schedule and a simpler one that does not require any knowledge of the failure-free schedule, beyond a priority list to decide in which order to start executing the tasks.
- We report comprehensive simulations results based on WorkflowHub testbeds [19], which demonstrate the significant gain brought by CheckMore over MinExp for almost all testbeds.

Table 1. Key Notations

m	Total number of processors
p	Number of processors per task
n	Number of tasks
$\mu_{ind} = \frac{1}{\lambda}$	Individual processor's MTBF
C	Checkpoint time
R	Recovery time
D	Downtime
T_{base}	Task duration without failures
N_{opt}	Number of segments with LAMBERT strategy
N_{ME}	Number of segments with MINEXP strategy
W_{ME}	Segment length with MINEXP strategy

The article is organized as follows. We first describe the model in Section 2. We assess the performance of MINEXP in Section 3; performance bounds are proven both for independent tasks and for general workflows. Section 4 presents the novel strategy CHECKMORE that checkpoints workflow tasks more often than MINEXP and analyzes its theoretical performance. The experimental evaluation in Section 5 presents extensive simulation results comparing both strategies. Finally, we discuss related work in Section 6 and conclude in Section 7.

2 MODEL AND BACKGROUND

In this section, we first detail the platform and application models and describe how to practically deploy a workflow with checkpointed jobs. Then, we discuss the objective function before providing background on the optimal checkpointing period for preemptible tasks and getting back to the example of the introduction. Key notations are summarized in Table 1.

2.1 Platform

We consider a large parallel platform with m identical processors, or nodes. These nodes are subject to fail-stop errors or failures. A failure interrupts the execution of the node and provokes the loss of its whole memory. There are many causes of failures, including power outages or network errors, and they cause the node to stall or crash [14, 34]. Consider a parallel application running on several nodes: When one of these nodes is struck by a failure, the state of the application is lost, and execution must restart from scratch, unless a fault-tolerance mechanism has been deployed.

The classical technique to deal with failures makes use of a checkpoint-restart mechanism: The state of the application is periodically checkpointed, i.e., all participating nodes take a checkpoint simultaneously. This is the standard coordinated checkpointing protocol, which is routinely used on large-scale platforms [10], where each node writes its share of the application data to stable storage (checkpoint of duration C). When a failure occurs, the platform is unavailable during a downtime D, which is the time to enroll a spare processor that will replace the faulty processor [12, 25]. Then, all application nodes (including the spare) recover from the last valid checkpoint in a coordinated manner, reading the checkpoint file from stable storage (recovery of duration R). Finally, the execution is resumed from that point onward, rather than starting again from scratch. Note that failures can strike during checkpoint and recovery but not during downtime (otherwise, we can include the downtime in the recovery time).

Throughout the article, we add a final checkpoint at the end of each application task to write final outputs to stable storage. Symmetrically, we add an initial recovery when re-executing the first checkpointed segment of a task (to read inputs from stable storage) if it has been struck by

a failure before completing the checkpoint. See Section B of the **Web Supplementary Material (WSM)** for an extension relaxing either or both assumptions.

We assume that each node experiences failures whose inter-arrival times follow an Exponential distribution $Exp(\lambda)$ of parameter $\lambda > 0$, whose Probability Density Function is $f(x) = \lambda e^{-\lambda x}$ for $x \geq 0$. The individual MTBF of each node is $\mu_{ind} = \frac{1}{\lambda}$. Even if each node has an MTBF of several years, large-scale parallel platforms are composed of so many nodes that they will experience several failures per day [8, 18]. Hence, a parallel application using a significant fraction of the platform will typically experience a failure every few hours.

2.2 Application

We focus on HPC applications expressed as workflow graphs, such as those available in WorkflowHub [19] (formerly Pegasus [36]). The shape of the task graph is arbitrary, and the tasks can be parallel. We further assume that all tasks are preemptible, i.e., that we can take a checkpoint at any instant.

For the theoretical analysis, we use workflows whose tasks can be rigid or moldable parallel tasks. A moldable task can be executed on an arbitrary number of processors, and its execution time depends on the number of processors allotted to it. This corresponds to a variable static resource allocation, as opposed to a fixed static allocation (rigid tasks) and a variable dynamic allocation (malleable tasks) [16]. Scheduling rigid or moldable workflows is a difficult NP-hard problem (see the related work in Section 6). We take as input a failure-free schedule for the workflow and transform it by adding checkpoints as follows. The failure-free schedule provides an ordered list of tasks, sorted by non-decreasing starting times. Our failure-aware algorithms are list schedules that greedily process the tasks (augmented with checkpoints) in this order: If task T is number i in the original failure-free schedule, then T is scheduled after the $i-1$ first tasks in the failure-aware schedule, and no other task can start before T does. Hence, the processors allocated to T in the failure-aware schedule may differ from those allocated in the failure-free schedule. Enforcing the same ordering of execution of the tasks may be sub-optimal, but it is the key to guarantee approximation ratios for the total execution time.

For the experiments, we restrict to workflows with uni-processor tasks, in accordance with the characteristics of the workflow benchmarks from WorkflowHub.

2.3 Implementation in a Cluster Environment

This section briefly describes two approaches to deploy a workflow with checkpointed jobs in a cluster environment.

The first approach is to use the job scheduler LSF [28] and to submit a set of jobs with their dependencies: There are as many jobs as tasks in the workflow, and these jobs are declared *checkpointable*. The system will relaunch a job after it is hit by a failure, from the last checkpoint on and until success (see the "job failover" section in Reference [28]). If the failed job was using j processors, then it releases $j-1$ surviving processors right after the failure; if there is at least one other processor available, then the job can be rescheduled right away (jobs usually get high priority when they are rescheduled after a failure). Otherwise, the failed job will have to wait and this waiting time, a.k.a. the re-submission time, is dependent on the platform scheduling policy and on the availability of nodes.

A second approach is to submit a single job with $p+q$ processors, where p processors represent the allotment for the whole workflow and q processors are spare. The job uses a master process that spans the workflow tasks and controls how their execution progresses; the tasks are checkpointed using a standard software such as VeloC [9]. The spare nodes are mutualized across the tasks either by using a fault-tolerant MPI library like ULFM [6, 15] or by having the master process launch each

task as independent MPI applications spanning on subgroups of the reservation and re-launching them from their last checkpoint on the surviving nodes and the spare nodes if some task is subject to failure.

In the first approach, the downtime would be non-constant, because it corresponds to the re-submission time, while in the second approach with spares, the downtime can be approximated as a constant. Regardless, all the results of this article are taken in expectation, and they extend to using an average value of the downtime whenever a fixed value is not appropriate.

Finally, we stress that this work is agnostic of system management policies and does not modify any parameter specified by the user for the job allocations; we simply increase the checkpoint frequency when needed, which results in shorter execution time and better processor utilization for the workflow.

2.4 Objective Function

Given a workflow composed of a set of tasks, where each task executes on a given number of processors, the objective function is to minimize the expected makespan of the workflow, i.e., the expected total execution time to complete all tasks. We aim at determining the best checkpointing strategy for the tasks that compose the workflow. This is the only parameter that we modify in the execution: We keep the number of processors specified by the user, and we even keep the order of the tasks as given by the user schedule. The replacement of failed nodes, or the resubmission of failed tasks, is decided by the system and does not depend upon the checkpointing policy, either à la Young/Daly or one of our new strategies.

As a result, minimizing the expected makespan of the workflow also maximizes processor utilization of the platform, because the processors reserved by the user will be released earlier and with no additional cost for the rest of the platform.

In the analysis of the checkpointing strategies, we focus on bounding the *ratio*, which is defined as the expected makespan of the workflow (i.e., the expected total execution time) divided by the makespan in the failure-free execution (no checkpoints nor failures), given a user-specified schedule. Hence, the ratio shows the overhead induced by failures and the checkpointing strategy: The closer to one the better.

2.5 Checkpointing Period

Consider an application A composed of a single parallel task executing on p processors. Assume that the task is preemptible, which means that it can be checkpointed at any instant. The key for an efficient checkpointing policy is to decide when to checkpoint. If the application A runs for a duration T_{base} (base time without checkpoints nor failures), then the optimal checkpointing strategy, i.e., the strategy minimizing the expected execution time of the application, can be derived as shown below.

Lemma 1. *The expected time $\mathbb{E}(W, C, R)$ to execute a segment of W seconds of work followed by a checkpoint of C seconds and with recovery cost R seconds is*

$$\mathbb{E}(W, C, R) = \left(\frac{1}{p\lambda} + D\right) e^{p\lambda R} \left(e^{p\lambda(W+C)} - 1\right). \tag{1}$$

Proof. This is the result of Reference [7, Theorem 1]. Note that Lemma 1 also applies when the segment is not followed by a checkpoint (take $C = 0$). □

The *slowdown function* is defined as $f(W, C, R) = \frac{\mathbb{E}(W,C,R)}{W}$. We have the following properties:

Lemma 2. *The slowdown function $W \mapsto f(W, C, R)$ has a unique minimum W_{opt} that does not depend on R, is decreasing in the interval $[0, W_{opt}]$ and s increasing in the interval $[W_{opt}, \infty)$.*

Proof. Again, this is the result of Reference [7, Theorem 1]. The exact value of W_{opt} is obtained using the Lambert $\mathcal{W}$ function (see Section A of the WSM for background), but a first-order approximation is the Young/Daly formula $W_{YD} = \sqrt{\frac{2C}{p\lambda}}$. □

Lemma 2 shows that infinite tasks should be partitioned into segments of size W_{YD} followed by a checkpoint. What about finite tasks? Back to our application A of duration T_{base}, we partition it into N_c segments of length W_i, $1 \le i \le N_c$, each followed by a checkpoint C. By linearity of the expectation, the expected time to execute the application A is

$$\mathbb{E}(A) = \sum_{i=1}^{N_c} \mathbb{E}(W_i, C, R) = \left(\frac{1}{p\lambda} + D\right) e^{p\lambda R} \sum_{i=1}^{N_c} \left(e^{p\lambda(W_i+C)} - 1\right),$$

where $\sum_{i=1}^{N_c} W_i = T_{base}$. By convexity of the Exponential function, or by using Lagrange multipliers, we see that $\mathbb{E}(A)$ is minimized when the W_i's take a constant value, i.e., all segments have same length. Thus, we obtain $W_i = \frac{T_{base}}{N_c}$ for all i, and we aim at finding N_c that minimizes

$$\mathbb{E}(A) = N_c \mathbb{E}\left(\frac{T_{base}}{N_c}, C, R\right) = f\left(\frac{T_{base}}{N_c}, C, R\right) \times T_{base},$$

where f is the slowdown function. Define $K_{opt} = \frac{T_{base}}{W_{opt}}$, where W_{opt} achieves the minimum of the slowdown function. K_{opt} would be the optimal value if we could have a non-integer number of segments. Lemma 2 shows that the optimal value N_{ME} of N_c is either $N_{opt} = \max(1, \lfloor K_{opt} \rfloor)$ or $N_{ME} = \lceil K_{opt} \rceil$, whichever leads to the smallest value of $\mathbb{E}(A)$. In the experiments, to avoid the numerical evaluation of the Lambert function for W_{opt}, we use the simplified expression $N_{ME} = \lceil \frac{T_{base}}{W_{YD}} \rceil$:

Definition 1. The MinExp checkpointing strategy partitions a parallel task of length T_{base}, with p processors and checkpoint time C, into $N_{ME} = \lceil \frac{T_{base}}{W_{YD}} \rceil$ equal-length segments, each followed by a checkpoint, where $W_{YD} = \sqrt{\frac{2C}{p\lambda}} = \sqrt{\frac{2\mu_{ind}C}{p}}$. Each segment is of length $W_{ME} = \frac{T_{base}}{N_{ME}}$.

