Optimal Checkpointing Strategies for Iterative Applications

Yishu Du, Loris Marchal, Guillaume Pallez, and Yves Robert, *Fellow, IEEE*

Abstract—This work provides an optimal checkpointing strategy to protect iterative applications from fail-stop errors. We consider a general framework, where the application repeats the same execution pattern by executing consecutive iterations, and where each iteration is composed of several tasks. These tasks have different execution lengths and different checkpoint costs. Assume that there are n tasks and that task a_i, where $0 \leq i < n$, has execution time t_i and checkpoint cost c_i. A naive strategy would checkpoint after each task. Another naive strategy would checkpoint at the end of each iteration. A strategy inspired by the Young/Daly formula would work for $\sqrt{2\mu c_{ave}}$ seconds, where μ is the application MTBF and c_{ave} is the average checkpoint time, and checkpoint at the end of the current task (and repeat). Another strategy, also inspired by the Young/Daly formula, would select the task $a_{\min}$ with smallest checkpoint cost $c_{\min}$ and would checkpoint after every p^{th} instance of that task, leading to a checkpointing period pT, where $T = \sum_{i=0}^{n-1} a_i$ is the time per iteration. One would choose the period so that $pT \approx \sqrt{2\mu c_{\min}}$ to obey the Young/Daly formula. All these naive and Young/Daly strategies are suboptimal. Our main contribution is to show that the optimal checkpoint strategy is globally periodic, and to design a dynamic programming algorithm that computes the optimal checkpointing pattern. This pattern may well checkpoint many different tasks, and this across many different iterations. We show through simulations, both from synthetic and real-life application scenarios, that the optimal strategy outperforms the naive and Young/Daly strategies.

Index Terms—Iterative application, checkpoint strategy, fail-stop error, resilience

1 Introduction

Deploying scientific applications at large scale requires fault-tolerant mechanisms. State-of-the-art supercomputers such as Fugaku, Summit, or Sierra (respectively ranked 1st, 2nd, and 3rd in the TOP500 ranking [1]) are now embedding millions of cores (with a peak at 10.6M for Sunway TaihuLight (4th)). These very large systems are prone to failures: even if each of their cores has a very low probability of failure, the failure probability of the whole system is much higher. More precisely, assume that the *Mean Time Between Failure* (MTBF) of each computing resource is around 10 years, which means that such a resource should experience an error only every ten years on average, and which explains why computing resources are individually very reliable. When running a simulation code on 100,000 of these resources in parallel, the MTBF is reduced to only 50 minutes [2]: on average one node of the computing platform crashes every 50 minutes. With one million of such resources, the MTBF gets as small as five minutes, while codes deployed on such extreme-scale platforms usually last for hours or days. As the demand for computing power increases, failures cannot be ignored anymore, and fault-tolerant mechanisms must be deployed.

The classical way of dealing with failures in extreme-scale computing systems consists of Checkpoint/Rollback mechanisms. A *checkpoint* of the application is taken periodically, that is, the state of the application (usually the whole content of its memory) is written onto reliable storage. Whenever one of the computing resources experiences a failure, the application pauses and restarts from the last valid checkpoint. Several studies have focused on the crucial question of the optimal checkpointing period, defined as the time between two consecutive checkpoints. On the one hand, if checkpoints are taken too often, time is wasted in costly I/O operations. On the other hand, if checkpoints are too infrequent, time will be wasted in recomputing large portions of the application after each failure. Interestingly, reliability was already a question in the early days of computing: in the 70s, Young has proposed a first-order approximation of the optimal time between two checkpoints that minimizes the expected duration of the whole computation [3]. Young's approximation has then be refined by Daly thirty years later [4]. The Young/Daly approach assumes that a checkpoint can be taken anytime during the computation, and that the time needed to take a checkpoint is constant (which corresponds to a constant size of the data to save).

When designing checkpoint/restart strategies for task-based workflows, it is natural to take checkpoint between the completion of some task and the beginning of its successor. This way, the checkpoint mechanism can be provided by the workflow management system without having to modify the code of each task. However, this restricts the

- *Yishu Du is with the Tongji University, Shanghai 200092, China, and also with the LIP, ENS Lyon, 69342 Lyon, France. E-mail: yishu.du@ens-lyon.fr.*
- *Loris Marchal is with the LIP, École Normale Supérieure de Lyon, CNRS & Inria, 69342 Lyon, France. E-mail: loris.marchal@ens-lyon.fr.*
- *Guillaume Pallez is with the Inria & Université de Bordeaux, 33405 Talence, France. E-mail: guillaume.pallez@inria.fr.*
- *Yves Robert is with the LIP, École Normale Supérieure de Lyon, CNRS & Inria, 69342 Lyon, France, and also with the University of Tennessee Knoxville, Knoxville, TN 37996 USA. E-mail: yves.robert@ens-lyon.fr.*

*Manuscript received 27 Oct. 2020; revised 3 July 2021; accepted 12 July 2021. Date of publication 26 July 2021; date of current version 5 Aug. 2021. (Corresponding author: Yves Robert.)
Recommended for acceptance by K. Mohror.
Digital Object Identifier no. 10.1109/TPDS.2021.3099440*

1045-9219 © 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: University of Glasgow. Downloaded on August 19,2021 at 17:53:12 UTC from IEEE Xplore. Restrictions apply.

time-steps at which checkpoints can be taken and makes the optimization problem of selecting the best checkpoint times more difficult. Furthermore, the data to checkpoint, is now the output of the tasks and may have different sizes for different tasks of the workflow.

In this paper, we focus on designing optimal checkpoint strategies for iterative workflows expressed as *pipelined linear workflows*: we consider workflows made of a large number of iterations, each iteration being a linear chain of parallel tasks. The typical example is an application consisting of an outer loop "`While convergence is not met, do`", and where the loop body includes a sequence of large parallel operations. The objective is to find which task outputs should be saved on stable storage in order to minimize the expected duration of the whole computation. To the best of our knowledge, this is an open problem, despite the practical importance and ubiquity of pipelined linear workflows in High-Performance Computing (HPC). Indeed, the simple case of a unique linear chain of tasks (a pipelined linear workflow with a single iteration) has been solved by Toueg *et al.* [5] using a dynamic programming algorithm. On the contrary, the problem for workflows with general directed graphs has been shown #P-complete[1] [6]. The study for pipelined linear workflows (a linear chain with several iterations) is challenging, and the main contribution of this paper is to provide a complete answer:

- We prove that there exists an optimal checkpointing strategy which is periodic. It consists in a pattern of task outputs to checkpoint, where this pattern spans over a set of iterations of bounded size. This pattern is repeated over and over throughout the execution.
- We provide a dynamic programming algorithm which is polynomial in the number of operations included in the outer loop to compute the optimal periodic checkpoint pattern. The complexity of the algorithm does not depend on the number of iterations of the outer loop. This pattern may well checkpoint many different tasks, and this across many different iterations.
- We conduct an extensive set of simulations to compare the optimal strategy to four natural competitor strategies: the first checkpoints after each task, the second after each iteration, while the last two are extensions of the Young/Daly formula for iterative applications. Our simulations with both synthetic and real-life workflow instances demonstrate that our optimal strategy provide improvement over the simpler competitors.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 presents a detailed model for the problem and states the objective function. Section 4 outlines the optimal checkpoint strategy. Section 5 reports a comprehensive set of experimental results, based upon both synthetic workflows and on workflows arising from two real-life applications. Finally, Section 6 provides concluding remarks and directions for future work.

1. #P-complete problems are at least as hard as NP-complete problems.

2 Related Work

We survey related work in this section. We start with checkpointing in Section 2.1. Then we discuss iterative applications with cyclic tasks in Section 2.2. We end with fault-tolerance methods for iterative applications in Section 2.3.

2.1 Checkpointing

Checkpoint-restart is one of the most used strategy to deal with fail-stop errors, and several variants of this policy have been studied, see [2] for an overview. The natural strategy is to checkpoint periodically, and one must decide how often to checkpoint, i.e., must 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 a divisible-load application, given the checkpointing cost C and platform MTBF μ, the classical formula due to Young [3] and Daly [4] states that the optimal checkpointing period is $P_{YD} = \sqrt{2\mu C}$.

Going beyond divisible-load applications, some works have studied linear workflows, i.e., applications that can be expressed as a linear chain of (parallel) tasks. Checkpointing is only possible right after the completion of a task, and the problem is to determine which tasks should be checkpointed. This problem has been solved by Toueg and Babaoglu [5] using a dynamic programming algorithm. We stress that this latter approach is not suited to iterative applications. Indeed, consider an iterative application with a large number of iterations, say $N_{iter} = 10,000$ iterations, and assume that $n = 10$ (10 tasks per iteration). One solution to find an optimal checkpointing strategy could be: (i) unroll the loop and build a linear chain of $n \times N_{iter} = 100,000$ tasks; (ii) apply the algorithm of [5] to this huge chain and return the optimal solution. However, the cost of this algorithm is quadratic in the value of N_{iter}. Worse, if we re-execute the same application for $N_{iter} = 20,000$ iterations, we have to recompute the optimal solution from scratch. On the contrary, our approach provides a generic and compact solution that does not depend upon the value of N_{iter}.

Recently, the results of [5] have been extended to deal with linear chains whose tasks do not have constant execution times but instead obey some probability distributions [7]. As pointed out above, for general workflows, deciding which tasks to checkpoint has been shown #P-complete [6], but the results of [8] show that if the graph is scheduled in a sequential manner (linearized), then one can derive an optimal checkpointing strategy. In this paper, we focus on pipelined linear workflows, i.e., on applications expressed as a linear chain of tasks that repeats iteratively.

2.2 Iterative Applications

Iterative methods are popular for solving large sparse linear systems, which have a wide range of applications in several scientific and industrial problems. There are many classic iterative methods including stationary iterative methods like the Jacobi method [9], the Gauss-Seidel method [9] and the Successive Overrelaxation method (SOR) [10], [11], and non-stationary iterative methods like Krylov subspace methods, including Generalized Minimal Residual method (GMRES) [12], Bi-conjugate Gradient Stabilized method (BiCGSTAB) [13], Generalized Conjugate Residual method

Authorized licensed use limited to: University of Glasgow. Downloaded on August 19,2021 at 17:53:12 UTC from IEEE Xplore. Restrictions apply.

(GCR) [14], together with their ABFT (algorithm-based fault-tolerance) variants [15], [16], [17].

Krylov subspace methods fit perfectly our model: the outer iteration corresponds to an iteration of the application in our model, while the inner iterations in each increasing Krylov subspace correspond to the tasks in our model. We report experiments with the GCR algorithm in the Web Supplementary Material (WSM), which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2021.3099440.

The class of iterative applications goes well beyond sparse linear solvers. Uncertainty Quantification (UQ) workflows explore a parameter space in an iterative fashion [18], [19]. This class also encompasses many image and video processing software which operate a chain of computations kernels (each being a task) on a sequence of data sets (each corresponding to an iteration). Examples include image analysis [20], video processing [21], motion detection [22], signal processing [23], [24], databases [25], molecular biology [26], medical imaging [27], and various scientific data analyses, including particle physics [28], earthquake [29], weather and environmental data analyses [26].

2.3 Fault-Tolerance Methods for Iterative Applications

The literature devoted to the study of fault-tolerance methods for iterative linear solvers can be divided into two categories, depending upon whether the focus is on soft errors or on fail-stop errors.

There are some works dealing with soft errors. Chen presented online-ABFT in [30], a technique that can detect soft errors in the specific Krylov subspace iterative methods by leveraging the orthogonality relationship of two vectors in the middle of the program execution. For general iterative methods, Tao *et al.* [31] presented a new online-ABFT approach to detect and recover soft errors by combining a novel checksum-based encoding scheme with a checkpoint/rollback scheme. According to the specific properties of GMRES algorithm, Bridges *et al.* [32] and Elliott *et al.* [33] proposed the FT-GMRES algorithm using selective reliability. Similarly, Sao and Vuduc [34] proposed the self-stabilizing conjugate gradient method (CG) in view of the special properties of CG algorithm: they check that orthogonality is preserved by recomputing scalar products that should be zero and restarting whenever a threshold is exceeded. Ozturk *et al.* [35] proposed a decreasing energy norm based on the mathematical properties to detect soft errors leading to silent data corruption (SDC) for GMRES, CG and Conjugate Residual method (CR).