The experimental results in Section 5 show that using N_{ME}, whose value is based upon the Young/Daly formula, leads to almost the same results as when using N_{opt}, whose value is based on the Lambert function. See Section A of the WSM for background on how to compute N_{opt}.

2.6 Back to the Example

In the Introduction, we used the example of 300 identical tasks, each with $T_{base} = 10$ hours, $p = 30$, and $C = 6$ minutes. We also had $D = 1$ minute and $R = C$. We assume that each task has 0.5% chances to fail during execution, which corresponds to an individual MTBF μ_{ind} such that $1 - e^{-\frac{pT_{base}}{\mu_{ind}}} = 0.005$. This equality leads to $\mu_{ind} = 59,850$ hours. We derive $W_{YD} = \sqrt{2\mu_{ind}C/p} \approx 20$ hours, hence $N_{ME} = 1$. With a single segment, we then compute the optimal expected execution time $\mathbb{E}(T_{1\text{-}task})$ for each task as

$$\mathbb{E}(T_{1\text{-}task}) = \left(\frac{\mu_{ind}}{p} + D\right) e^{\frac{pR}{\mu_{ind}}} \left(e^{\frac{p}{\mu_{ind}}(T_{base}+C)} - 1\right) \approx 10.4.$$

With 300 tasks executing concurrently, we compute that the expectation of the total time required to complete all tasks is at least $\mathbb{E}(T_{all\text{-}tasks}) > 14$, hence the ratio is $\frac{14}{10} = 1.4$. Indeed, there is no failure at all with probability $(e^{-\frac{p(T_{base}+C)}{\mu_{ind}}})^{300} < 0.23$, and in this case the execution time is $T_{base} + C = 10.1$. The other case, happening with a probability larger than 0.77, is when at least one failure occurs in the process, and we will bound its expected execution time if exactly one failure

occurs, which is clearly lower than the actual expected execution time. To that end, we compute the expected time lost before the failure occurs when attempting to successfully execute for $T = T_{base} + C$ hours: $\mathbb{E}(T_{lost}(T)) = \int_0^\infty x\mathbb{P}(X = x|X < T)dx = \frac{1}{\mathbb{P}(X<T)} \int_0^T xp\lambda e^{-p\lambda x}dx$, with $\mathbb{P}(X < T) = 1 - e^{-p\lambda T}$. Integrating by parts, we derive that

$$\mathbb{E}(T_{lost}(T)) = \frac{1}{p\lambda} - \frac{T}{e^{p\lambda T} - 1}. \tag{2}$$

In the example, we have $T = T_{base} + C = 10.1$, $p = 30$, and $\lambda = \frac{1}{\mu_{ind}} = \frac{-\ln(0.995)}{pT_{base}}$. Thus, if a failure strikes one of the tasks, then the expected time lost is higher than 5.045 hours. After that, we also have to wait $D > 0.016$ hour of downtime and recover for a duration of $R = 0.1$ hour. Overall, the expected execution time satisfies $\mathbb{E}(T_{all\text{-}tasks}) \geq 10.1 + 0.77 \times (\mathbb{E}(T_{lost}(T)) + R + D) > 10.1 + 0.77 \times 5.161 > 14$. Note that this lower bound is far from tight.

When adding four intermediate checkpoints to each task, we obtain $\mathbb{E}(T_{all\text{-}tasks}) < 12.75$. Indeed, the tasks are now slightly longer (10.5 hours without failure), and they fail with probability $1 - e^{-\frac{30\times 10.5}{59850}} < 0.006$. Let M_f denote the maximum number of failures of any tasks. Clearly, we have $\mathbb{P}\{M_f \geq k\} \leq 300 \times 0.006^k$. The worst-case scenario for each failure is when it happens just before the end of a checkpoint, and in that case we loose at most $2 + 0.1 + 0.1 + 0.017 < 2.22$ for each failure (the length of a segment, the checkpoint time, the recovery time and the downtime). Thus, $\mathbb{E}(T_{all\text{-}tasks}) < 10.5 + 2.22 \sum_{k\geq 1} \mathbb{P}\{M_f \geq k\} < 10.5 + 2.22 + 2.22 \times 300 \times \sum_{k\geq 2} 0.006^k < 12.75$, hence a ratio lower than 1.275 to compare with 1.4 with the MinExp strategy. Note that this upper bound is far from tight. This example shows that the optimal checkpointing strategy should not only be based upon the task profiles but also upon the number of other tasks that are executing concurrently.

3 YOUNG/DALY FOR WORKFLOWS: THE MINEXP STRATEGY

In this section, we prove performance bounds for the MinExp checkpointing strategy, which adds N_{ME} checkpoints to each task, thereby minimizing the expected execution time for each task. We start in Section 3.1 with independent tasks, first identical and then arbitrary, that can be executed concurrently (think of a shelf of tasks). Next, we move to general workflows in Section 3.2.

3.1 MinExp for Independent Tasks

We start with a word of caution: Throughout this section, the proofs of the theorems and the analysis of the examples are long and technically involved. We state the results and provide proof sketches in the text below; all details are available in the WSM.

3.1.1 Identical Independent Tasks. First, we consider identical independent tasks that can be executed concurrently. Recall that m is the total number of processors. We identify a task $\mathcal{T}$ with its type (i.e., set of parameters) $\mathcal{T} = (T_{base}, p, C, R)$: length T_{base}, number of processors p, checkpoint time C, and recovery time R.

Theorem 1. *Consider n identical tasks of same type $\mathcal{T} = (T_{base}, p, C, R)$ to be executed concurrently on $n \times p \leq m$ processors with individual failure rate $\lambda = \frac{1}{\mu_{ind}}$. The downtime is D. For the MinExp strategy, N_{ME} is the number of checkpoints, and W_{ME} is the length of each segment, as given by Definition 1. Let $P_{suc}(\tilde{R}) = e^{-p\lambda(W_{ME}+C+\tilde{R})}$ be the probability of success of a segment with re-execution cost $\tilde{R}$ ($\tilde{R} = 0$ if no re-execution, or $\tilde{R} = R$ otherwise), and $Q^* = \frac{1}{1-P_{suc}(R)}$. Let the ratio be $r_{id}^{ME}(n, \mathcal{T}) = \frac{\mathbb{E}(T_{tot})}{T_{base}}$,*

where $\mathbb{E}(T_{tot})$ is the expectation of the total time T_{tot} of the MinExp strategy. We have the following:

$$r_{id}^{ME}(n,\mathcal{T}) \leq \left(\frac{\log_{Q^*}(n)}{N_{ME}} + \log_{Q^*}(\log_{Q^*}(n)) + 1 + \frac{\ln(Q^*)}{12N_{ME}} + \frac{1}{\ln(Q^*)N_{ME}}\right) \times \left(1 + \frac{C+R+D}{W_{ME}}\right) + \frac{C}{W_{ME}} + 1 + o(1). \tag{3}$$

Note that if n is small, then the ratio holds by replacing all negative or undefined terms by 0.

Proof. First, a segment consists of the re-execution cost $\tilde{R}$, the work W_{ME}, and the checkpoint cost C. Since failures may occur during recovery or checkpoint, the total processing time is $W_{ME} + \tilde{R} + C$. Thus, given the exponential failure probability, we have $P_{suc}(\tilde{R}) = e^{-p\lambda(W_{ME}+C+\tilde{R})}$. The MinExp strategy is a $r_{id}^{ME}(n,\mathcal{T})$-approximation of the base time $T_{base} = N_{ME}W_{ME}$, hence also of the optimal expected execution time. Let M_f be the maximum number of failures over all tasks. We process N_{ME} segments of length $W_{ME} + C$, and each failure in a segment incurs an additional time upper bounded by $D+R+W_{ME}+C$. The expectation $\mathbb{E}(T_{tot})$ of the total time T_{tot} of the MinExp strategy is at most

$$\mathbb{E}(T_{tot}) \leq T_{base} + N_{ME}C + \mathbb{E}(M_f)(W_{ME} + C + R + D),$$

and hence

$$r_{id}^{ME}(n,\mathcal{T}) = \frac{\mathbb{E}(T_{tot})}{T_{base}} \leq 1 + \frac{C}{W_{ME}} + \frac{\mathbb{E}(M_f)}{N_{ME}}\left(1 + \frac{C+R+D}{W_{ME}}\right). \tag{4}$$