There are some works dealing with fail-stop errors. To reduce the fault tolerance overhead incurred by checkpointing, Chen [36] proposed a recovery method for iterative methods without checkpointing based on the specific properties of iterative methods. Tao *et al.* [37] improved the checkpointing performance for iterative methods under a novel lossy checkpointing scheme. Langou *et al.* [17] presented a lossy approach which is a checkpoint-free fault tolerant scheme for parallel iterative methods. The iterative method is restarted with a new vector which is a new approximate solution recovered from a fail-stop error by using the data of the non-failed processors. Agullo *et al.* [15], [16] extended this approach by computing a well-suited initial guess which is defined by interpolating the lost entries of the current iterate vector available on surviving nodes, in order to restart the Krylov method. Pachajoa *et al.* [38] compared the exact state reconstruction (ESR) approach based on the method proposed by Chen [36] with the heuristic linear interpolation (LI) approach by Langou *et al.* [17] and Agullo *et al.* [15], [16]. They later extended the ESR approach for protecting the PCG method against multiple and simultaneous node failures [39], [40]. Altogether, fault-tolerance methods proposed to mitigate the impact of fail-stop errors in iterative applications are application-specific, and can only be applied to a particular class of iterative algorithms. Moreover, their performance highly depends upon specific properties of the algorithms, and, for instance, considerably vary from one Krylov method to another.

The main contribution of this work is to provide a general-purpose approach to deal with fail-stop errors in iterative applications. Our optimal checkpointing strategy is agnostic of any specific property of the target iterative application. Instead, it abstracts the iterative application as a chain of cyclic tasks, and provides the optimal periodic checkpoint pattern based only upon generic information such as task durations and checkpoint costs.

3 Model

In this section, we detail the application and platform models. Then we define checkpoint strategies and formally state the optimization objective. See Table 1 for a summary of main notations.

3.1 Application Model

We consider an iterative application $\mathcal{A}$. Each iteration of the application consists of n parallel tasks a_i, where $0 \leq i < n$, task a_i has length t_i and memory footprint M_i. We define the length of an iteration as $T = \sum_{i=0}^{n-1} t_i$.

The tasks are executed consecutively: let $i[n]$ denote the remainder of the integer division of i by n (modulo operation); then a task a_i is always followed by a task $a_{i+1[n]}$. We assume that the application executes for a long time and consider an unbounded number of iterations (but we use 1,000 iterations in the experiments). For short we write $\mathcal{A} = (a_0, \ldots, a_{n-1})^\infty$. As stated before, we execute tasks one after the other. The first executed task is a_0, followed by a_1, and so on. The nth task is a_{n-1} and the $(n+1)$st task is a_0 again. In general, the kth task is $a_{k-1[n]}$. Note that we index tasks from 0 to use the modulo operation, hence this shift when counting executed tasks.

We assume that the tasks of the application can be checkpointed at the end of their execution. We consider a general model where the checkpoint time of task a_i is c_i and its recovery time is r_i. We refer to c_i and r_i as operations of type i. We do not assume that $c_i = r_i$; instead, we simply assume monotone I/O costs

$$\text{for all } i, j, c_i \geq c_j \Rightarrow r_i \geq r_j. \tag{1}$$

Essentially this assumption states that if a task is longer to checkpoint than another one, then restarting from this checkpoint is also longer. This is coherent with the fact that

Authorized licensed use limited to: University of Glasgow. Downloaded on August 19,2021 at 17:53:12 UTC from IEEE Xplore. Restrictions apply.

TABLE 1
Summary of Main Notations

Application	
n	number of tasks per iteration
N_{iter}	number of iterations
a_i	task number i, $0 \leq i < n$, in each iteration
t_i	duration of task a_i
c_i, r_i	checkpoint and recovery time for task a_i
T	length of an iteration
Platform	
D	downtime
$\mu = 1/\lambda$	platform MTBF (λ is the parameter of the failure distribution)
Schedule $\mathcal{S}$	
m_i	task number m_i is checkpoint number i in the schedule
$C_i^{\mathcal{S}}$	checkpoint cost at end of chunk number i
$W_i^{\mathcal{S}}$	length of chunk number i (including tasks number $m_{i-1}+1$ to m_i)
$R_{i-1}^{\mathcal{S}}$	recovery cost when re-executing chunk number i
$\mathrm{SD}(\mathcal{S})$	slowdown of schedule $\mathcal{S}$
P	checkpoint path or pattern
$\ell(P)$	length of checkpoint path P
$\mathcal{C}(P)$	expected execution time, or cost, of checkpoint path P
$\mathrm{SD}(P)$	slowdown of a path P
w_{c_i}	Young/Daly period for checkpoint type c_i, where $w_{c_i} = \sqrt{\frac{2c_i}{\lambda}}$
$2nM^*$	upper bound for length of optimal period, where $M^* = \max_i w_{c_i} + T$
k^*	number of iterations $\lceil \frac{M^*}{T} \rceil$ taking place during time M^*

checkpoint and recovery costs are often closely related, and are a function of the volume of data to save. Furthermore, this assumption is general enough to account for different read and write bandwidths.

We assume that all task parameters (execution time, checkpoint, recovery) are known. This is a natural assumption for iterative applications which repeat each task a large number of times and can determine their characteristics either through an analytical model or by repetitive sampling. However, to assess the robustness of the approach, we also report experiments using stochastic execution times derived from a Normal probability distribution.

3.2 Platform Model

We consider a parallel platform whose nodes are subject to fail-stop errors, or failures. A failure, such as a node crash, interrupts the execution of the node and provokes the loss of its whole memory. 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 consists of using a checkpoint-restart mechanism: the state of the application is periodically checkpointed, which means that all participating nodes take a checkpoint simultaneously: this is the standard coordinated checkpointing protocol which is routinely used on large-scale platforms [41], where each node writes its share of 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 which will replace the faulty processor [2], [4]. 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). Then the execution is resumed from that point on, rather than starting again from scratch.

We assume that the iterative application experiences failures whose inter-arrival times follow an Exponential distribution $Exp(\lambda)$ of parameter $\lambda > 0$, whose PDF (Probability Density Function) is $f(x) = \lambda e^{-\lambda x}$ for $x \geq 0$. The MTBF is $\mu = \frac{1}{\lambda}$ and corresponds to the MTBF of individual processors divided by the total number of processors enrolled in the application [2]. As stated in the introduction, 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 [42], [43]. Hence, a parallel applications using a significant fraction of the platform will typically experience a failure every few days.

The key for an efficient checkpointing policy is to decide how often to checkpoint. Young [3] and Daly [4] derived the well-known Young/Daly formula $\mathcal{P}_{YD} = \sqrt{2\mu C}$ for the optimal checkpointing period, where μ is the application MTBF and C is the checkpoint duration, as defined above.

3.3 Schedule

Informally, a schedule defines which tasks are checkpointed. A priori, there is no reason for a schedule to enforce a regular pattern of checkpoints that repeats over time. In other words, a schedule can be aperiodic. However, one major contribution of this work is to show that periodic schedules are optimal, and to exhibit the optimal period as the output of a polynomial-time algorithm. We need a few definitions before stating the objective function to be minimized by *optimal* schedules. First we identify a schedule with the list of the tasks that it checkpoints:

Definition 1 (Schedule). *A schedule $\mathcal{S}$ is an infinite increasing sequence $\mathcal{S} = (m_1, m_2, \ldots)$ which represents the list of checkpointed tasks: the m_i^{th} task (i.e., task number m_i) is checkpointed, and the tasks whose number does not belong to the list are not checkpointed.*

In other words, checkpoint number i in the schedule takes place at the end of task number m_i. The cost to checkpoint that task m_i is $c_{m_i-1[n]}$ (because of the index shift noted above). Without loss of generality, we assume that the schedule checkpoints infinitely many tasks, i.e., $\lim_{i\to\infty} m_i = \infty$. Indeed, consider any task in the application: eventually there must be a checkpoint after that task, otherwise the expected execution time from that task on is not bounded, because the fault-rate λ is nonzero.

A schedule $\mathcal{S}$ can be viewed as a succession of task chunks between two consecutive checkpoints. We use the following notations for the ith chunk between checkpoint number $i-1$ (or the beginning of the execution if $i = 1$) and checkpoint number i, see Fig. 1:

Authorized licensed use limited to: University of Glasgow. Downloaded on August 19,2021 at 17:53:12 UTC from IEEE Xplore. Restrictions apply.

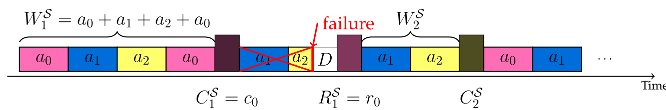


Fig. 1. Notations drawn in a schedule for an application with $n = 3$ tasks.

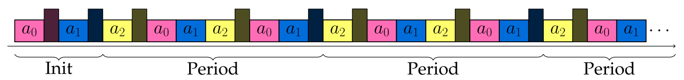


Fig. 2. A periodic schedule where the period is repeated over time.

- The length of the tasks in the chunk is

$$W_i^{\mathcal{S}} = \sum_{j=m_{i-1}+1}^{m_i} t_{j-1[n]}.$$

- The checkpoint cost at the end of the chunk is the cost of checkpoint number i, namely

$$C_i^{\mathcal{S}} = c_{m_i-1[n]}.$$

- The recovery cost when re-executing the chunk is the cost of recovering from checkpoint number $i-1$, namely

$$R_{i-1}^{\mathcal{S}} = r_{m_{i-1}-1[n]}.$$

  When $i = 1$ (first chunk), we let $m_0 = 0$, and $R_0^{\mathcal{S}}$ denotes the cost of reading input data.

3.4 Objective Function

Intuitively, a good schedule will minimize the slowdown during the execution. This slowdown comes from two sources of overhead: the checkpoints that are inserted, and the time lost due to failures. When a failure strikes during execution, the work executed since the last checkpoint is lost; there is a downtime, followed by a recovery, and then the re-execution of the work that has been lost due to the failure. Altogether, the overhead is not deterministic and varies from one execution to the other, hence we aim at minimizing the slowdown in expectation.

Given a schedule $\mathcal{S}$, we rely on a well-known formula to compute the expected execution time of a chunk. Indeed, the expected execution time $\mathbb{E}_\lambda(w, c, r)$ to execute w consecutive seconds of work followed by a checkpoint of size c with a recovery of size r is given by [2]

$$\mathbb{E}_\lambda(w, c, r) = \left(\frac{1}{\lambda} + D\right) e^{\lambda r} \left(e^{\lambda(w+c)} - 1\right). \tag{2}$$

The expected time to execute the chunk number i is thus $\mathbb{E}_\lambda(W_i^{\mathcal{S}}, C_i^{\mathcal{S}}, R_{i-1}^{\mathcal{S}})$. Hence the expected time to execute the first i chunks is $\sum_{j=1}^{i} \mathbb{E}_\lambda(W_j^{\mathcal{S}}, C_j^{\mathcal{S}}, R_{j-1}^{\mathcal{S}})$, The slowdown incurred for the first i chunks (i.e., up to checkpoint number i) is therefore

$$\mathrm{SD}_i(\mathcal{S}) = \frac{\sum_{j=1}^{i} \mathbb{E}_\lambda(W_j^{\mathcal{S}}, C_j^{\mathcal{S}}, R_{j-1}^{\mathcal{S}})}{\sum_{j=1}^{m_i} t_{j-1[n]}}. \tag{3}$$

In Equation (3), the numerator is the expected time to execute the first i chunks, while the denominator is the duration of the tasks up to checkpoint number i, which corresponds to the resilience-free and failure-free execution.