We continue with the computation of $\mathbb{E}(M_f)$. We first study the **random variable (RV)** N_f of the number of failures before completing a given task. We have identical segments $(s_1, s_2, \ldots)$ to process, each of them having a probability of success $p_{s_i} \in \{P_{suc}(R), P_{suc}(0)\}$, and we stop upon reaching the N_{ME} successes. Hence, s_1 is the first trial of the first segment; if s_1 succeeds, which happens with probability $P_{suc}(0)$, then s_2 corresponds to the first trial of the second segment and succeeds with probability $P_{suc}(0)$; otherwise, s_2 corresponds to the second trial of the first segment and succeeds with probability $P_{suc}(R)$. We are interested in the number of failures N_f before having N_{ME} successes. Clearly, if N_f' represents the RV for the same problem except that all segments have the same probability of success $P_{suc}(R)$, then all segments are less likely or equally likely to succeed, and

$$\forall x, \mathbb{P}\{N_f' \leq x\} \leq \mathbb{P}\{N_f \leq x\}. \tag{5}$$

Now, let M_f' be the RV equal to the maximum of n independent and identically distributed RVs following N_f'. Equation (5) leads to $\mathbb{E}(M_f') \geq \mathbb{E}(M_f)$. Each N_f' is a negative binomial RV with parameters $(N_{ME}, P_{suc}(R))$. We refine the analysis from Reference [21] by bounding the sum of some Fourier coefficients (see Section C of the WSM for details) to show that

$$\mathbb{E}(M_f') \leq \log_{Q^*}(n) + (N_{ME} - 1)\log_{Q^*}(\log_{Q^*}(n)) + N_{ME} + \left(\frac{\ln(Q^*)}{12} + \frac{1}{\ln(Q^*)}\right) + o(1). \tag{6}$$

Recall that $Q^* = \frac{1}{1-P_{suc}(R)}$. Here, we assume for convenience that $\log_{Q^*}(\log_{Q^*}(n)) \geq 0$, but otherwise we can replace it by 0, and the ratio holds. Plugging the bound of Equation (6) back into Equation (4) leads to Equation (3). □

We provide an informal simplification of the bound in Equation (3). Under reasonable settings, we have $C, D, R \ll \mu_{ind}$, and the probability of success P_{suc} of each segment is pretty high, hence $Q^* > e$. For this reason, we have (i) $\forall x, \log_{Q^*}(x) < \ln(x)$, (ii) $\frac{C+R+D}{W_{ME}} \approx 0$, (iii) $\frac{\ln(Q^*)}{12N_{ME}} \leq 1$, and (iv) $\frac{1}{\ln(Q^*)N_{ME}} \approx 0$. Altogether, the bound simplifies to

$$r_{id}^{ME}(n,\mathcal{T}) \leq \frac{\ln(n)}{N_{ME}} + \ln(\ln(n)) + 3 + o(1). \tag{7}$$

Here is a more precise statement (proof in Section D of the WSM):

Proposition 1. *We have* $r_{id}^{ME}(n,\mathcal{T}) \leq \frac{4}{5}(\frac{\ln(n)}{N_{ME}} + \ln(\ln(n))) + 3 + \frac{3}{N_{ME}} + o(1)$ *under the following assumptions:*

- *A checkpoint of length C succeeds with probability at least* 0.99*;*
- $D \leq R \leq C$*;*
- *A segment of length* W_{ME} *fails with probability at least* 10^{-10}*;*
- $T_{base} > 2(C + R + D)$ *(otherwise the tasks are so small that no checkpoints are needed).*

3.1.2 Tightness of the Bound $r_{id}^{ME}(n,\mathcal{T})$ *of Theorem 1.* Consider a set $\mathcal{T}$ of n identical uniprocessor tasks with $T_{base} = 2K - 1$, $C = 1$, $D = R = 0$, and $\lambda = \frac{\ln(1+\frac{1}{2K})}{2K}$ so that $e^{-\lambda(T_{base}+C)} = \frac{2K}{2K+1}$. Here, $K \geq 2$ is fixed, and n is the variable. We assume that all tasks execute in parallel, i.e., $m \geq n$. Under these settings, we show in Section H.1 of the WSM that $r_{id}^{ME}(n,\mathcal{T}) = \Theta(\ln(n))$, thereby showing the tightness of the bound given in Theorem 1.

3.1.3 Arbitrary Independent Tasks. We now proceed with different independent tasks that can be executed concurrently:

Theorem 2. *Consider a set* $\mathcal{T}$ *of n tasks. The* i*th task has profile* $\mathcal{T}_i = (T_{base}^i, p_i, C_i, R_i)$*. These tasks execute concurrently, hence* $\sum_{i=1}^{n} p_i \leq m$*. The individual fault rate on each processor is* λ*. The downtime is D. For the* MinExp *strategy,* N_{ME}^i *is the number of checkpoints, and* W_{ME}^i *is the length of each segment, for task i. Let* $P_{suc}^i(R_i) = e^{-p_i\lambda(W_{ME}^i + C_i + R_i)}$ *be the probability of success of a segment of task i with re-execution cost* R_i*, and* $Q_i^* = \frac{1}{1-P_{suc}^i(R_i)}$*. Then, the* MinExp *strategy is a* $r^{ME}(n,\mathcal{T})$*-approximation of the failure-free execution time, hence also of the optimal expected execution time, where*

$$r^{ME}(n,\mathcal{T}) \leq 2 \max_{1\leq i\leq n} \left(r_{id}^{ME}(n,\mathcal{T}_i)\right). \tag{8}$$

The key element of the (very long) proof of Theorem 2 is an important new result (to the best of our knowledge) on expectations of RVs. Please refer to Section E of the WSM. Similarly to identical tasks, under reasonable assumptions, we derive a simplified bound:

$$r^{ME}(n,\mathcal{T}) \leq 2\frac{\ln(n)}{\min_{1\leq i\leq n}(N_{ME}^i)} + 2\ln(\ln(n)) + 6 + o(1).$$

3.2 MinExp for Workflows

We proceed to the study of MinExp for a workflow of tasks, with task dependencies. We build upon the results for identical tasks (see Equation (3)) that can be reused for each task of the workflow.

Theorem 3. *Let* $\mathbb{S}$ *be a failure-free schedule of a workflow* $\mathcal{W}$ *of n tasks. The* i*th task has profile* $\mathcal{T}_i = (T_{base}^i, p_i, C_i, R_i)$*. The individual fault rate on each processor is* λ*. The downtime is D. Let* Δ *be the maximum number of tasks processed concurrently by the failure-free schedule* $\mathbb{S}$ *at any instant. Then, the* MinExp *strategy is a* $r^{ME}(\Delta,\mathcal{W})$*-approximation of the failure-free execution time, where*

$$r^{ME}(\Delta,\mathcal{W}) \leq 2 \max_{1\leq i\leq n} r_{id}^{ME}(\Delta,\mathcal{T}_i). \tag{9}$$

In other words, the degree of parallelism Δ of the schedule becomes the key parameter to bound the performance of the MinExp strategy rather than the total number n of tasks in the workflow. Similarly to independent tasks, under reasonable assumptions, we derive a simplified bound:

$$r^{ME}(\Delta,\mathcal{W}) \leq 2\frac{\ln(\Delta)}{\min_{1\leq i\leq n}(N_{ME}^i)} + 2\ln(\ln(\Delta)) + 6 + o(1).$$

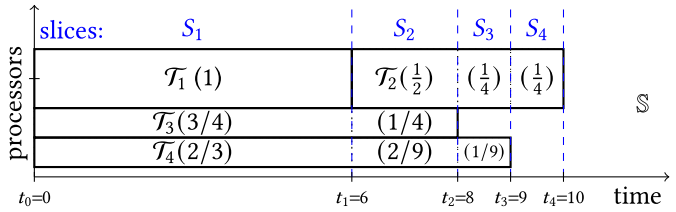

Fig. 1. Example for the proof of Theorem 3: Schedule $\mathbb{S}$.

Proof. As stated in Section 2.2, we enforce the same ordering of starting times in the initial schedule $\mathbb{S}$ and in the failure-aware schedule $\mathbb{S}$' returned by MinExp: If task i starts after task j in $\mathbb{S}$, then the same will hold in $\mathbb{S}$'. However, we greedily start a task as soon as enough processors are available, which may result in using different processors for a given task in $\mathbb{S}$ and $\mathbb{S}$'. Consider an arbitrary failure scenario, and let T_i be the execution time of task i in $\mathbb{S}$'. Let $T(\mathbb{S}')$ be the total execution time of $\mathbb{S}$'. We want to prove that

$$\mathbb{E}(T(\mathbb{S}')) \leq 2 \max_{1\leq i\leq n} r_{id}^{ME}(\Delta, \mathcal{T}_i) T(\mathbb{S}), \tag{10}$$

where $T(\mathbb{S})$ is the (deterministic) total execution time of $\mathbb{S}$.

To analyze $\mathbb{S}$', we partition $\mathbb{S}$ into a series of execution slices, where a slice is determined by two consecutive events. An event is either the starting time or the ending time of a task. Formally, let s_i be the starting time of task i in $\mathbb{S}$, and e_i be its ending time. We let $\{t_j\}_{0\leq j\leq K} = \cup_{i=1}^{n}\{s_i, e_i\}$ denote the set of events, labeled such that $\forall j \in [0, K-1], t_j < t_{j+1}$. Note that we may have $K+1 < 2n$ if two events coincide. We partition $\mathbb{S}$ into K slices S_j, $1 \leq j \leq K$, which are processed sequentially. Slice S_j spans the interval $[t_{j-1}, t_j]$. In other words, the length of S_j is $t_j - t_{j-1}$. Let $B_j \subset \mathcal{W}$ denote the subset of tasks that are (partially or totally) processed during slice S_j; note that $\Delta = \max_{j\in[1,K]} |B_j|$. Finally, for a task i in B_j, let $a_{i,j}$ be the fraction of the task that is processed during S_j (and let $a_{i,j} = 0$ if $i \notin B_j$).

As an example, we consider a workflow $\mathcal{W}$ consisting of $n = 4$ independent tasks, with $T^1_{base} = 6$, $T^2_{base} = 4$, $T^3_{base} = 8$, and $T^4_{base} = 9$. We have $m = 4$, $p_1 = p_2 = 2$, and $p_3 = p_4 = 1$. The optimal failure-free schedule $\mathbb{S}$ is shown in Figure 1 and has length 10. Note that task i is represented by its profile $\mathcal{T}_i$. There are five timesteps where an event occurs, thus $K = 4$ and $\{t_j\}_{0\leq j\leq K} = \{0, 6, 8, 9, 10\}$. Therefore, $\mathbb{S}$ is decomposed into four slices, S_1 running in $[0, 6]$, S_2 in $[6, 8]$, S_3 in $[8, 9]$, and S_4 in $[9, 10]$. The $(a_{i,j})_{i\in[1,n],j\in[1,K]}$ are represented in brackets. Finally, $B_1 = \{1, 3, 4\}$, $B_2 = \{2, 3, 4\}$, $B_3 = \{2, 4\}$, $B_4 = \{4\}$, and $\Delta = 3$. We use the decomposition into slices to define a virtual schedule $\mathbb{S}^{virt}$, which consists of scaling the slices S_j to account for failures in $\mathbb{S}$'. For each slice S_j, the scaling is the largest ratio $\frac{T_i}{T^i_{base}}$ over all tasks $i \in B_j$. Hence, $\mathbb{S}^{virt}$ is composed of K slices S^{virt}_j whose length is $T(S^{virt}_j) = (\max_{i\in B_j} \frac{T_i}{T^i_{base}})T(S_j)$. Within each slice S^{virt}_j, for each task $i \in B_j$, we execute the same fraction $a_{i,j}$ of task i as in the original schedule $\mathbb{S}$, for a duration $a_{i,j}T_i$, so that some tasks in B_j may not execute during the whole length of S^{virt}_j, contrarily to during the initial schedule $\mathbb{S}$.