Unfortunately, there is no reason that $\lim_{i\to\infty} \mathrm{SD}_i(\mathcal{S})$ would exist. However, we can use the upper limit of $\mathrm{SD}_i(\mathcal{S})$ to define the slowdown of schedule $\mathcal{S}$

Definition 2 (Slowdown). *The slowdown* SD *of a schedule is*

$$\mathrm{SD}(\mathcal{S}) = \overline{\lim_{i\to\infty}} \mathrm{SD}_i(\mathcal{S}). \tag{4}$$

We know that this upper limit is bounded for some schedules. Consider for instance the schedule $\mathcal{S}$ that checkpoint all tasks: $m_i = i$ for all $i \geq 1$. This schedule repeats the same pattern of checkpoints every iteration, so that its slowdown is

$$\mathrm{SD}(\mathcal{S}) = \frac{\mathrm{SD}_n(\mathcal{S})}{T} = \frac{\sum_{j=0}^{n-1} \left(\frac{1}{\lambda} + D\right) e^{\lambda r_{j-1[n]}} \left(e^{\lambda(t_j+c_j)} - 1\right)}{T}.$$

Recall that T is the length of an iteration. We are now ready to define an optimal schedule:

Definition 3 (Optimal schedule). *A schedule is optimal if its slowdown* $\mathrm{SD}(\mathcal{S})$ *is minimal over all possible schedules.*

Note that the definition does not assume that there exists a unique optimal schedule. A major contribution of this paper is to show that there exists an optimal schedule which is periodic, i.e., which repeats the same pattern of checkpoints after some point (see below for the formal definition). This important result will allow us to consider only a finite number of candidate schedules, and to design a polynomial-time algorithm to find an optimal schedule.

3.5 Periodic Schedules

Periodic schedules are natural schedules that can be expressed in a compact form. As already mentioned, after some possible initialization phase, a periodic schedule repeats the same sequence of checkpoints over and over. Here is the formal definition:

Definition 4 (Periodic schedules). *A schedule* $(m_1, m_2, \ldots)$ *is periodic if there exists two indices* i_0 *and* k_0 *such that for all* $i > i_0$, $m_i - m_{i-1} = m_{i+k_0} - m_{i+k_0-1}$.

An example of periodic schedule is given in Fig. 2. Intuitively, the schedule enters its steady state after checkpoint number i_0 (with possibly $i_0 = 0$): the period starts right after task number m_{i_0}, and then repeats the same sequence of k_0 checkpoints: the first checkpoint of the period is taken after $m_{i_0+1} - m_{i_0}$ tasks, the second one after $m_{i_0+2} - m_{i_0+1}$ tasks, until the last checkpoint of the period, that of task number $m_{i_0+k_0}$. Then the period repeats indefinitely.

For a periodic schedule, the limit $\lim_{i\to\infty} \mathrm{SD}_i(\mathcal{S})$ always exists, and is given by the slowdown incurred during each (infinitely repeating) period. Specifically, given i_0 and k_0 in Definition 4, we see from Equation (3) that this slowdown becomes

Authorized licensed use limited to: University of Glasgow. Downloaded on August 19,2021 at 17:53:12 UTC from IEEE Xplore. Restrictions apply.

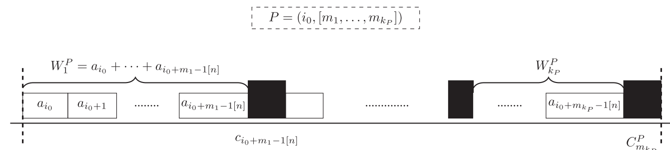


Fig. 3. Sequence of operations of a checkpoint path $P = (i_0, [m_1, \ldots, m_{k_P}])$. Its length is the sum of its *useful* work (white boxes). Its cost corresponds to its expected execution time if a checkpoint was taken right before its start.

$$\frac{\sum_{i=i_0+1}^{i_0+k_0} \mathbb{E}_\lambda(W_i^{\mathcal{S}}, C_i^{\mathcal{S}}, R_{i-1}^{\mathcal{S}})}{\sum_{j=m_{i_0}+1}^{m_{i_0+k_0}} t_{j-1[n]}}.$$

We prove this result formally below. The major results of this work are the following two theorems, which we prove in Section 4 below.

Theorem 1. *There exists a periodic schedule that is optimal.*

Theorem 2. *We can compute an optimal periodic schedule in polynomial time.*

Proof Sketch. The proof has several steps. First we prove that there exists an optimal periodic schedule, i.e., a periodic schedule whose slowdown is minimal. Then we show how to bound the length of the period of this schedule. Once this is done, we have a finite number of periods to look for, and we exhibit a dynamic programming algorithm that determines the optimal period in polynomial time, independently of the number of iterations.

4 Optimal Checkpoint Strategy

In this section, we present several theoretical results and prove Theorems 1 and 2. Specifically, we start by showing that we can indeed focus on periodic algorithms (Theorem 1) in Section 4.3. Then in Section 4.4, we show that we can compute an optimal periodic schedule in polynomial time (Theorem 2).

Beforehand, we introduce the definition of a *pattern* which is at the heart of periodic algorithms (Section 4.1), and we present several important properties of patterns in Section 4.2.

4.1 Paths and Patterns

Definition 5 (Checkpoint Paths). *A* Checkpoint Path *$(P = (i_0, [m_1, \ldots, m_{k_P}]))$ is a sequence of m_{k_P} tasks $b_0, \ldots, b_{m_{k_P}-1}$ such that*

1) *for $0 \le i \le m_{k_P} - 1$, $b_i = a_{i_0+i[n]}$;*
2) *for $1 \le j \le k_P$, b_{m_j-1} is checkpointed.*

Thus the path starts at task $b_0 = a_{i_0}$ and includes m_{k_P} tasks, up to task $b_{m_{k_P}-1} = a_{i_0+m_{k_P}-1[n]}$. The path includes k_P checkpoints, including the checkpoint of its last task. The m_i^{th} task of the pattern is checkpointed, for $1 \le i \le k_P$. See Fig. 3for an illustration.

We use the following notations for a path P: for $1 \le i \le k_P$, we define $W_i^P = \sum_{j=m_{i-1}}^{m_i-1} t_{i_0+j[n]}$ (with the special case $m_0 = 0$), $C_i^P = c_{i_0+m_i-1[n]}$, $R_i^P = r_{i_0+m_i-1[n]}$ (with the special case: $R_0^P = r_{i_0-1[n]}$). We define the length (ℓ) of a checkpoint path $\ell(P) = \sum_{i=1}^{k_P} W_i^P$ and its expected execution time, or cost ($\mathcal{C}$)

$$\mathcal{C}(P) = \sum_{i=1}^{k_P} \mathbb{E}_\lambda\big(W_i^P, C_i^P, R_{i-1}^P\big)$$
$$= \left(\frac{1}{\lambda} + D\right) \sum_{i=1}^{k_P} e^{\lambda R_{i-1}^P}\left(e^{\lambda(W_i^P + C_i^P)} - 1\right).$$

Definition 6 (Patterns). *A* Checkpoint Pattern *is a checkpoint path $P = (i_0, [m_1, \ldots, m_{k_P}])$ such that $m_{k_P} = 0[n]$. Note, for pattern P, its length is $\ell(P) = \frac{m_{k_P}}{n} T$, where $T = \sum_{i=0}^{n-1} t_i$.*

Such patterns are basic blocks to define periodic schedules. We detail this relation below in Section 4.3.

Definition 7 (Slowdown of a path (or pattern)). *The slowdown of a path is defined as: $SD(P) = \frac{\mathcal{C}(P)}{\ell(P)}$.*

4.2 Pattern Properties

Using these definitions, we show the following result:

Theorem 3. *Given a schedule $\mathcal{S}$ of slowdown $\mathrm{SD}(\mathcal{S})$, there exists a pattern $\mathcal{P}$ such that $\mathrm{SD}(\mathcal{P}) \le \mathrm{SD}(\mathcal{S})$.*

Following the Proof. Theorem 3 aims at showing the existence of a pattern whose slowdown is at most that of any algorithm (hence including optimal algorithms). To do so, we construct a sequence of patterns whose slowdown converges to the requested slowdown (Lemma 1) based on sequences taken from the algorithm $\mathcal{S}$. In addition, we impose that this sequence satisfies a *size property*, i.e., that each element of this sequence contains at most n checkpoints (Corollary 1). This then helps to find a pattern whose slowdown is exactly that of $\mathcal{S}$.

We start by proving a series of results.

Lemma 1. *Given a schedule $\mathcal{S}$, there exists a sequence of patterns $(\mathcal{P}_r)$ such that, for all $r \in \mathbb{N}$, $\mathrm{SD}(\mathcal{P}_r) \le \mathrm{SD}(\mathcal{S}) + 1/r$.*

Proof. Consider a schedule $\mathcal{S} = (m_i)_{i\in\mathbb{N}}$ of finite slowdown $\mathrm{SD}(\mathcal{S})$. There exists a checkpoint type i_0 which is taken an infinite number of times. We denote by σ_{i_0} the function such that, for all i, $\sigma_{i_0}(i)$ is the ith occurence of checkpoint c_{i_0} in the schedule $\mathcal{S}$ (we set $\sigma_{i_0}(0) = 0$).

In the following, we partition the schedule into paths:

$$\begin{cases} \mathcal{M}_1 = (0, [m_1, \ldots, m_{\sigma_{i_0}(1)}]) \\ \mathcal{M}_i = (i_0 + 1[n], [(m_{\sigma_{i_0}(i-1)+1} - m_{\sigma_{i_0}(i-1)}), \ldots, \\ \qquad (m_{\sigma_{i_0}(i)} - m_{\sigma_{i_0}(i-1)})]) \qquad (\forall i > 1) \end{cases}.$$

Intuitively, $\mathcal{M}_1$ is the beginning of the schedule until the first checkpoint of type i_0. Then $\mathcal{M}_2$ is the pattern starting right after and extending up to the second checkpoint of type i_0, and so on. In the definition of $\mathcal{M}_i$, checkpoint indices are shifted to account for the location where the path starts. See Fig. 4 for an illustration. By construction, each $\mathcal{M}_i$ is indeed a pattern, except for $\mathcal{M}_1$, which is only a path if $i_0 \neq n - 1$.

We now study the slowdown $\mathrm{SD}_{\sigma_{i_0}(i)}$ up to the ith checkpoint of type i_0, i.e., the slowdown of the first i segments. We have

Authorized licensed use limited to: University of Glasgow. Downloaded on August 19,2021 at 17:53:12 UTC from IEEE Xplore. Restrictions apply.

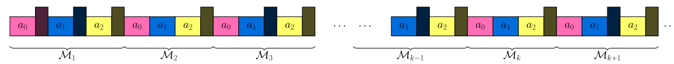


Fig. 4. Partitioning the schedule into patterns around the checkpoints c_2 (yellow).

$$\mathrm{SD}_{\sigma_{i_0}(i)} = \frac{\sum_{k=1}^{i} \sum_{j=\sigma_{i_0}(k-1)+1}^{\sigma_{i_0}(k)} \mathbb{E}_\lambda(W_j^{\mathcal{S}}, C_j^{\mathcal{S}}, R_{j-1}^{\mathcal{S}})}{\sum_{k=1}^{i} \sum_{j=m_{\sigma_{i_0}(k-1)+1}}^{m_{\sigma_{i_0}(k)}} t_{j-1[n]}}$$
$$= \frac{\sum_{k=1}^{i} \mathcal{C}(\mathcal{M}_k)}{\sum_{k=1}^{i} \ell(\mathcal{M}_k)} = \sum_{k=1}^{i} \alpha_{i,k} \cdot \mathrm{SD}(\mathcal{M}_k), \tag{5}$$

where $\alpha_{i,k} = \frac{\ell(\mathcal{M}_k)}{\sum_{j=1}^{i} \ell(\mathcal{M}_j)}$. Hence $\sum_{k=1}^{i} \alpha_{i,k} = 1$, and we have expressed $\mathrm{SD}_{\sigma_{i_0}(i)}$ as a weighted average of the path slowdowns $\mathrm{SD}(\mathcal{M}_k)$.