We can then bound the total execution time of $\mathbb{S}^{virt}$ by using the result on independent tasks on each slice, each with a maximum degree of parallelism of Δ. Finally, to obtain Equation (10), there remains to show that $\mathbb{E}(T(\mathbb{S}')) \leq \mathbb{E}(T(\mathbb{S}^{virt}))$. In fact, we show that under any failure scenario, $T(\mathbb{S}') \leq T(\mathbb{S}^{virt})$, and the result follows. We relabel the tasks by non-decreasing starting time in $\mathbb{S}$ and prove by induction that no task starts nor ends later in $\mathbb{S}$ ′ than in $\mathbb{S}^{virt}$. The key element is that the ordering of starting times from $\mathbb{S}$ is preserved in both $\mathbb{S}^{virt}$ and $\mathbb{S}'$, see Section F of the WSM for details.

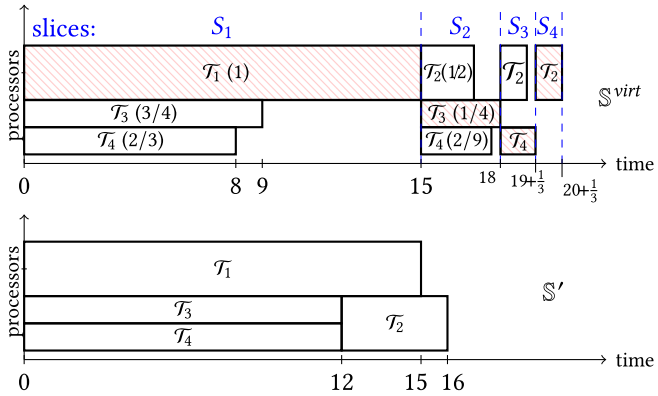

Fig. 2. Schedules $\mathbb{S}^{virt}$ (top) and $\mathbb{S}'$ (bot.) for the example.

Going back to the example, assume that $T_1 = 15$, $T_2 = 4$, and $T_3 = T_4 = 12$ in $\mathbb{S}'$. Then, we obtain the task with the largest ratio $\frac{T_i}{T_{base}^i}$ for each slice: task 1 for S_1, task 3 for S_2, task 4 for S_3, and task 2 for S_4. The schedule $\mathbb{S}^{virt}$ is shown at the top of Figure 2 and has length $T(\mathbb{S}^{virt}) = 20 + 1/3$ (and the tasks with largest ratio in each slice are hatched). Finally, the schedule $\mathbb{S}'$ is shown at the bottom of Figure 2, and $T(\mathbb{S}') = 16$. □

We point out that Theorem 3 applies to workflows with arbitrary dependencies, and with rigid or moldable tasks. The bound given for $r^{ME}(\Delta, \mathcal{W})$ is relative to the execution time of the failure-free schedule. If this failure-free schedule is itself a ρ-approximation of the optimal solution, then we have derived a $r^{ME}(\Delta, \mathcal{W}) \times \rho$ approximation of the optimal solution.

4 THE CHECKMORE STRATEGIES

The previous section has shown that, in the presence of failures, the ratio of the actual execution time of a workflow over its failure-free execution time, critically depends upon the maximum degree of parallelism Δ achieved by the initial schedule.

In this section, we introduce CheckMore strategies, which checkpoint workflow tasks more often than MinExp, with the objective to decrease the ratio above. The number of checkpoints for each task becomes a function of the degree of parallelism in the execution. We define SafeCheck(δ), the number of checkpoints for a task, given a parameter δ (typically the degree of parallelism):

Definition 2. SafeCheck(δ) partitions a parallel task of length T_{base}, with p processors and checkpoint time C, into $N_{SC}(\delta) = \lceil \frac{(\ln(\delta)+1)T_{base}}{W_{YD}} \rceil$ equal-length segments, each followed by a checkpoint, where $W_{YD} = \sqrt{\frac{2C}{p\lambda}} = \sqrt{\frac{2\mu_{ind}C}{p}}$. Each segment is of length $W_{SC}(\delta) = \frac{T_{base}}{N_{SC}(\delta)}$.

Note that MinExp corresponds to applying SafeCheck (1) to all tasks, since $N_{SC}(1) = N_{ME}$. The key building block of the analysis of MinExp is Theorem 1 for identical independent tasks. The good news is that Theorem 1 holds for any checkpointing strategy, not just for the Young/Daly approach, and can easily be extended if each task is checkpointed following SafeCheck(δ):

Theorem 4. *Consider n identical tasks of same type $\mathcal{T} = (T_{base}, p, C, R)$ to be executed concurrently on $n \times p \leq m$ processors with individual failure rate $\lambda = \frac{1}{\mu_{ind}}$. The downtime is D. For the SafeCheck(δ) strategy, $N_{SC}(\delta)$ is the number of checkpoints, and $W_{SC}(\delta)$ is the length of each segment, as given by Definition 1. Let $P_{suc}(\tilde{R}) = e^{-p\lambda(W_{SC}+C+\tilde{R})}$ be the probability of success of a segment with re-execution*

cost $\tilde{R}$ ($\tilde{R} = 0$ *if no re-execution or* $\tilde{R} = R$ *otherwise), and* $Q^* = \frac{1}{1-P_{suc}(R)}$. *Let* $r_{id}^{SC}(\delta, n, \mathcal{T}) = \frac{\mathbb{E}(T_{tot})}{T_{base}}$, *where* $\mathbb{E}(T_{tot})$ *is the expectation of the total time* T_{tot} *of the* SafeCheck(δ) *strategy. Then*

$$r_{id}^{SC}(\delta, n, \mathcal{T}) \leq \left(\frac{\log_{Q^*}(n)}{N_{SC}(\delta)} + \log_{Q^*}(\log_{Q^*}(n)) + 1 + \frac{\ln(Q^*)}{12N_{SC}(\delta)} + \frac{1}{\ln(Q^*)N_{SC}(\delta)}\right) \times \left(1 + \frac{C+R+D}{W_{SC}(\delta)}\right) + \frac{C}{W_{SC}(\delta)} + 1 + o(1). \tag{11}$$

Note that if n *is small, then the ratio holds by replacing all negative or undefined terms by* 0.

To prove this theorem, we reuse the proof of Theorem 1: We just need to replace N_{ME} by $N_{SC}(\delta)$ and W_{ME} by $W_{SC}(\delta)$.

The idea behind SafeCheck(δ) is the following: When processing δ jobs in parallel, the expected maximum number of failures given by Equation (6) eventually grows proportionally to its first term, $\log_{Q^*}(\delta)$, which is $\Theta(\ln(\delta))$. To accommodate this growth, we reduce the segment length by a factor $\ln(\delta)$, so that the total failure-induced overhead does not increase much. This is exactly what SafeCheck(δ) does, when δ tasks are processed in parallel. Similarly, the first term $\frac{\log_{Q^*}(n)}{N_{ME}(n)}$ of the ratio in Equation (3) was dominant for MinExp, while it becomes almost constant in Equation (11). To that extent, CheckMore generalizes this idea to general workflows using SafeCheck(δ) as a subroutine. We provide two variants of CheckMore as follows:

Definition 3. Consider a failure-free schedule $\mathbb{S}$ for a workflow $\mathcal{W}$ of n tasks:

- The CheckMore algorithm applies SafeCheck(Δ_i) to each task i, where Δ_i is the largest number of tasks that are executed concurrently during the processing of task i.
- The BasicCheckMore algorithm applies SafeCheck($\min(n, m)$) to all tasks, where m is the number of processors.

The main reason for introducing BasicCheckMore is that we do not need to know the maximum degree Δ of parallelism in $\mathbb{S}$ to execute BasicCheckMore (because we always have $\Delta \leq \min(n, m)$). In fact, we do not even need to know the failure-free schedule for BasicCheckMore (contrarily to CheckMore), we just need an ordered list of tasks and to greedily start them in this order.

Theorem 5. *Let* $\mathbb{S}$ *be a failure-free schedule of a workflow* $\mathcal{W}$ *of* n *tasks. The* i*th task has profile* $\mathcal{T}_i = (T_{base}^i, p_i, C_i, R_i)$. *Let* Δ_i *be the maximum number of tasks processed concurrently to task* i *by* $\mathbb{S}$ *at any instant, and let* $\Delta = \max_{1 \leq i \leq n} \Delta_i$. *Then* CheckMore *is a* $r^{CM}((\Delta_i)_{i \leq n}, \mathcal{W})$*-approximation of the failure-free execution time in expectation,*

$$r^{CM}((\Delta_i)_{i \leq n}, \mathcal{W}) \leq 2 \max_{1 \leq i \leq n} r_{id}^{SC}(\Delta_i, \Delta_i, \mathcal{T}_i). \tag{12}$$

And BasicCheckMore *is a* $r^{BCM}(\min(n, m), \mathcal{W})$*-approximation of the failure-free execution time in expectation,*

$$r^{BCM}(\min(n, m), \mathcal{W}) \leq 2 \max_{1 \leq i \leq n} r_{id}^{SC}(\min(n, m), \Delta, \mathcal{T}_i). \tag{13}$$

See Section G of the WSM for the proof. Note that for all i, $\Delta_i \leq \Delta$, and $\Delta_i \leq min(n, m)$, so it is extremely likely that the bound obtained for r^{CM} is smaller than the one obtained for r^{BCM}. To illustrate the difference between the bounds of CheckMore and MinExp, we show in Section I of the WSM that for a shelf of n identical uni-processor tasks running in parallel, r_{id}^{CM} is an order of magnitude lower than r_{id}^{ME} under reasonable assumptions and when n is large enough.

We conclude this section by returning to the example of Section 3.1.2 and showing that CheckMore (equivalent to BasicCheckMore in this case) can be arbitrarily better than MinExp. The proof is given in Section H.2 of the WSM.

Proposition 2. *Consider a set $\mathcal{T}$ of $n(K)$ identical uni-processor tasks with type $\mathcal{T} = (2K - 1, 1, 10)$, $D = 0$, and $\lambda(K) = \frac{\ln(1+\frac{1}{2K})}{2K}$. We assume that all tasks execute in parallel, i.e., $m \geq n(K)$. When letting $n(K) = \lfloor e^{\sqrt{2/\lambda(K)-1}} \rfloor$ (hence $\ln(n(K)) = \Theta(K)$), and K tending to infinity, we have $r^{ME}(n(K), \mathcal{T}) = \Theta(\frac{K}{\ln(K)})$ and $r^{BCM}(n(K), \mathcal{T}) = \Theta(1)$.*

5 EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of the different checkpointing strategies through simulations. We describe the simulation setup in Section 5.1, present the main performance comparison results in Section 5.2, and assess the impact of different parameters on the performance in Section 5.3. We further provide some performance statistics in Section 5.4 and conclude with a brief summary in Section 5.5. Our in-house simulator is written in C++ and is publicly available for reproducibility purpose [1].

5.1 Simulation Setup

We evaluate and compare the performance of the three checkpointing strategies MinExp, CheckMore, and BasicCheckMore. All strategies are coupled with a failure-free schedule computed by a list scheduling algorithm (see below). The workflows used for evaluation are generated from WorkflowHub [19] (formerly Pegasus [36]), which offers realistic synthetic workflow traces with a variety of characteristics. They have been shown to accurately resemble the ones from real-world workflow executions [3, 19]. Specifically, we generate the following nine types of workflows offered by WorkflowHub that model applications in various scientific domains: Blast, Bwa, Epigenomics, Genome, SoyKB, and Sras are bioinformatics workflows; Cycles is an agroecosystem workflow; Montage is an astronomy workflow; and Seismology is a seismology workflow; see Section J.1 of the WSM for more details.

Each trace defines the general structure of the workflow, whose number of tasks and total execution time can be specified by the user.[2] All tasks generated in WorkflowHub are uni-processor tasks. In the experiments, we evaluate the checkpointing strategies under the following parameter settings:

- Number of processors: $m = 2^{14} = 16{,}384$;
- Checkpoint/recovery/downtime: $C = R = 1$ min, $D = 0$;
- MTBF of individual processor: $\mu_{ind} = 10$ years;
- Number of tasks of each workflow: $n \approx 50{,}000$.

Furthermore, the total failure-free execution times of all workflows are generated such that they complete in 3–5 days. This is typical of the large scientific workflows that often take days to complete as observed in some production log traces [2, 33] (Section J.2 of the WSM also presents similar experimental results, which utilize small workflow traces that take less than a day to complete). Section 5.2 will present the comparison results of different checkpointing strategies under the above parameter settings. In Section 5.3, we will further evaluate the impact of different parameters (i.e., m, C, μ_{ind}, and n) on the performance.

The evaluation methodology is as follows: For each set of parameters and each type of workflow trace, we generate 30 different workflow instances and compute their failure-free schedules. We use the list scheduling algorithm that orders the tasks using the **Longest Processing Time (LPT)** first policy: If several tasks are ready and there is at least one processor available, then the longest

[2]Note that the workflow generator may offer a different number of tasks so as to guarantee the structure of the workflow. The difference, however, is usually small.

ready task is assigned to the available processor to execute. Since all tasks are uni-processor tasks, LPT is known to be a 2-approximation algorithm [22]; also LPT is known to be a good heuristic for ordering the tasks [30]. This order of execution will be enforced by all the checkpointing strategies. For each workflow instance, we further generate 50 different failure scenarios. Here, a failure scenario consists of injecting random failures to the tasks by following the Exponential distribution as described in Section 2.1. The same failure scenario will then be applied to each checkpointing strategy to evaluate its execution time for the workflow. We finally compute the *ratio* of a checkpointing strategy under a particular failure scenario as $\frac{T}{T_{base}}$, where T_{base} is the failure-free execution time of the workflow and T is the execution time under the failure scenario. The statistics of these $30 \times 50 = 1{,}500$ experiments are then compared using boxplots (that show the mean, median, and various percentiles of the ratio) for each checkpointing strategy. The boxes bound the first to the third quantiles (i.e., 25th and 75th percentiles), the whiskers show the 10th percentile to the 90th percentile, the black lines show the median, and the stars show the mean.

5.2 Performance Comparison Results

Figure 3 shows the boxplots of the three checkpointing strategies in terms of their ratios for the nine different workflows.

First, we observe that CkptMore and BasicCkptMore have very similar performance, which in most cases are indistinguishable. This shows that BasicCkptMore offers a simple yet effective solution without the need to inspect the failure-free schedule, thus making it an attractive checkpointing strategy in practice. Also, both versions of CkptMore perform significantly better and with less variation than MinExp, except for the few workflows where the ratios of all strategies are very close to 1 (e.g., Bwa, SoyKB, Sras). Overall, the 90th percentile ratio of CkptMore never exceeds 1.08, whereas that of MinExp is much higher for most workflows and reaches almost 1.5 for Montage. Similarly, the average ratio of CkptMore never exceeds 1.03, while that of MinExp is again significantly higher and reaches more than 1.2 for Seismology and Montage.

We now examine a few workflows more closely to better understand the performance. For Sras, MinExp is slightly better than CkptMore, but the ratios of all strategies are near optimal (i.e., <1.003). In this workflow, very few tasks are extremely long while many others are very short, and there are very few dependencies among them. Thus, failures hardly ever hit the long tasks due to their few number, while failures that hit short tasks have little impact on the overall execution time. This is why the ratio is so small for all strategies. It also explains why MinExp outperforms CkptMore: Although the maximum degree of parallelism is important, only a few tasks matter, and they should be checkpointed à la Young/Daly to minimize their own expected execution time and thereby that of the entire workflow. SoyKB and Bwa also have very low ratios. In the case of SoyKB, there is just not enough parallelism during the majority of the execution time, so all strategies are making reasonable checkpointing decisions, with CkptMore performing slightly better for taking into account this small parallelism. Bwa, however, has two source tasks that must be executed first and two sink tasks that must be executed last. Among them, one source task and one sink task are extremely long, so failures in other tasks have little impact (as in the case of Sras). Yet the small tasks are not totally negligible here, because the dominant sink task must be processed after all of them, so it is still worth to optimize these tasks with CkptMore, which explains why it is slightly better than MinExp.

For all the other workflows, CkptMore performs better than MinExp by a significant margin. This is due to CkptMore's more effective checkpointing strategies given the specific structure of these workflows. For instance, Montage has some key tasks that are dominant, so a failure that strikes most of the other tasks does not impact the overall execution time. This is similar to

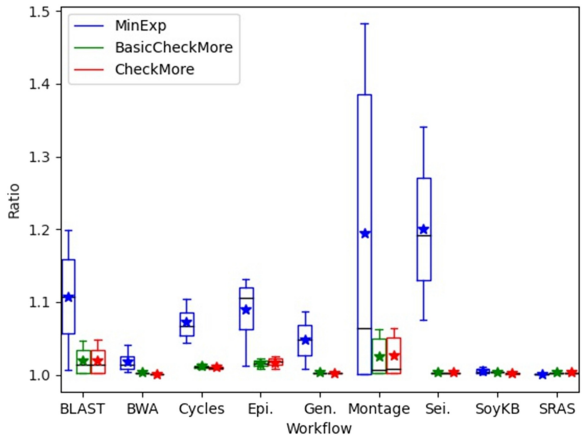

Fig. 3. Performance (ratio) comparison of the three checkpointing strategies for the nine different workflows.

the case of Sras and explains why, for all strategies, the first quantile of the ratio is very low (i.e., around 1). However, when a failure does strike one of the key tasks, the execution time will be heavily impacted. The difference with Sras is that Montage contains more key tasks that can run in parallel, so it is much more likely that one of them will fail, which is why checkpointing them with CheckMore is better. Next, Blast and Seismology have some source and sink tasks (as Bwa), which, however, are not so dominant in length, making the difference between CheckMore and MinExp higher even from the first quantile. Other workflows also have similar structures, which eventually contribute to the better performance of CheckMore over MinExp.

5.3 Impact of Different Parameters

We now study the impact of different parameters on the performance of the checkpointing strategies. In each set of experiments below, we vary a single parameter while keeping the others fixed at their base values. The results for all nine workflows are available in Section J.3 of the WSM; due to lack of space, we focus here on Blast, Seismology, Genome, and Sras only, whose results are shown in Figures 4–7. Note that the scale of the y axis is kept the same for ease of comparison. For some figures with really small values, zoomed-in plots are also provided on the original figure for better viewing.

Impact of Number of Processors (m). We first assess the impact of the number of processors, which is varied between 4,096 and 50,000, and the results are shown in Figure 4. In general, increasing the number of processors increases the ratio. This corroborates our theoretical analysis, because for most types of workflows, having more processors means having a larger Δ and thus a larger potential ratio, until m surpasses the width of the dependence graph. However, CheckMore and BasicCheckMore appear less impacted than MinExp.

For Blast, Seismology, and Genome, the ratio is very close to 1 when m is small for all checkpointing strategies. In fact, for these workflows, most tasks are quite independent. Thus, when n is large compared to m, even if a failure strikes a task, it will have little impact on the starting times of the other tasks. This is because we only maintain the order of execution but do not stick to the same mapping as in the failure-free schedule. For this reason, it is better to minimize each task's own execution time by using MinExp (i.e., CheckMore checkpoints a bit too much). However, when m becomes large, the performance of MinExp degrades significantly, with an average ratio even reaching 1.7 for Blast at $m = 50{,}000$, whereas it stays below 1.1 for CheckMore.