By definition of $\overline{\lim}$ we have

- $\overline{\lim}_{i\to\infty} \mathrm{SD}_{\sigma_{i_0}(i)}(\mathcal{S}) \le \overline{\lim}_{i\to\infty} \mathrm{SD}_i(\mathcal{S}) = \mathrm{SD}(\mathcal{S})$;
- For all r, there exists i_r such that $\forall i > i_r$,

$$\mathrm{SD}_{\sigma_{i_0}(i)}(\mathcal{S}) \le \overline{\lim_{i\to\infty}} \mathrm{SD}_{\sigma_{i_0}(i)}(\mathcal{S}) + \frac{1}{r} \le \mathrm{SD}(\mathcal{S}) + \frac{1}{r}.$$

Using Equation (5), we obtain

$$\sum_{k=1}^{i} \alpha_{i,k} \mathrm{SD}(\mathcal{M}_k) \le \mathrm{SD}(\mathcal{S}) + \frac{1}{r}.$$

Since this is a weighted average, it means that there exists k_r, where $1 \le k_r \le i$ such that $\mathrm{SD}(\mathcal{M}_{k_r}) \le \mathrm{SD}(\mathcal{S}) + \frac{1}{r}$. If $k_r \ne 1$, or if $k_r = 1$ and $i_0 = n-1$, we have found the desired pattern by letting $\mathcal{P}_r = \mathcal{M}_{k_r}$. Otherwise, we redo the same proof using the truncated schedule $\widetilde{\mathcal{S}}$ where we delete the first $i_0 + 1$ tasks. Then $\widetilde{\mathcal{S}}$ is a valid schedule for a rotation of the original application, namely for the application $\widetilde{\mathcal{A}} = (a_{i_0+1[n]}, \ldots, a_{i_0})^\infty$, and it has same slowdown as $\mathcal{S}$. The path $\widetilde{\mathcal{M}}_i$ of $\widetilde{\mathcal{S}}$ is the same as the path $\mathcal{M}_{i+1}$ of $\mathcal{S}$ for all i, hence all the paths of $\widetilde{\mathcal{S}}$ are patterns. We then derive the result just as above. □

Lemma 2. *For all pattern $\mathcal{P}$, there exists a pattern $\tilde{\mathcal{P}}$ such that:*

1) $\mathrm{SD}(\tilde{\mathcal{P}}) \le \mathrm{SD}(\mathcal{P})$;
2) *$\tilde{\mathcal{P}}$ contains at most n checkpoints.*

Proof. We show this result by induction on the number of checkpoints in $\mathcal{P}$. Assume $\mathcal{P} = (i_0, [m_1, \ldots, m_k])$, with $k > n$. Then there exists $i_1 < i_2$ such that: $i_0 + m_{i_1}[n] = i_0 + m_{i_2}[n]$ (i.e., the $m_{i_1}^{\text{th}}$ and $m_{i_2}^{\text{th}}$ tasks of the pattern are identical and equal to $a_{i_0+m_{i_1}-1[n]}$).

We now consider the two patterns

$$\mathcal{P}_1 = (i_0 + m_{i_1}[n], [m_{i_1+1} - m_{i_1}, \ldots, m_{i_2} - m_{i_1}]);$$
$$\mathcal{P}_2 = (i_0 + m_{i_2}[n], [m_{i_2+1} - m_{i_2}, \ldots, m_k - m_{i_2}, m_1 + (m_k - m_{i_2}), \ldots, m_{i_1} + (m_k - m_{i_2})]).$$

Here, we have decomposed the original pattern into three paths, $\mathcal{P}_{\text{begin}}$ (up to the $m_{i_1}^{\text{th}}$ task), $\mathcal{P}_1$ (from the next task up to the $m_{i_2}^{\text{th}}$ task) and $\mathcal{P}_{\text{end}}$ (from the next task up to

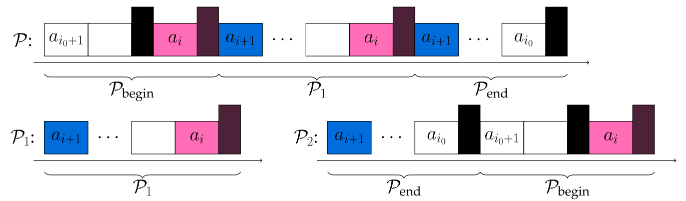


Fig. 5. From a pattern $\mathcal{P}$ with two identical checkpoints, c_i, to its decomposition into $\mathcal{P}_1$ and $\mathcal{P}_2$.

the end of the pattern). Now, $\mathcal{P}_2$ is simply the concatenation of $\mathcal{P}_{\text{end}}$ followed by $\mathcal{P}_{\text{begin}}$. See Fig. 5 for an illustration.

We immediately have

$$\mathrm{SD}(\mathcal{P}) = \frac{\ell(\mathcal{P}_1)}{\ell(\mathcal{P}_1) + \ell(\mathcal{P}_2)} \mathrm{SD}(\mathcal{P}_1) + \frac{\ell(\mathcal{P}_2)}{\ell(\mathcal{P}_1) + \ell(\mathcal{P}_2)} \cdot \mathrm{SD}(\mathcal{P}_2).$$

Because this is a weighted average, then $\min(\mathrm{SD}(\mathcal{P}_1), \mathrm{SD}(\mathcal{P}_2)) \le \mathrm{SD}(\mathcal{P})$. Each of these patterns has fewer checkpoints than the initial pattern, which concludes the proof. □

Corollary 1. *Given a schedule $\mathcal{S}$, there exists a sequence of patterns $(\tilde{\mathcal{P}}_r)$ for all $r \ge 1$ such that:*

1) *For all r, $\tilde{\mathcal{P}}_r$ contains at most n checkpoints;*
2) $\mathrm{SD}(\tilde{\mathcal{P}}_r) \le \mathrm{SD}(\mathcal{S}) + 1/r$.

This corollary is a direct consequence of Lemma 1, for the existence of a sequence that satisfies the slowdown constraint, and of Lemma 2, for transforming this sequence into a sequence of patterns that include at most n checkpoints.

Following the Proof: At this point, we have constructed a sequence of patterns, whose slowdown converges towards $\mathrm{SD}(\mathcal{S})$. It remains to show the existence of a pattern that reaches the limit. In order to do so, we show that if the number of checkpoints in a pattern is bounded, then the length of the pattern has to be bounded too, otherwise its slowdown would diverge. This is the result shown in Lemma 3.

Lemma 3. *Given M and k, if $\mathcal{P}$ is a pattern with at most k checkpoints and $\mathrm{SD}(\mathcal{P}) \le M$, then there exists a constant $W_{M,k}$ such that $\ell(\mathcal{P}) \le W_{M,k}$.*

Proof. Given a pattern $\mathcal{P}$ with k checkpoints, and of length $\ell(\mathcal{P}) = W$, we let $W_1^{\mathcal{P}}, W_2^{\mathcal{P}}, \ldots, W_k^{\mathcal{P}}$ denote the work between its checkpoints. By definition, $\sum_{i=1}^{k} W_i^{\mathcal{P}} = \ell(\mathcal{P})$. Hence there exists i_1 such that $W_{i_1}^{\mathcal{P}} \ge \frac{1}{k}\ell(\mathcal{P})$.

We are now interested in the slowdown of the pattern

$$\mathrm{SD}(\mathcal{P}) = \frac{\sum_{i=1}^{k} \mathbb{E}_\lambda(W_i^{\mathcal{P}}, C_i^{\mathcal{P}}, R_{i-1}^{\mathcal{P}})}{\ell(\mathcal{P})}$$
$$\ge \frac{\mathbb{E}_\lambda(W_{i_1}^{\mathcal{P}}, C_{i_1}^{\mathcal{P}}, R_{i_1-1}^{\mathcal{P}})}{\ell(\mathcal{P})}$$
$$\ge \frac{\mathbb{E}_\lambda(W_{i_1}^{\mathcal{P}}, 0, 0)}{\ell(\mathcal{P})} = \left(\frac{1}{\lambda} + D\right) \frac{e^{\lambda W_{i_1}^{\mathcal{P}}} - 1}{\ell(\mathcal{P})}$$
$$\ge \left(\frac{1}{\lambda} + D\right) \frac{e^{\frac{\lambda}{k}\ell(\mathcal{P})} - 1}{\ell(\mathcal{P})}.$$

Authorized licensed use limited to: University of Glasgow. Downloaded on August 19,2021 at 17:53:12 UTC from IEEE Xplore. Restrictions apply.

But $\frac{e^{\frac{\lambda}{k}x}-1}{x}$ tends to infinity when x tends to infinity; hence, because $\mathrm{SD}(\mathcal{P}) \leq M$, we have that $\ell(\mathcal{P})$ is bounded by a function of M and k. Hence the result. □

Proof of Theorem 3. We now conclude the proof of Theorem 3. From Corollary 1, we have a sequence of patterns $(\tilde{\mathcal{P}}_r)$ with at most n checkpoints and of slowdown $\mathrm{SD}(\tilde{\mathcal{P}}_r) \leq \mathrm{SD}(\mathcal{S}) + 1/r \leq 2\mathrm{SD}(\mathcal{S})$.

From Lemma 3, there exists an upper bound such that, for all r, $\ell(\tilde{\mathcal{P}}_r) \leq \tilde{W}$. We show that there is only a bounded number of patterns that satisfy this property:

- Since the length of a pattern is a multiple of $T = \sum_{i=0}^{n-1} t_i$, there are at most $K = \lfloor \frac{\tilde{W}}{T} \rfloor$ possible lengths.
- For a length kT, $1 \leq k \leq K$, there are kn possible checkpoint locations and at most n checkpoints, hence at most $\binom{kn}{n}$ patterns.

Hence in total, the number of possible patterns is upper bounded by $K\binom{Kn}{n}$. The set $\{\mathrm{SD}(\tilde{\mathcal{P}}_r) | r \geq 1\}$ is finite and admits a minimum $S_{\min}$. Let r_0 be one index achieving the minimum: $S_{\min} = \mathrm{SD}(\tilde{\mathcal{P}}_{r_0})$.

Finally, we show that $S_{\min} \leq \mathrm{SD}(\mathcal{S})$: indeed, otherwise there would exist r such that $S_{\min} > \mathrm{SD}(\mathcal{S}) + \frac{1}{r}$ and we would have $\mathrm{SD}(\tilde{\mathcal{P}}_{r_0}) > \mathrm{SD}(\tilde{\mathcal{P}}_r)$, thereby contradicting the minimality. Hence the result. □

4.3 Periodic Schedules

Using the properties of patterns, we are ready to derive Theorem 1. We start by rewriting the definition of periodic schedules using patterns. Indeed, the values i_0 and k_0 from Definition 4 allow us to define a pattern that is repeated all throughout the execution. We then select the pattern of minimal length that occurs as early as possible:

Definition 8 (Pattern of a periodic schedule). *Given a periodic schedule $\mathcal{S} = (m_1, m_2, \ldots)$. Let (k_0, i_0) be the smallest pair (for the lexicographic order) that satisfies: for all $i > i_0$, $m_i - m_{i-1} = m_{i+k_0} - m_{i+k_0-1}$, and $m_{i_0+k_0} - m_{i_0} = 0[n]$. We say that*

$$P_{\mathcal{S}} = (m_{i_0}[n], [m_{i_0+1} - m_{i_0}, \ldots, m_{i_0+k_0} - m_{i_0}]),$$

is the pattern of the schedule.

The lexicographic order means that we select first a pattern of minimal length, and in case of a tie, the pattern that starts as early as possible.