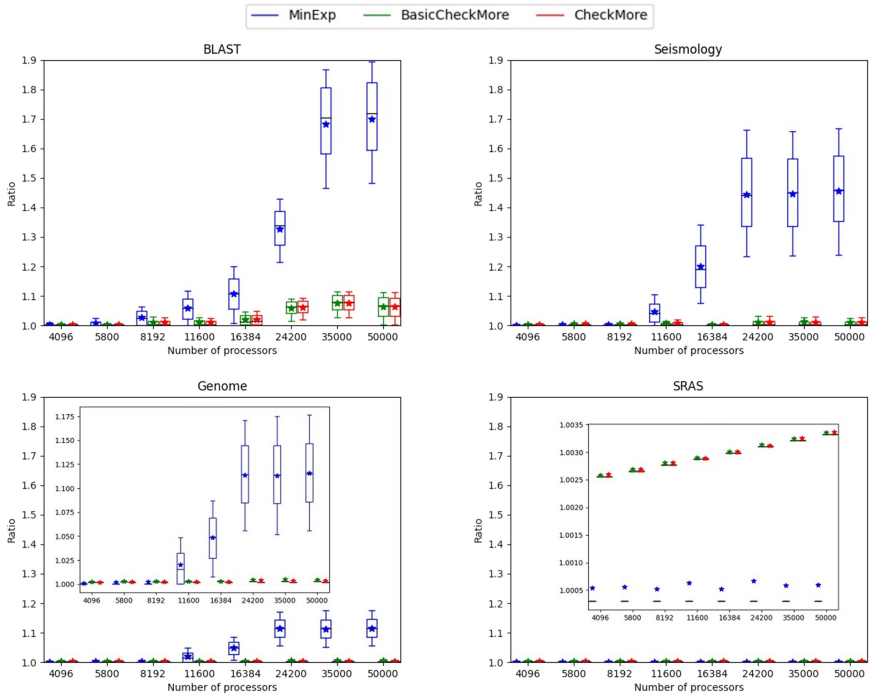

Fig. 4. Impact of number of processors (m) on the performance of the checkpointing strategies for four different workflows (Blast, Seismology, Genome, and Sras).

Finally, for Sras, as the number of dominating tasks that could be run in parallel is way less than 4,096, the ratio of MinExp does not vary much with m, while that of CheckMore increases with m as it tends to checkpoint more with an increasing number of processors. Also, in more than 90% of the cases, the failures have strictly no impact on the overall execution time, since they do not hit the dominating tasks. This is why the average ratio is above the 90th percentile for all checkpointing strategies.

Impact of Checkpoint Time (C). We now evaluate the impact of the checkpoint time by varying it between 15 and 240 seconds, and the results are shown in Figure 5. The ratio generally increases with C; this is consistent with Equation (3). When $R = C$ and $D = 0$, the approximation ratio satisfies $r \leq (\frac{X}{N_c} + Y)(\frac{2C}{W} + 1) + \frac{C}{W} + Z$, where X, Y, and Z barely depend on C, N_c decreases with C, and $\frac{C}{W} \approx \sqrt{\frac{C}{2\mu}}$ increases with C. Intuitively, the checkpoint time impacts the ratio in two ways. First, as C increases, we pay more for each checkpoint, which could lead to an increased ratio. Second, as we use $W_{YD} = \sqrt{\frac{2C}{p\lambda}}$ to determine the checkpointing period and hence the number of checkpoints, a task will become less safe when C increases, because it will be checkpointed less, and this could also increase the ratio.

Looking at Genome under MinExp, we can see a clear increase in the ratio when C increases from 15 to 21. This is because the typical number of checkpoints for the critical tasks (that affect the overall execution time the most) drops from 3 to 2, thus the time wasted due to a failure increases from 33% to 50%. As C increases from 60 to 85, the typical number of checkpoints of these tasks further drops from 2 to 1, making the waste per failure increase to 100%, and so the ratio also greatly increases. For values of C between 21 and 42, even if the number of checkpoints

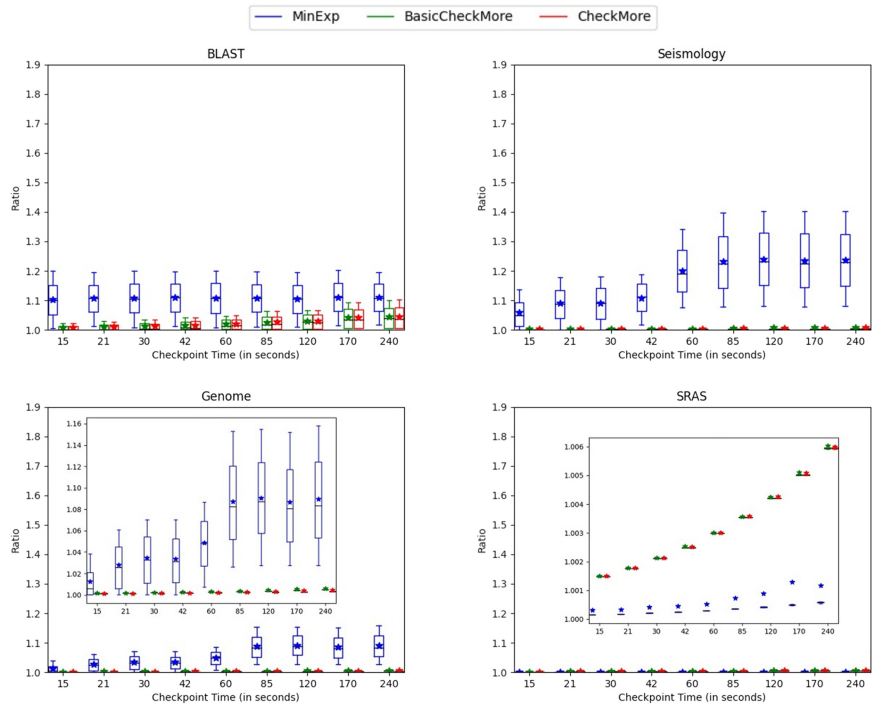

Fig. 5. Impact of checkpoint time (C) on the performance of the checkpointing strategies for four different workflows (Blast, Seismology, Genome, and Sras).

does not change, the ratio increases smoothly due to the increase in checkpoint time. The ratio of CheckMore, however, only increases slightly with the checkpoint time, which is, however, not visible in the figure due to the small values. Seismology also clearly illustrates these phenomena. For Sras, since most failures do not affect the overall execution time, the ratio of all strategies is only impacted by the checkpoint time. For Blast under MinExp, because most tasks are short and we have a single checkpoint to start with, the increase in checkpoint time is negligible compared to the waste induced by failures.

Impact of Individual MTBF (μ_{ind}). We evaluate the impact of individual processor's MTBF by varying it between 30 months and 40 years, and the results are shown in Figure 6. Intuitively, when μ_{ind} increases (or, equivalently, the failure rate λ decreases), we would have fewer failures and expect the ratio to decrease. This is generally true for CheckMore but not always for MinExp. To understand why, we refer again to the simplified approximation ratio $r \leq (\frac{X}{N_c} + Y)(\frac{2C}{W} + 1) + \frac{C}{W} + Z$, where X, Y, and Z are barely affected by μ_{ind}. Here, when the number of failures decreases, $W_{YD} = \sqrt{\frac{2C}{p\lambda}}$ increases, so the number of checkpoints decreases and the time wasted for each failure increases. This could potentially lead to an increase in the ratio. To illustrate this compound effect, we again look at Genome under MinExp. When μ_{ind} goes from 2.5 to 3.5 years, the typical number of checkpoints for the critical tasks (that affect the overall execution time the most) drops from 3 to 2, which increases the waste per failure by around 50%. This together with the fact that MinExp does not take into account the parallelism results in an increase in the ratio. When μ_{ind} goes from 3.5 to 7 years, the ratio decreases simply because we have fewer failures. As μ_{ind} continues to increase to 14 years, the number of checkpoints for the critical tasks further drops from 2 to 1.

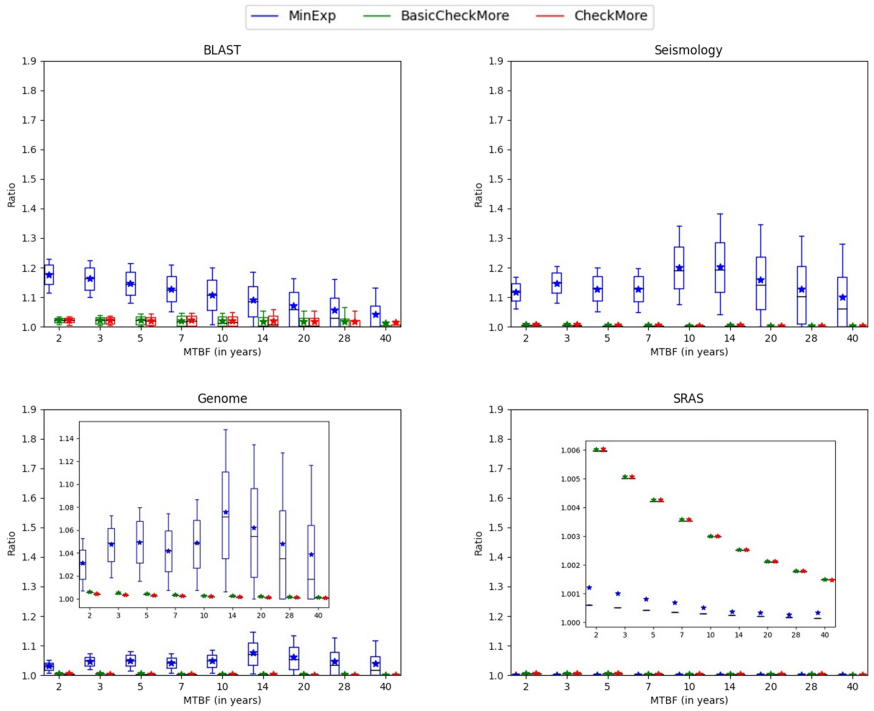

Fig. 6. Impact of individual MTBF (μ_{ind}) on the performance of the checkpointing strategies for four different workflows (Blast, Seismology, Genome, and Sras).

This increases the waste per failure to 100%, which again leads to an increase in the ratio. From this point on, the ratio will just decrease with μ_{ind}, again due to fewer failures. The same can be observed for Seismology. In Blast and Sras, the ratio simply decreases with μ_{ind}. For Blast, even when μ_{ind} is small, we only checkpoint once, so the ratio decreases due to fewer failures. For Sras, failures usually do not impact the overall execution time, so the decrease in ratio is mainly due to the decrease in the number of checkpoints.

Finally, it is worth noting that the ratio variance increases as μ_{ind} increases. This is because when there are only a few failures and the length of the segments is large, the failure location (inside the segments) will matter significantly, especially for MinExp.

Impact of Number of Tasks (n). Finally, we study the impact of the number of tasks in the workflow, which is varied between 8,800 and 70,000, and the results are shown in Figure 7. Again, the ratio is impacted by the number of tasks in two different ways. First, when n increases, the width of the graph increases and so does Δ, and this would increase the ratio according to our analysis. Second, when n increases and m is fixed, the average number of tasks executed by each processor increases. This means that if a failure occurs early in the execution, then it is less likely to have a significant impact on the ratio, since multiple other tasks will be processed afterwards to balance the load, especially if the tasks are relatively independent.

These two phenomena are clearly observed in Blast under MinExp. This workflow mainly consists of a large batch of independent tasks. When n increases to 17,680, which is approximately the number of processors ($m = 16{,}384$), the ratio increases because Δ increases. After that, the ratio starts to decrease because $n > m$. In this case, when a failure strikes an early task, the subsequent tasks could be assigned to other processors to reduce the impact of the failure. Ultimately, if $n \gg m$,

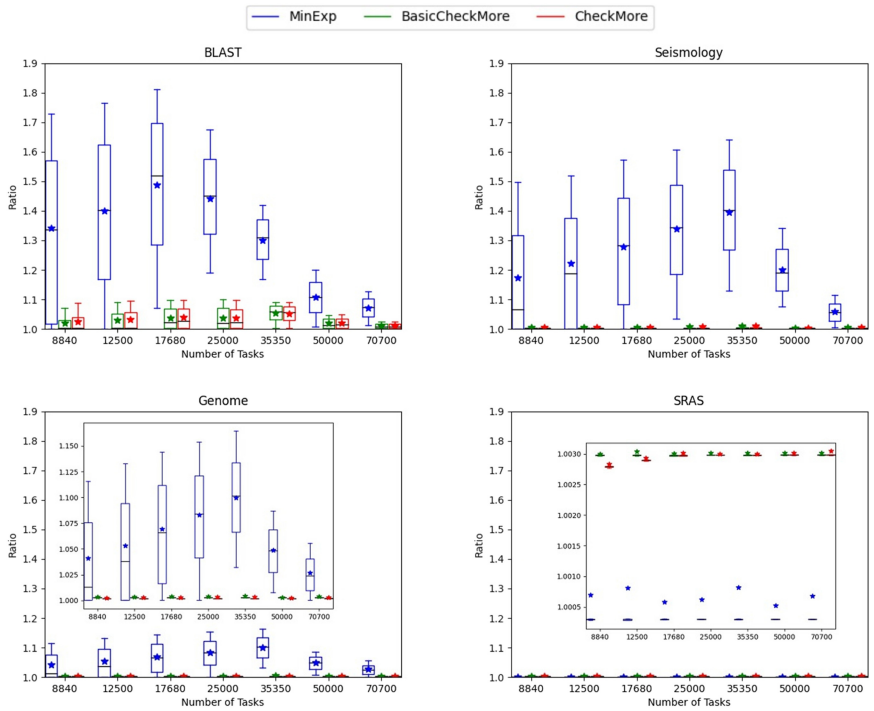

Fig. 7. Impact of number of tasks (n) on the performance of the checkpointing strategies for four different workflows (Blast, Seismology, Genome, and Sras).

then MinExp would become more efficient. Indeed, since the tasks are almost independent and uni-processor tasks, list scheduling is able to dynamically balance the loads of different processors. Thus, minimizing the expected execution time of each individual task using MinExp would be a good strategy for the overall execution time of the workflow.

For Seismology and Genome, we observe the same up-and-down effect as a result of these two phenomena, but nor for Sras, which is not impacted by the number of tasks. For this workflow, only a few key dominating tasks matter and their width remains well below the number of processors. Since these tasks form a small proportion of the total number of tasks, varying n does not significantly alter their chance of being hit by a failure, so the ratio remains close to 1.

5.4 Statistics

We provide some statistics related to the experiments of Section 5.3, still focusing on the four workflows Blast, Seismology, Genome, and Sras. First, we check the quality of the strategy MinExp for each task, hence with N_{ME} segments: We make a comparison with the strategy Lambert that uses the exact optimal number of segments N_{opt} (with the notations of Section 2.5). In Table 2, we report the mean and standard deviation of the ratio of the expected execution time achieved by Lambert over that achieved by MinExp. Hence, a value greater than 1 means that MinExp is better. Because we consider statistics on ratios, we use the geometric mean and standard deviation instead of classical values. For each workflow type, an instance in Table 2 corresponds to a given set of parameters (of 30 possible sets), which is tested for 30 different task graphs. Hence there are 900 instances per workflow type. For each instance, the expected execution times are averaged over 50 failure scenarios.

Table 2. Geometric Mean and Standard Deviation of the Ratio of the Expected Execution Time Achieved by Lambert over That Achieved by MinExp

	Blast	Genome	Seismology	Sras
Geometric Mean	1.00017	1.02691	1.02895	1.000041
Geometric SD	1.01564	1.02486	1.05626	1.00077
Instances where MinExp is better	462(51.3%)	778(86.4%)	677(75.2%)	405(45%)
Instances where Lambert is better	436(48.4%)	122(13.6%)	220(24.4%)	444(49.3%)
Instances where both are identical	2(0.2%)	0(0%)	3(0.3%)	51(5.7%)
# tasks with identical number of checkpoints (often 1)	95.5%	93.1%	94.2%	91.5%
# tasks where Lambert has one less checkpoint	4.5%	6.9%	5.8%	8.5%
# tasks where Lambert has neither of above	0.0%	0.0%	0.0%	0.0%

Table 3. Average Performance Ratio of Each Checkpoint Strategy

	Blast	Genome	Seismology	Sras
MinExp	1.19228	1.08755	1.19537	1.000737
Lambert	1.19256	1.11702	1.23072	1.000778
BasicCheckMore	1.02758	1.00688	1.00714	1.00329
CheckMore	1.02723	1.00698	1.00717	1.00330

In the last three rows of Table 2, we compare the number of checkpoints for each task in each graph, with a total of around 25,000,000 task comparisons per workflow type. The 0.0% is exactly 0 of around 25 million: We never found a task for which Lambert would not have either the same number of checkpoints as MinExp or one less checkpoint than MinExp. Altogether, we conclude that MinExp and Lambert perform almost the same. The slight superiority of MinExp in terms of performance is due to its conservative approach: MinExp rounds up the number of checkpoints of each task to the higher number (taking the ceiling instead of the floor), which turns out to be a good decision when several tasks execute in parallel.

Next, in Table 3, for each of the four workflow types, we report the average value, over all 900 instances, of the performance ratio of each checkpointing strategy. As in Section 5.2, the performance ratio is $\frac{T}{T_{base}}$, where T_{base} is the failure-free execution time of the workflow and T is the execution time under the failure scenario. The major difference from the results of Section 5.3 is that we now add the Lambert strategy to the comparison. Clearly, MinExp and Lambert are quite similar, while CheckMore and BasicCheckMore bring huge benefits, except for Sras.

5.5 Summary

Our experimental evaluation demonstrates that MinExp and Lambert are not resilient enough for checkpointing workflows, although they provide an optimal strategy for each individual task. However, CheckMore proves to be a very useful strategy, except for Sras whose ratios are extremely low. When varying the key parameters, the simulation results nicely corroborate our theoretical analysis. Furthermore, the easy-to-implement BasicCheckMore strategy always leads to ratios that are close to those of CheckMore, regardless of the parameters.

6 RELATED WORK

6.1 Scheduling Workflows

Scheduling a computational workflow consisting of a set of tasks in a dependency graph to minimize the overall execution time (or makespan) is a well-known NP-complete problem [20]. Only a few special cases are known to be solvable in polynomial time, such as when all tasks are of the

same length and the dependency graph is a tree [27] or when there are only two processors [11]. For the general case, some branch-and-bound algorithms [26, 35] have been proposed to compute the optimal solution, but the problem remains tractable only for small instances. In the seminal work, Graham [22] showed that the list scheduling strategy, which organizes all tasks in a list and schedules the first ready task at the earliest time possible, achieves an execution time that is no worse than $2 - \frac{1}{m}$ times the optimum, where m denotes the total number of processors, i.e., the algorithm is a $(2-\frac{1}{m})$-approximation. This performance guarantee holds regardless of the order of the tasks in the list. Some heuristics further explore the impact of different task orderings on the overall execution time, with typical examples including task execution times, bottom-levels, and critical paths (see Reference [30] for a comprehensive survey of the various heuristic strategies).

While the results above are for workflows with uni-processor tasks (or tasks that share the same degree of parallelism), scheduling workflows with parallel tasks has also been considered. Li [32] proved that, for precedence constrained tasks with fixed parallelism of different degrees (i.e., rigid tasks), the worst-case approximation ratio for list scheduling under a variety of task ordering rules is m. However, if all tasks require no more than qm processors, where $0 < q < 1$, then the approximation ratio becomes $\frac{(2-q)m}{(1-q)m+1}$. Demirci et al. [13] proved an $O(\log n)$-approximation algorithm for this problem using divide-and-conquer, where n is the number of tasks in the workflow. Furthermore, for parallel tasks that can be executed using a variable number of processors at launch time (i.e., moldable tasks), list scheduling is shown to be an $O(1)$-approximation when coupled with a good processor allocation strategy under reasonable assumptions on the tasks' speedup profiles [17, 29, 31].

In this article, we augment the workflow scheduling problem with the checkpointing problem for its constituent tasks. We analyze the approximation ratios of some checkpointing strategies while relying on the ratios of existing scheduling algorithms to provide an overall performance guarantee for the combined problem.

6.2 Checkpointing Workflows

Checkpoint-restart is one of the most widely used strategy to deal with fail-stop errors. Several variants of this policy have been studied; see Reference [25] for an overview. The natural strategy is to checkpoint periodically, and one must decide how often to checkpoint, i.e., derive the optimal checkpointing period. An optimal strategy is defined as a strategy that minimizes the expectation of the execution time of the application. For an preemptible application, given the checkpointing cost C and platform MTBF μ, the classical formula due to Young [38] and Daly [12] states that the optimal checkpointing period is $W_{YD} = \sqrt{2\mu C}$.