Theorem 4 (Slowdown of a periodic schedule). *Given a periodic schedule $\mathcal{S}$, its slowdown is equal to the slowdown of its pattern.*

Proof. Given a periodic schedule $\mathcal{S}$, let

$$P_{\mathcal{S}} = (m_{i_0}[n], [m_{i_0+1} - m_{i_0}, \ldots, m_{i_0+k_0} - m_{i_0}]),$$

be its pattern. We study the function $\mathrm{SD}_i(\mathcal{S})$ by decomposing the schedule up to its ith checkpoint into three parts: a first part, up to the beginning of the pattern, i.e., up to checkpoint number i_0, then a number $k = \lfloor \frac{i-m_{i_0}}{k_0} \rfloor$ of repeating patterns, then a final part (whose length is smaller than $\ell(P_{\mathcal{S}})$. The first and final part become negligible as i tends to infinity, hence

$$\mathrm{SD}(\mathcal{S}) = \overline{\lim_{i\to\infty}} \mathrm{SD}_i(\mathcal{S}) = \frac{\mathcal{C}(P_{\mathcal{S}})}{\ell(P_{\mathcal{S}})} = \mathrm{SD}(P_{\mathcal{S}}).$$

□

Proof of Theorem 1. Finally, putting everything together, we obtain the final result: Theorem 3 states that there exists a pattern P whose slowdown is smaller or equal to that of an optimal schedule. In addition, Theorem 4 states that a periodic schedule whose pattern is P has a slowdown equal to that of P, hence it is optimal. □

4.4 Finding the Optimal Pattern

In this section, we show how to compute the pattern of an optimal periodic algorithm. In the following, we say that a pattern $\mathcal{P} = (i_0, [m_1, m_2, \ldots, m_{k_0}])$ is an *optimal pattern*, if it has minimal slowdown.

4.4.1 Bounding the Length

Following the Proof: In Lemma 3, we have shown that it was possible to bound the length of an optimal pattern, which was helpful to prove the existence of an optimal pattern. In order to derive an optimal solution, we want to use a dynamic program whose complexity depends on the number of tasks in a pattern. Unfortunately, the previous bound may lead to a number of tasks in the pattern which is not polynomially bounded. We now show how one can get a tighter bound (Theorem 5). At the end of this section, we discuss the size of this bound as a function of the problem instance.

In this section we make intensive use of the following *slowdown function*:

$$f(w, c, r) = \frac{\mathbb{E}_\lambda(w, c, r)}{w} = \left(\frac{1}{\lambda} + D\right) e^{\lambda r} \left(\frac{e^{\lambda(w+c)} - 1}{w}\right). \quad (6)$$

Note that we implicitly used the slowdown function when we defined the slowdown of a schedule. We have the following properties:

Lemma 4. *We have the following properties of the slowdown function:*

1) *$w \mapsto f(w, c, r)$ has a unique minimum w_c , is decreasing in the interval $[0, w_c]$ and is increasing in the interval $[w_c, \infty)$*
2) *$c \mapsto f(w, c, r)$ (resp. $r \mapsto f(w, c, r)$) are increasing functions of c (resp. r).*

Proof. 2) is obvious. 1) is the result of [44, Theorem 1]. Note that a first-order approximation of w_c is the well-known Young/Daly formula $w_c = \sqrt{\frac{2c}{\lambda}}$ [3], [4]. □

While one might want to use w_c to minimize f, this is only possible for divisible applications. Here, we can checkpoint only at the end of a task, and the amount of work w can only be the sum of some task durations.

Consider a path starting after a checkpoint c_i (hence with a recovery r_i), and ending in a checkpoint c_j. The amount of

Authorized licensed use limited to: University of Glasgow. Downloaded on August 19,2021 at 17:53:12 UTC from IEEE Xplore. Restrictions apply.

computation w between these two checkpoints is necessarily of the form

$$W_{i,j}(k) = W_{i,j} + k \times T, \text{ for some } k \in \mathbb{N},$$

where

(i) T is the length of the iterations ($T = \sum_{\ell=0}^{n-1} t_\ell$), and
(ii) $W_{i,j}$ is the length between the end of task a_i and the end of task a_j (possibly of the next iteration), i.e.: $W_{i,j} = \sum_{\ell=i+1}^{j} t_\ell$ (case $j > i$), or $W_{i,j} = T - W_{j,i}$ (case $j < i$), or $W_{i,j} = T$ (case $j = i$).

Additionally, $k \mapsto W_{i,j}(k)$ is an increasing function, hence for all pairs (r_i, c_j), there exists $k^\star_{i,j}$ that minimizes the function $k \mapsto f(W_{i,j}(k), c_j, r_i)$. Let $W^\star_{i,j} = W_{i,j}(k^\star_{i,j})$. Because $f(w, c_j, r_i)$ is decreasing for $w < w_{c_j}$, we have $W^\star_{i,j} - T < w_{c_j}$ (otherwise $W^\star_{i,j} - T$ would be a better solution). Finally, we denote

$$M^\star = \max_i w_{c_i} + T.$$

Then, $M^\star \geq \max_{i,j} W^\star_{i,j}$, and by construction, we have the following property for $M^\star$:

Lemma 5. *For all* i, j, k_1, k_2 *such that* $W_{i,j} + k_1 \cdot T \geq W_{i,j} + k_2 \cdot T \geq M^\star$,

$$f(W_{i,j} + k_1 \cdot T, c_j, r_i) > f(W_{i,j} + k_2 \cdot T, c_j, r_i).$$

Finally, we let

$$k^\star = \lfloor M^\star / T \rfloor, \tag{7}$$

denote the number of iterations that take place during time $M^\star$. We are ready to bound the length between two successive checkpoints within an optimal pattern:

Lemma 6. *Given an optimal pattern,* $(i_0, [m_1, m_2, \ldots, m_{k_0}])$, *then for all* $1 \leq i \leq k_0$, $m_i - m_{i-1} \leq 2M^\star$ *(using* $m_0 = 0$*).*

Following the Proof: In order to show this result, we show that if the length between two consecutive checkpoints was larger than the bound, then we could add an intermediate checkpoint and create a pattern of smaller slowdown.

Proof. We start by a preliminary property that we use in the following: we show that if the length between two checkpoints is too high, then we can create a pattern of better slowdown by incorporating a checkpoint in the oversized interval.

Given a pattern $P = (i_0, [m_1, m_2, \ldots, m_{k_0}])$, and given a transformation of this pattern into a pattern P' of equal length with an extra checkpoint of cost C (and recovery R) located between the $(i-1)^{\text{th}}$ and the ith checkpoint of P, after W units of work, one can verify that

$$\begin{aligned} &\mathrm{SD}(P) - \mathrm{SD}(P') = \\ &\frac{\mathbb{E}_\lambda(W_i^P, C_i^P, R_{i-1}^P) - \mathbb{E}_\lambda(W, C, R_{i-1}^P) - \mathbb{E}_\lambda(W_i^P - W, C_i^P, R)}{\ell(P)} \\ &= \frac{W_i^P}{\ell(P)} f(W_i^P, C_i^P, R_{i-1}^P) - \frac{W}{\ell(P)} f(W, C, R_{i-1}^P) \\ &\quad - \frac{W_i^P - W}{\ell(P)} f(W_i^P - W, C_i^P, R). \end{aligned} \tag{8}$$

Indeed, $\ell(P) = \ell(P')$, and all inter-checkpoint intervals are identical (and have an equal cost) in P and P', except for the interval inside which the extra checkpoint has been added.

We can now prove the result. We show the result by contradiction: assume there exists $i \leq k_0$ such that $m_i - m_{i-1} > 2M^\star$. We denote by $i_1 = i_0 + m_{i-1} - 1[n]$ and $i_2 = i_0 + m_i - 1[n]$

• Assume first that $c_{i_2} \geq c_{i_1}$. By monotony, $r_{i_2} \geq r_{i_1}$. We create the pattern P' such that we add to P an additional checkpoint after the task of type i_1 at the location $m_{i-1} + n \cdot k^\star$ (which indeed corresponds to a task of type i_1). Then, using the properties of the slowdown function f, and because $W_i^P > T \cdot k^\star$, we know that

$$\begin{aligned} f(T \cdot k^\star, C_{i-1}^P, R_{i-1}^P) &\leq f(T \cdot k^\star, C_i^P, R_{i-1}^P) \text{(growth)} \\ &< f(W_i^P, C_i^P, R_{i-1}^P) \text{ (shape of } f). \end{aligned}$$

Similarly, $W_i^P > W_i^P - T \cdot k^\star \geq M^\star$, then we have

$$f(W_i^P - T \cdot k^\star, C_i^P, R_{i-1}^P) \leq f(W_i^P, C_i^P, R_{i-1}^P).$$

Finally, plugging back these values into Equation (8), we obtain that P' has a better slowdown than P, contradicting the optimality.

• Assume now that $c_{i_2} \leq c_{i_1}$. By monotony, $r_{i_2} \leq r_{i_1}$. With a similar demonstration, we show that by including a checkpoint of size c_{i_2} at location $m_i - n \cdot k^\star$ (which indeed corresponds to a task of type i_2), leads to the same result. □

Theorem 5. *There exists an optimal pattern* P *whose length satisfies* $\ell(P) \leq 2nM^\star$, *and which includes at most* $2n^2(k^\star + 1)$ *tasks.*

Proof. From Lemma 2, we know that there exists an optimal pattern with at most n checkpoints. Using Lemma 6 which gives a bound on the inter-checkpoint time, we obtain the bound on the length. Thanks to Equation (7), we know that a length of $M^\star$ corresponds to at most $k^\star + 1$ iterations (of n tasks each), which leads to the bound on the number of tasks. □

We now need to check that $k^\star$ is polynomial in the size of the input. The size of the input is $O(n \max_i \log t_i)$, or equivalently $O(n \log T)$, because the n values t_i are encoded in binary. Here we make the natural assumption that $c_i = O(T)$ and $r_i = O(T)$ for $0 \leq i \leq n$, meaning that the largest checkpoint/recovery is not longer than a whole iteration.[2] Recall that $w_{c_i} \approx \sqrt{\frac{2c_i}{\lambda}}$ [3], [4], hence $M^\star = O(\sqrt{\frac{\max_i c_i}{\lambda}} + T)$ and $k^\star = O\left(\frac{1}{\sqrt{\lambda T}} + 1\right)$. We obtain a polynomial value $k^\star =$

2. Technically, we can relax the assumption to $c_i, r_i = O(T^n)$ without increasing the problem size.

Authorized licensed use limited to: University of Glasgow. Downloaded on August 19,2021 at 17:53:12 UTC from IEEE Xplore. Restrictions apply.

$O(n\log T)$ as soon as $\mu = \frac{1}{\lambda} = O(T(n\log T)^2)$. This requires that the application MTBF is not too large in front of the iteration length, which makes full sense because otherwise we would not checkpoint more than very rarely, once every many iterations. In Section 4.4.2, we present a dynamic programming algorithm to compute an optimal pattern, whose complexity is polynomial in n and $k^\star$. This complexity is indeed polynomial in the size of the instance under the very natural assumptions that we made.