Going beyond preemptible applications, some works have studied task-based applications, using a model where checkpointing is only possible right after the completion of a task. The problem is then to determine which tasks should be checkpointed. This problem has been solved for linear workflows (where the task graph is a simple linear chain) by Toueg and Babaoglu [37], using a dynamic programming algorithm. This algorithm was later extended in Reference [5] to cope with both fail-stop and silent errors simultaneously. Another special case is that of a workflow whose dependence graph is arbitrary but whose tasks are parallel tasks that each executes on the whole platform. In other words, the tasks have to be serialized. The problem of ordering the tasks and placing checkpoints is proven NP-complete for simple join graphs in Reference [4], which also introduces several heuristics. Finally, for general workflows, deciding which tasks to checkpoint has been shown #P-complete [23], but several heuristics are proposed in Reference [24].

In this article, we depart from the above model [5, 23, 24, 37] and assume that each workflow task is a preemptible task that can be checkpointed at any instant. This assumption is quite natural for many applications, such as those involving dense linear algebra kernels or tensor operations.

It is even mandatory for coarse-grained workflows: Unless the failure rate can be decreased below the current standard, the successful completion of any large task, say, executing a few hours with 1K nodes, is very unlikely.

7 CONCLUSION

In this article, we have investigated checkpointing strategies for parallel workflows, whose tasks are either sequential or parallel and in the latter case either rigid or moldable. Because HPC tasks may have a large granularity, we assume that they can be checkpointed at any instant. Starting from a failure-free schedule, the natural MinExp strategy consists in checkpointing each task so as to minimize its expected execution time; hence MinExp builds upon the classical results of Young/Daly and uses the optimal checkpointing period for each task. We derive a performance bound for MinExp and exhibit an example where this bound is tight.

Intuitively, MinExp may perform badly in some cases, because there is an important risk that the delay of one single task will slow down the whole workflow. To mitigate this risk, we introduce CheckMore strategies that may checkpoint some tasks more often than other tasks and more often than in the MinExp strategy. This comes in two flavors. CheckMore decides, for each task, how many checkpoints to take, building upon its degree of parallelism in the corresponding failure-free schedule. BasicCheckMore is just using, as degree of parallelism for each task, the maximum possible value $\min(n, m)$ (hence it is equivalent to CheckMore for independent tasks all running in parallel). The theoretical bounds for BasicCheckMore are not as good as those of CheckMore, but its performance in practice is very close, and thus BasicCheckMore proves to be very efficient despite its simplicity.

An extensive set of simulations is conducted at large scale, using realistic synthetic workflows from WorkflowHub with between 8k and 70k tasks, and running on a platform with up to 50k processors. The results are impressive, with ratios very close to 1 on all workflows for both CheckMore strategies, while MinExp has much higher ratios, for instance 1.7 on average for Blast and 1.46 for Seismology. Hence, the simulations confirm that it is indeed necessary in practice to checkpoint workflow tasks more often than the classical Young/Daly strategy.

As future work, we plan to extend the simulation campaign to parallel tasks (rigid or moldable), as soon as workflow benchmarks with parallel tasks are available to the community. We will also investigate the impact of the failure-free list schedule on the final performance in a failure-prone execution, both theoretically and experimentally. Indeed, list schedules that control the degree of parallelism in the execution may provide a good tradeoff between efficiency (in a failure-free framework) and robustness (when many failures strike during execution).

ACKNOWLEDGMENTS

We thank Julien Moatti for his help in Section C of the WSM. We also thank the reviewers for their comments and suggestions, which greatly helped improve the final version of the article.

REFERENCES

[1] Anne Benoit, Lucas Perotin, Yves Robert, and Hongyang Sun. 2021. Checkpointing Workflows à la Young/Daly Is Not Good Enough: Code for In-house Simulator. (June 2021). https://graal.ens-lyon.fr/~yrobert/simulator.zip.

[2] Argonne Leadership Computing Facility (ALCF). Mira Log Traces. Retrieved from https://reports.alcf.anl.gov/data/mira.html.

[3] Malcolm Atkinson, Sandra Gesing, Johan Montagnat, and Ian Taylor. 2017. Scientific workflows: Past, present and future. *Fut. Gener. Comput. Syst.* 75 (2017), 216–227.

[4] Guillaume Aupy, Anne Benoit, Henri Casanova, and Yves Robert. 2016. Scheduling computational workflows on failure-prone platforms. *Int. J. Netw. Comput.* 6, 1 (2016), 2–26.

[5] Anne Benoit, Aurélien Cavelan, Yves Robert, and Hongyang Sun. 2016. Assessing general-purpose algorithms to cope with fail-stop and silent errors. *ACM Trans. Parallel Comput.* 3, 2 (2016).
[6] Wesley Bland, Aurelien Bouteiller, Thomas Herault, Joshua Hursey, George Bosilca, and Jack J. Dongarra. 2013. An evaluation of User-Level Failure Mitigation support in MPI. *Computing* 95, 12 (2013), 1171–1184.
[7] Marin Bougeret, Henri Casanova, Mikael Rabie, Yves Robert, and Frédéric Vivien. 2011. *Checkpointing Strategies for Parallel Jobs*. Research Report 7520. INRIA, France.
[8] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. 2014. Toward exascale resilience: 2014 update. *Supercomput. Front. Innov.* 1, 1 (2014).
[9] F. Cappello, K. Mohror, et al. 2019. VeloC: Very Low Overhead Checkpointing System. Retrieved from https://veloc.readthedocs.io/en/latest/.
[10] K. M. Chandy and L. Lamport. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.* 3, 1 (1985), 63–75.
[11] E. G. Coffman and R. L. Graham. 1972. Optimal scheduling for two-processor systems. *Acta Inf.* 1, 3 (1972), 200–213.
[12] J. T. Daly. 2006. A higher order estimate of the optimum checkpoint interval for restart dumps. *Fut. Gener. Comput. Syst.* 22, 3 (2006), 303–312.
[13] Gökalp Demirci, Henry Hoffmann, and David H. K. Kim. 2018. Approximation algorithms for scheduling with resource and precedence constraints. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS'18)*. 25:1–25:14.
[14] Nosayba El-Sayed and Bianca Schroeder. 2013. Reading between the lines of failure logs: Understanding how HPC systems fail. In *Proceedings of the 43rd International Conference on Dependable Systems and Networks (DSN'13)*. IEEE, 1–12.
[15] Fault-Tolerance Research Hub. 2021. User Level Failure Mitigation. Retrieved from https://fault-tolerance.org.
[16] Dror G. Feitelson and Larry Rudolph. 1996. Toward convergence in job schedulers for parallel supercomputers. In *Job Scheduling Strategies for Parallel Processing*. Springer, 1–26.
[17] Anja Feldmann, Ming-Yang Kao, Jiří Sgall, and Shang-Hua Teng. 1998. Optimal on-line scheduling of parallel jobs with dependencies. *J. Combin. Optim.* 1, 4 (1998), 393–411.
[18] K. Ferreira, J. Stearley, J. H. III Laros, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. 2011. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*. ACM.
[19] Rafael Ferreira da Silva, Loïc Pottier, Tainã Coleman, Ewa Deelman, and Henri Casanova. 2020. WorkflowHub: Community framework for enabling scientific workflow research and development. In *Proceedings of the IEEE/ACM Workflows in Support of Large-Scale Science (WORKS'20)*. 49–56. https://doi.org/10.1109/WORKS51914.2020.00012
[20] M. R. Garey and D. S. Johnson. 1979. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman & Company.
[21] Pter J. Grabner and Helmut Prodinger. 1997. Maximum statistics of N random variables distributed by the negative binomial distribution. *Combin. Probab. Comput.* 6, 2 (1997), 179–183.
[22] R. L. Graham. 1969. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.* 17, 2 (1969), 416–429.
[23] Li Han, Louis-Claude Canon, Henri Casanova, Yves Robert, and Frédéric Vivien. 2018. Checkpointing workflows for fail-stop errors. *IEEE Trans. Comput.* 67, 8 (2018), 1105–1120.
[24] Li Han, Valentin Le Fèvre, Louis-Claude Canon, Yves Robert, and Frédéric Vivien. 2018. A generic approach to scheduling and checkpointing workflows. In *Proceedings of the 47th Int. Conf. on Parallel Processing (ICPP'18)*. IEEE Computer Society Press.
[25] Thomas Herault and Yves Robert (Eds.). 2015. *Fault-Tolerance Techniques for High-Performance Computing*. Springer Verlag.
[26] Udo Hönig and Wolfram Schiffmann. 2003. A parallel branch-and-bound algorithm for computing optimal task graph schedules. In *Proceedings of the 2nd International Workshop on Grid and Cooperative Computing (GCC'03)*. 18–25.
[27] T. C. Hu. 1961. Parallel sequencing and assembly line problems. *Operat. Res.* 9, 6 (1961), 841–848.
[28] IBM Spectrum LSF Job Scheduler. 2021. Fault Tolerance and Automatic Management Host Failover. Retrieved from https://www.ibm.com/docs/en/spectrum-lsf/10.1.0?topic=cluster-fault-tolerance.
[29] Klaus Jansen and Hu Zhang. 2006. An approximation algorithm for scheduling malleable tasks under general precedence constraints. *ACM Trans. Algor.* 2, 3 (2006), 416–434.
[30] Yu-Kwong Kwok and Ishfaq Ahmad. 1999. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.* 31, 4 (1999), 406–471.
[31] Renaud Lepère, Denis Trystram, and Gerhard J. Woeginger. 2001. Approximation algorithms for scheduling malleable tasks under precedence constraints. In *Proceedings of the European Symposium on Algorithms (ESA'01)*. 146–157.
[32] Keqin Li. 1999. Analysis of the list scheduling algorithm for precedence constrained parallel tasks. *J. Combin. Optim.* 3, 1 (1999), 73–88.

[33] National Energy Research Scientific Computing Center (NERSC). Cori Log Traces. Retrieved from https://docs.nersc.gov/systems/cori/.
[34] B. Schroeder and G. A. Gibson. 2007. Understanding failures in petascale computers. *J. Phys.: Conf. Ser.* 78, 1 (2007).
[35] Ahmed Zaki Semar Shahul and Oliver Sinnen. 2010. Scheduling task graphs optimally with A*. *J. Supercomput.* 51 (2010), 310–332.
[36] Pegasus Team. 2014. Pegasus Workflow Generator. Retrieved from https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator.
[37] Sam Toueg and Özalp Babaoğlu. 1984. On the optimum checkpoint selection problem. *SIAM J. Comput.* 13, 3 (1984).
[38] John W. Young. 1974. A first order approximation to the optimum checkpoint interval. *Commun. ACM* 17, 9 (1974), 530–531.

Received 18 November 2021; revised 7 July 2022; accepted 11 July 2022