Algorithm 1. Finding the Minimum Slowdown of a Pattern of Size at Most $2n^2(k^\star+1)$

```
1:  procedure Pattern k*, n
2:    maxK ← 2n(k* + 1)
3:    for i0 = 0 to n − 1 do ▷ Initialization of ProgDyn
4:      for ℓ = 0 to 2n²(k* + 1) do
5:        for b = 1 to n do
6:          if ℓ = 0 then
7:            C_min(i0, ℓ + 1, ℓ, b) ← 0
8:          else
9:            C_min(i0, min(maxK, ℓ + 1), ℓ, b) ← ∞
10:       for k = 1 to min(maxK, ℓ) do
11:         C_min(i0, k, ℓ, 0) ← ∞
12:   for i0 = 0 to n − 1 do ▷ Precompute Σ_{i=1}^{k} t_{i0+i[n]}
13:     W[i0, 1] ← t_{i0+1[n]}
14:     for k = 2 to 2n²(k* + 1) do
15:       W[i0, k] ← W[i0, k − 1] + t_{i0+k[n]}
16:   for ℓ = 1 to 2n²(k* + 1) do ▷ Computing the ProgDyn
17:     for i0 = 0 to n − 1 do
18:       for k = min(maxK − 1, ℓ) downto 1 do
19:         for b = 1 to n do
20:           C_min(i0, k, ℓ, b) ← min(C_min(i0, k + 1, ℓ, b), E_λ(W[i0, k], C_{i0+k[n]}, R_{i0}) + C_min(i0 + k[n], 1, ℓ − k, b − 1))
21:   SD = ∞ ▷ Computing the minimal slowdown.
22:   T = Σ_{i=1}^{n} t_i
23:   for i0 = 0 to n − 1 do
24:     for m = 1 to 2n(k* + 1) do
25:       SD_temp = C_min(i_o, 1, mn, n)/mT
26:       if SD_temp < SD then
27:         SD ← SD_temp
28:   return SD
```

4.4.2 Computing an Optimal Pattern

In the previous section, we have shown the existence of an *optimal* pattern of polynomial length. Here, we show how one can compute an optimal pattern through a dynamic programming algorithm. This dynamic programming algorithm relies upon the previous results:

- We study patterns P of length at most $2nM^\star$ (thanks to Theorem 5), and we know that an optimal pattern of this length contains a polynomial number of tasks;
- We consider different initial tasks in the pattern ($i_0 \in \{0,\ldots,n-1\}$);
- We use the fact that there can be at most n checkpoints in the optimal pattern (thanks to Lemma 2).

The following lemma characterizes the minimal cost of a checkpoint path.

Lemma 7 (Minimum cost of a path). *The minimal expected execution time (or cost) of a checkpoint path (i) of ℓ tasks, (ii) whose first task is $a_{i_0+1[n]}$, (iii) with at most b checkpoints, (iv) where the $k-1$ first tasks are not checkpointed and, (v) where the last task is checkpointed, is given by;*

$$\mathcal{C}_{\min}(i_0,k,\ell,b) = \min\begin{cases}\mathcal{C}_{\min}(i_0,k+1,\ell,b);\\ \mathbb{E}_\lambda(\sum_{i=1}^{k} t_{i_0+i[n]}, C_{i_0+k[n]}, R_{i_0})\\ \quad + \mathcal{C}_{\min}(i_0+k[n],1,\ell-k,b-1)\end{cases}, \quad (9)$$

when $\ell > 0$ and $b > 0$, and where we consider the following initialisation cases:

$$(a)\quad \mathcal{C}_{\min}(i_0,\ell+1,\ell,b) = \begin{cases}0 & \text{if } \ell = 0\\ \infty & \text{otherwise}\end{cases}$$

$$(b)\quad C(i_0,k,\ell>0,0) = \infty.$$

Please refer to Fig. 6 for a graphical representation of $C(i_0,k,\ell,b)$.

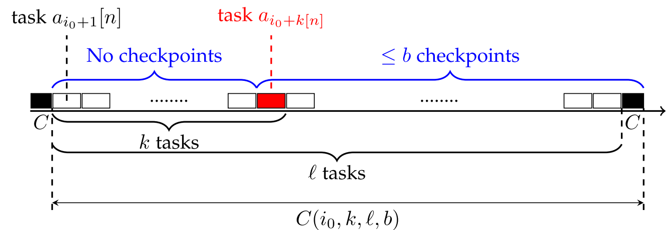


Fig. 6. Illustration of the minimum expected execution time $\mathcal{C}_{\min}(i_0,k,\ell,b)$ of a series of ℓ tasks as characterized by Lemma 7.

Proof. The result is proven recursively. We start with the initialisation cases. When no checkpoint is allowed ($b = 0$, case (b)), it is not even possible to checkpoint the last task (as required by condition (v), so the cost is infinite. The case $k = \ell+1$ leads to ℓ tasks not being checkpointed (condition (iv)), which contradicts the fact that the last task is checkpointed (condition (v)), except when the number of tasks is zero: in this case, we assume that no task is performed in this path and no checkpoint is taken.

We now move to the general case. Considering a path that verifies the condition of the lemma, we distinguish two cases:

(i) The kth task is not checkpointed, which leads to the first k tasks not being checkpointed, hence the minimum cost is $\mathcal{C}_{\min}(i_0,k+1,\ell,b)$;

(ii) The kth task is checkpointed. The cost of the first part $k-1$ tasks not checkpointed followed by this k^{th} task and its checkpoint is given by $\mathbb{E}_\lambda(\sum_{i=1}^{k} t_{i_0+i[n]}, C_{i_0+k[n]}, R_{i_0})$. The cost of the rest of the path is recursively expressed as the minimal cost of a path of length $\ell-k$ that starts after task $i_0+k[n]$ with $b-1$ checkpoints.

We then select the case that leads to the minimal expected execution time.

Thanks to Lemma 6, we know that in an optimal pattern, there are at most $2n(k^\star+1)$ tasks between two checkpoints. So we can safely restrict our search space to $k = 1\ldots 2n(k^\star+1)$ and consider that the cost for larger values of k is infinite. Hence, the previous recursive

Authorized licensed use limited to: University of Glasgow. Downloaded on August 19,2021 at 17:53:12 UTC from IEEE Xplore. Restrictions apply.

TABLE 2
Tasks of the Neuroscience Application

Task	a_0	a_1	a_2	a_3	a_4	a_5	a_6
Mean μ_i (sec)	255	871	588	459	3050	804	1130
Stdev σ_i (sec)	96.7	322	76.8	48.1	263	393	568
Checkpoint time c_i (sec)	22.22	61.11	33.33	50	283.33	16.67	61.11
Recovery time r_i (sec)	8.89	24.44	13.33	20	113.33	6.67	24.44

definition of the cost is applied to the design of the dynamic programming algorithm (Algorithm 1). □

Theorem 6. *PATTERN*$(k^\star, n)$ *(Algorithm 1) returns the slowdown of the pattern of an optimal periodic schedule with time complexity* $O((k^\star)^2 n^5)$.

Proof. We use the fact that there exists an optimal periodic schedule whose pattern includes a number $m \times n$ of tasks with $m \leq 2n(k^\star + 1)$ and uses at most n checkpoints (see Theorem 5). Algorithm 1 computes the minimum cost of all patterns including at most this number of tasks, then computes the minimum cost of a pattern whose number of tasks is a multiple of n. The slowdown that we look for is indeed this cost. The complexity of the algorithm derives from the loop nest necessary to recursively compute $\mathcal{C}_{\min}$. □

5 SIMULATION RESULTS

In this section, we describe the experiments conducted to compare the proposed optimal checkpointing strategy with simpler heuristics. We perform simulations on three application scenarios: two from real-life applications (neuroscience and sparse linear solver), and one using synthetic parameters. However, due to lack of space, the description of the sparse linear solver (a Krylov Subspace method called GCR [14]), together with the corresponding results, are available in the Web Supplementary Material (WSM), available online.

The experimental methodology is presented in Section 5.1. The results for the neuroscience application are detailed in Section 5.2 . The results for the synthetic application are detailed in Section 5.3.

5.1 Experimental Methodology

We detail here the applications, the algorithms used in the simulations and the various settings. All algorithms have been implemented in MATLAB and R. The corresponding code is publicly available at [45].

5.1.1 Neuroscience Application

For the first application scenario, we extracted data from a representative neuroscience application, Spatially Localized Atlas Network Tiles (SLANT) [46]. This is an iterative application composed of $N = 10^3$ iterations, and each iteration has $n = 7$ tasks. These tasks are described in Table 2, with parameters taken from [47]. Table 2 reports the mean and standard deviation of the task execution times, which obey a Normal probability distribution. The Pearson correlation of the different tasks was studied in [47], which showed that the tasks are not correlated except for tasks a_0 and a_1 which are proportional. For the first set of experiments in Section 5.2.1, we consider that the tasks are deterministic (as assumed throughout this work) and use mean values as execution times ($t_i = \mu_i$). However, for the second set of experiments in Section 5.2.4, we assess the robustness of our approach and independently draw execution times from the Normal distributions for tasks $a_0, a_2, a_3, a_4, a_5, a_6$, while a_1 is set to be equal to $3.4 \times a_0$ due to its high correlation with a_0. We use a downtime $D = 5$.

TABLE 3
MTBF for the Neuroscience Application

p_{fail}	10^{-3}	10^{-2}	10^{-1}	$10^{-0.5}$	$10^{-0.1}$
MTBF	82.8 days	8.3 days	19.9 hours	6.3 hours	2.5 hours

5.1.2 Synthetic Application

The second application scenario is randomly generated. We consider an iterative application composed of $N = 10^3$ iterations, each iteration has $n = 10$ or 20 cyclic tasks. We assume that the execution time t_i of each task a_i follows a probability distribution $\mathcal{D}$, where $\mathcal{D}$ is UNIFORM$[a, b]$. The default instantiation for this distribution is $\mu_{\mathcal{D}} = 550$ for UNIFORM$[100, 1000]$.

For this application scenario, we set checkpoint times as $c_i = \eta t_i$, where η is the proportion of checkpoint time to the execution time of each task. We use $r = c$ for the recovery time and a fixed downtime $D = 5$, and we conduct experiments with $\eta = 0.1$. In the WSM, we report results for another instantiation of checkpoint times, which are then taken in UNIFORM$[10, 100]$, independently of the task running times.

5.1.3 Failure Scenarios

We consider a wide range of failure rates. To allow for consistent comparisons of results across different iterative processes, we fix the probability that a failure occurs during each iteration, which we denote as p_{fail}, and then simulate the corresponding failure rate. Formally, for a given p_{fail} value, we compute the failure rate λ such that $p_{\text{fail}} = 1 - e^{-\lambda T}$, where T is the execution time per iteration with n tasks. We conduct experiments for five p_{fail} values: $10^{-3}, 10^{-2}, 10^{-1}, 10^{-0.5}$ and $10^{-0.1}$. For each application, these different values of p_{fail} allow us to quantify the risk faced during execution. For example, $p_{\text{fail}} = 10^{-2}$ means one failure will occur every 100 iterations on average. The risk is highest for $p_{\text{fail}} = 10^{-0.1}$ which corresponds to 1 failure per 1.26 iterations on average, while the risk is lowest for $p_{\text{fail}} = 10^{-3}$ which corresponds to 1 failure per 1,000 iterations on average.

Table 3 provides the correspondence between p_{fail} and actual MTBF values for the neuroscience application. The base time (without checkpoint nor failure) for one iteration of the neuroscience application is 7,157 seconds, or almost 2 hours. Thus 1,000 iterations will last 83 days approximately. For instance we observe that $p_{\text{fail}} = 10^{-1}$ corresponds to one failure every 19.9 hours, which is typical of several large-scale HPC machines that experience around one failure per day. Smaller values of p_{fail} correspond to platforms with fewer failures, one per week or less. Larger values of p_{fail} represent more failure-prone platforms, with a failure every few hours.

Authorized licensed use limited to: University of Glasgow. Downloaded on August 19,2021 at 17:53:12 UTC from IEEE Xplore. Restrictions apply.

Altogether, varying the value of p_{fail} enables to explore a wide range of scenarios.

For the synthetic application, task execution times are defined up to a constant factor: we can envision an arbitrary unit of length, ranging from seconds to hours. Then the value of p_{fail} is more representative of the failure rate than the MTBF, whose calculation would need to fix the execution unit. On the contrary, using p_{fail} enables to directly quantify the risk faced by the application in terms of a failure probability per iteration.

For each experiment, the simulations are performed on 100 randomly generated instances $\{\mathcal{I}_1, \ldots, \mathcal{I}_{100}\}$. For all i, an instance $\mathcal{I}_i$ is a pair $(\mathcal{S}_i, \mathcal{F}_i)$, where $\mathcal{S}_i$ (resp. $\mathcal{F}_i$) is the application (resp. failure) scenario associated to the instance. For the neuro-science application, $\mathcal{S}_i$ corresponds to the values presented in the previous tables, while for the synthetic application scenario, $\mathcal{S}_i$ is randomly generated as described above.

5.1.4 Reference Strategies

We consider four reference strategies. The first two strategies are quite natural: (i) CkptEachIter consists in checkpointing at the end of each iteration, that is, a checkpoint is taken after the last task a_{n-1} of each iteration; and (ii) CkptEachTask consists in checkpointing after every task a_i of every iteration.

The other two strategies are extensions of the Young/Daly approach for divisible applications where one can checkpoint at any time-step with constant cost c: then the optimal period is to checkpoint every $w_c = \sqrt{\frac{2c}{\lambda}}$ seconds (see Lemma 4). For an iterative application, the corresponding approach is to work for w_c seconds and to checkpoint at the end of the current task (and repeat). The difficulty is that c is not well-defined here, because the tasks have different checkpoint costs. With n tasks of checkpoint costs c_i, $0 \leq i < n$, we take the average cost $c_{ave} = \frac{\sum_{0 \leq i < n} c_i}{n}$ and denote the previous strategy using $c = c_{ave}$ as CkptYDAve.[3] Finally, the fourth strategy CkptYDPer is a periodic extension of Young/Daly approach: it chooses the task of an iteration with minimum checkpoint size $c_{\min}$. Only the result of this task will (possibly) be checkpointed. Then it uses the Young-Daly formula to compute how many iterations to include in between two checkpoints, namely $\max\left(1, \text{round}\left(\frac{w_{c_{\min}}}{T}\right)\right)$.

5.1.5 Presenting Results

We report median values in all experiments, and in the scalability analysis we use boxplots. The color chart is the following: red for CkptEachIter, green for CkptEachTask, blue for CkptYDAve and purple for CkptYDPer.

5.2 Results for the Neuroscience Application

5.2.1 Comparison of the Strategies

In Fig. 7, the makespan of each reference strategy is normalized by the optimal makespan (obtained with Algorithm 1, hence the lower the better. This presentation allows us to directly quantify the performance overhead incurred by each strategy with respect to the optimal approach. Checkpointing after each task, as done by CkptEachTask, gives worst

3. We have also experimented with two variants using $c = c_{\min} = \min_{0 \leq i < n} c_i$, and $c = c_{\max} = \max_{0 \leq i < n} c_i$. Results are quite similar.

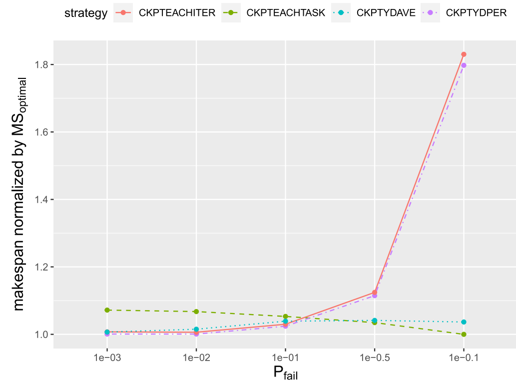


Fig. 7. Normalized performance overhead with different failure probabilities (neuroscience).

performance when p_{fail} is small (very few failures). Its performance improves significantly when the number of failures increases. This behavior is expected as it is a consequence of the very high number of checkpoints that are taken.

On the contrary, the Young-Daly inspired heuristics (CkptYDPer and CkptYDAve) gives almost optimal results when there are very few failures, and they get worse when the number of failures increases. Again, this behavior is expected: with very few failures, if the frequency of checkpointing is of the same order of magnitude as in the optimal solution, the fact that the checkpointing decision that is taken is not optimal has little impact, because the checkpoint overhead is very low. With numerous failures, CkptYDPer, which is limited to at most one checkpoint per iteration, does not checkpoint often enough, and the loss in work when there is a failure gets too expensive; but CkptYDAve can checkpoint more frequently, and its performance degrades less severely, only because it happens to checkpoint some tasks of high checkpoint cost.

Finally CkptEachIter is probably the less interesting strategy as its performance is always worse than CkptYDPer and CkptYDAve. As p_{fail} increases, its performance first improves and then gets worse: when p_{fail} is very small, (i) it does not choose the task with smallest checkpoint size and (ii) it checkpoints too often compared to CkptYDPer and CkptYDAve which would allow to checkpoint after several iterations and not just one; conversely, when p_{fail} is very large, checkpointing once after each iteration is not enough, thus the relative cost of the CkptEachIter strategy increases. It is still interesting to see that the difference with CkptYDPer remains always small, while the difference with CkptYDAve gets large for frequent failures. It seems that finding the smallest checkpoint size is not critical, while finding the best checkpoint frequency is more important.

For all strategies, when the frequency of failures reaches its maximal value $p_{\text{fail}} = 10^{-0.1}$ (approximately 4 failures every 5 iterations), then all greedy heuristics perform poorly, and the optimal solution provides significant gains, even over CkptYDAve which is the best competitor overall.

5.2.2 Absolute Overhead

In Fig. 8, we provide absolute values for the overhead of the strategies of Fig. 7, for two values of p_{fail}. The time spent for regular periodic checkpointing, or failure-free overhead, is represented in green; it is highest for CkptEachTask, as

Authorized licensed use limited to: University of Glasgow. Downloaded on August 19,2021 at 17:53:12 UTC from IEEE Xplore. Restrictions apply.

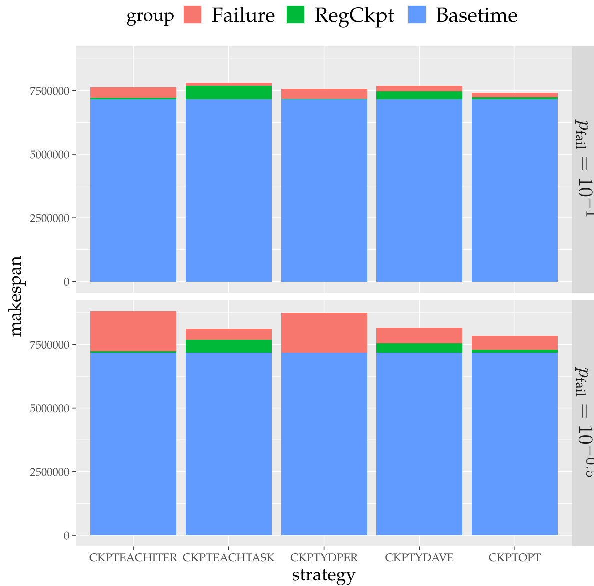


Fig. 8. Bar plots for absolute overhead (neuroscience).

expected. The failure-induced overhead (downtime, recovery and re-execution) is represented in red; it is higher for CkptEachIter and CkptYDPer. The details of the overheads are interesting: the optimal strategy is really able to trade-off checkpointing and failures; it spends roughly three times less checkpointing than the second best strategy CkptYDAve, for a similar failure-induced time. As observed in Fig. 8, with $p_{\text{fail}} = 10^{-1}$, the cumulated overhead (green and red) ranges from 3.37 to 8.25 percent, while for $p_{\text{fail}} = 10^{-0.5}$, it ranges from 8.64 to 18.73 percent .

5.2.3 Scalability

In Fig. 9, we study the scalability of the approach by varying the number of iterations from 10 to 1,000. We see that the variance is high at first but the performance of each strategy stabilizes from 100 iterations on.

5.2.4 Robustness

In Fig. 10, we study the robustness of the approach in front of variations in task execution times, which we draw from their Normal distributions as stated in Section 5.1.1. The results of CkptEachIter, CkptEachTask and CkptYDPer are similar with those in Fig. 7, with deterministic execution times. However, the results of CkptYDAve, which decides on the fly when to checkpoint, become better and close to the optimal strategy when p_{fail} gets smaller.

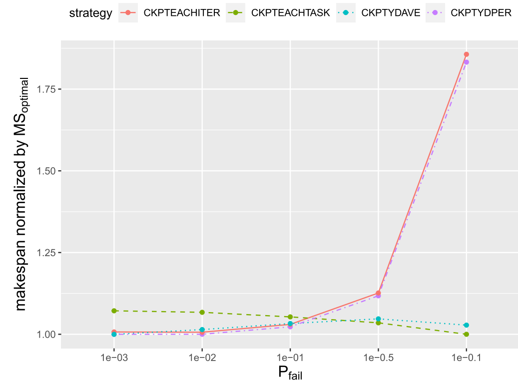


Fig. 10. Normalized performance overhead with stochastic execution times (neuroscience).

5.3 Results for the Synthetic Application

Results for the synthetic application scenario are reported in Figs. 11 and 12, with two values of n. When n increases from 10 to 20, CkptEachIter and CkptYDPer are closer to the optimal strategy when p_{fail} is small (for 10^{-3} and 10^{-2}), but further away when p_{fail} is large (for 10^{-1}, $10^{-0.5}$ and $10^{-0.1}$); on the contrary, CkptYDAve is closer to the optimal strategy for all p_{fail} values. Altogether, the results are quite similar to those obtained with the neuroscience application.

5.3.1 Impact of the Checkpoint Time

In the above experiment, we set checkpoint times as $c_i = \eta t_i$, where η is the coefficient of proportionality of checkpoint time over execution time for each task, and we conducted experiments with fixed $\eta = 0.1$. Here, we vary the value of η to study the impact of the checkpoint cost on the results. We conduct experiments with $\eta \in \{0.01, 0.05, 0.10, 0.15, 0.20\}$, thereby covering a wide range of scenarios (respectively low, balanced and high checkpointing cost). Lower checkpoint costs can be achieved with state-of-the-art in-memory or hierarchical checkpoint protocols [48], while larger checkpoint costs correspond to traditional protocols that save application data on remote disks. In Fig. 13, we provide performance

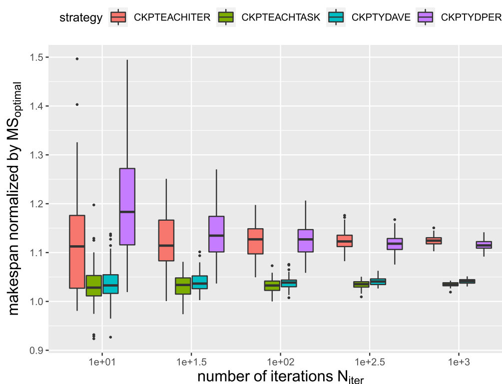


Fig. 9. Box plots for normalized performance overhead: varying the number of iterations (neuroscience, $p_{\text{fail}} = 10^{-0.5}$).

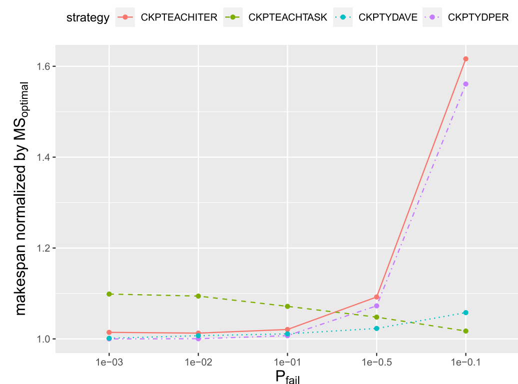


Fig. 11. Normalized performance overhead with different failure probabilities (synthetic, $n = 10$).

Authorized licensed use limited to: University of Glasgow. Downloaded on August 19,2021 at 17:53:12 UTC from IEEE Xplore. Restrictions apply.

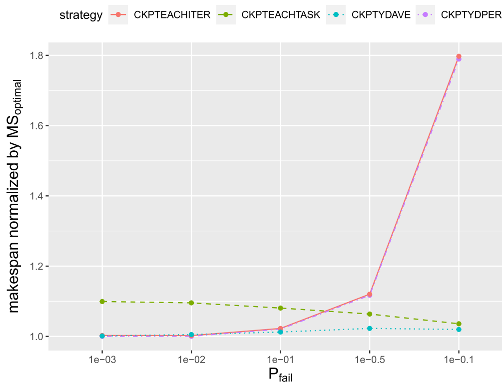


Fig. 12. Normalized performance overhead with different failure probabilities (synthetic, $n = 20$).

overhead with different values of η for the synthetic application with $n = 10$ when $p_{\text{fail}} = 10^{-0.5}$. This figure shows that CkptEachTask and CkptYDAve heuristics benefits from a small η: their performance gets close to optimal when the checkpoint cost becomes negligible. On the contrary, CkptEachIter and CkptYDPer heuristics have slightly worse performance compared to the optimal when η is reduced. For more details, please refer to the WSM.

5.4 Execution Time of the Dynamic Programming Algorithm

In Table 4, we report the execution time of the dynamic programming algorithm for all application scenarios. For the neuroscience application, the execution time is always below one minute. For the synthetic application, the execution time sharply increases when n doubles from 10 to 20, and reaches up to 10 minutes for $p_{\text{fail}} = 10^{-3}$. Table 5, explains why: the number of tasks in the optimal period, estimated by the upper bound of Theorem 5, becomes huge, while the actual number of tasks actually occurring in the optimal pattern is much lower. The bound of Theorem 5 is overly pessimistic, which increases the execution time of the dynamic programming algorithm. While 10 minutes for the algorithm is negligible in front of the 83 days of the application base time, one could easily decide to use CkptYDPer, the best reference strategy, instead of the optimal approach. Indeed, for $p_{\text{fail}} \leq 10^{-2}$, their overheads are of the same order.

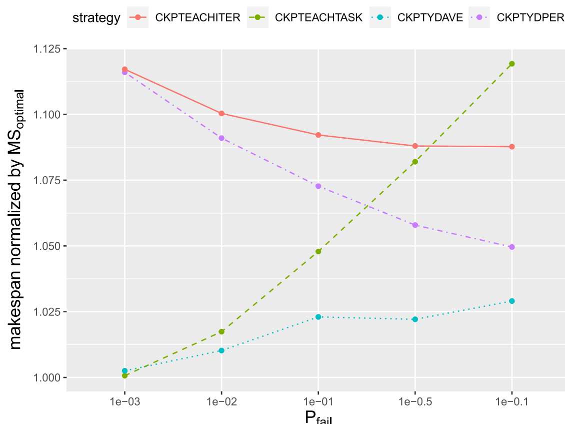


Fig. 13. Performance overhead with different values of η for the synthetic application with $n = 10$, $c_i = \eta t_i$ and $p_{\text{fail}} = 10^{-0.5}$.

TABLE 4
Execution Time (seconds) of the Dynamic Programming Algorithm

p_{fail}	10^{-3}	10^{-2}	10^{-1}	$10^{-0.5}$	$10^{-0.1}$
Neuroscience	5.46	0.88	0.21	0.21	0.22
Synthetic, $n = 10$	27.49	5.16	1.27	1.25	1.33
Synthetic, $n = 20$	550.89	95.49	46.75	43.37	46.07

5.5 Summary

Bar plots of the absolute overheads for all six applications variants and five values of p_{fail} are provided in the WSM. In summary, no reference heuristic is able to give close-to-optimal makespan for every value of p_{fail} : CkptYDPer is better with very few failures, while CkptEachTask and CkptYDAve are better when there are many failures. For these extreme scenarios, using the ad-hoc greedy heuristic is a good solution to trade-off the complexity of finding the solution with the gain in performance. However, in intermediary scenarios, the best reference heuristic can still increase the time to solution by 10 percent compared to the optimal one, showing the importance of computing the correct solution! Finally, when checkpoint costs can be kept very low, e.g., owing to checkpoint libraries such as VeloC [48], our experiments show that it is safe to use any heuristic that checkpoints sufficiently often, such as CkptEachTask or CkptYDAve, because their performance gets close to the optimal solution. Altogether, the best competitors are CkptYDPer and CkptYDAve, but none of them is always superior to the other, while our proposed optimal scheme enables us to carefully optimize the checkpoint pattern for all problem instances.

As for the relevance to exascale HPC scenarios, consider for instance the synthetic application (see Figs. 22 and 23 of the WSM), available online. When p_{fail} is low (10^{-3} to 10^{-1}), all methods are good, except CkptEachTask whose checkpoint overhead is prohibitive. When p_{fail} increases to $10^{-0.5}$, the overheads of CkptEachIter and CkptYDPer are twice larger than that of CkptYDAve, while CkptEachTask achieves intermediate results. Finally, for the highest value $p_{\text{fail}} = 10^{-0.1}$, the best competitor is CkptEachTask, followed by CkptYDAve, but the difference with the optimal solution gets much larger for all methods. How realistic is the latter value $p_{\text{fail}} = 10^{-0.1}$ for a future exascale HPC application running on 1 million cores, each with individual MTBF of 10 years? the application will experience a crash every 5

TABLE 5
Number of Tasks From the Bound of Theorem 5 and in the Optimal Pattern for the Neuroscience Application

p_{fail}	10^{-3}	10^{-2}	10^{-1}	$10^{-0.5}$	$10^{-0.1}$
Neuroscience: Bound	980	392	196	196	196
Neuroscience: Optimal pattern	14	7	7	7	7
Synthetic, $n = 10$: Bound	1,800	800	400	400	400
Synthetic, $n = 10$: Optimal pattern	150	50	50	20	10
Synthetic, $n = 20$: Bound	5,600	2,400	1,600	1,600	1,600
Synthetic, $n = 20$: Optimal pattern	200	180	20	20	20

Authorized licensed use limited to: University of Glasgow. Downloaded on August 19,2021 at 17:53:12 UTC from IEEE Xplore. Restrictions apply.

minutes; if an iteration lasts 4 minutes, this corresponds precisely to $p_{\text{fail}} = 10^{-0.1}$.

6 Conclusion

In this work, we have investigated checkpointing strategies for iterative applications. Each iteration is composed of a chain of tasks, and these tasks have different lengths and different checkpoint costs. Simple approaches would checkpoint either every task, or the last task at the end of each iteration. An approach inspired by the Young/Daly formula works for P_{YD} seconds, where P_{YD} comes from the Young/Daly formula with a checkpoint cost averaged over all tasks, and then checkpoints as soon as possible (and repeats). Another approach inspired by the Young/Daly formula selects the task with lowest checkpoint cost and checkpoints every p^{th} instance of that task, where p is computed so that the period length approximately obeys the formula. But what is the optimal strategy? The main contributions of this paper are threefold: (i) we have shown that there exists a periodic strategy that is optimal; (ii) we have provided a dynamic-programming algorithm that computes the optimal period; and (iii) we have shown through a set of experiments that the gains over the other approaches are significant, and that the optimal strategy is the only one achieving a robust solution for all problem instances cases. Given the importance of iterative applications in HPC, we expect that these contributions will greatly improve the deployment of resilient solutions at scale.

Future work will be devoted to dealing with iterative applications whose iterations are composed of a Directed Acyclic Graph (DAG) of tasks, not just a linear chain. Such applications are ubiquitous in real-time systems. However, the mere fact that several tasks may execute concurrently on the platform raises very complicated challenges [6], [49], and most likely only heuristic (suboptimal) algorithms will be obtained.

Acknowledgments

The authors would like to thank the reviewers for their comments and suggestions, which greatly helped improve the final version of the article.

References

[1] Top500, "Top 500 supercomputer sites," Nov. 2020. [Online]. Available: https://www.top500.org/lists/2020/11/
[2] T. Herault and Y. Robert, *Fault-Tolerance Techniques for High-Performance Computing*, Cham, Switzerland: Springer, 2015.
[3] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Commun. ACM*, vol. 17, no. 9, pp. 530–531, Sep. 1974.
[4] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Gener. Comput. Syst.*, vol. 22, no. 3, pp. 303–312, Feb. 2006.
[5] S. Toueg and O. Babaoğlu, "On the optimum checkpoint selection problem," *SIAM J. Comput.*, vol. 13, no. 3, pp. 630–649, Aug. 1984.
[6] L. Han, L.-C. Canon, H. Casanova, Y. Robert, and F. Vivien, "Checkpointing workflows for fail-stop errors," *IEEE Trans. Comput.*, vol. 67, no. 8, pp. 1105–1120, Aug. 2018.
[7] Y. Du, L. Marchal, G. Pallez, and Y. Robert, "Robustness of the Young/Daly formula for stochastic iterative applications," in *49th Int. Conf. Parallel Process.*, 2020, 1–11.
[8] G. Aupy, A. Benoit, H. Casanova, and Y. Robert, "Checkpointing strategies for scheduling computational workflows," *Int. J. Netw. Comput.*, vol. 6, no. 1, pp. 2–26, Jan. 2016.
[9] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed., Philadelphia, PA, USA: Soc. Ind. Appl. Math., 2003.
[10] S. P. Frankel, "Convergence rates of iterative treatments of partial differential equations," *Math. Tables Other Aids Comput.*, vol. 4, no. 30, pp. 65–75, Apr. 1950.
[11] D. Young, "Iterative methods for solving partial difference equations of elliptic type," *Trans. Amer. Math. Soc.*, vol. 76, no. 1, pp. 92–111, Jan. 1954.
[12] Y. Saad and M. H. Schultz, "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM J. Sci. Statist. Comput.*, vol. 7, no. 3, pp. 856–869, Mar. 1986.
[13] M. H. Gutknecht, "Variants of BICGSTAB for matrices with complex spectrum," *SIAM J. Sci. Comput.*, vol. 14, no. 5, pp. 1020–1033, May 1993.
[14] S. C. Eisenstat, H. C. Elman, and M. H. Schultz, "Variational iterative methods for nonsymmetric systems of linear equations," *SIAM J. Numer. Anal.*, vol. 20, no. 2, pp. 345–357, Feb. 1983.
[15] E. Agullo, L. Giraud, A. Guermouche, J. Roman, and M. Zounon, "Towards resilient parallel linear Krylov solvers: Recover-restart strategies," Ph.D. dissertation, INRIA, Rocquencourt, France, 2013.
[16] E. Agullo, L. Giraud, A. Guermouche, J. Roman, and M. Zounon, "Numerical recovery strategies for parallel resilient Krylov linear solvers," *Numer. Linear Algebra Appl.*, vol. 23, no. 5, pp. 888–905, May 2016.
[17] J. Langou, Z. Chen, G. Bosilca, and J. Dongarra, "Recovery patterns for iterative methods in a parallel unstable environment," *SIAM J. Sci. Comput.*, vol. 30, no. 1, pp. 102–116, Jan. 2008.
[18] W. L. Oberkampf and C. J. Roy, *Verification and Validation in Scientific Computing*. Cambridge, U.K.: Cambridge Univ. Press, 2010.
[19] C. J. Roy and W. L. Oberkampf, "A comprehensive framework for verification, validation, and uncertainty quantification in scientific computing," *Comput. Methods in Appl. Mech. Eng.*, vol. 200, no. 25–28, pp. 2131–2144, 2011.
[20] O. Sertel, J. Kong, H. Shimada, Ü. V. Çatalyürek, J. H. Saltz, and M. N. Gurcan, "Computer-aided prognosis of neuroblastoma on whole-slide images: Classification of stromal development," *Pattern Recognit.*, vol. 42, no. 6, pp. 1093–1103, 2009.
[21] F. Guirado, A. Ripoll, C. Roig, and E. Luque, "Optimizing latency under throughput requirements for streaming applications on cluster execution," in *Proc. Cluster Comput.*, Sep. 2005, pp. 1–10.
[22] K. Knobe, J. M. Rehg, A. Chauhan, R. S. Nikhil, and U. Ramachandran, "Scheduling constrained dynamic applications on clusters," in *Proc. ACM/IEEE Conf. Supercomput.*, New York, NY, USA, 1999, pp. 46–es.
[23] A. Choudhary *et al.*, "Design, implementation and evaluation of parallel pipelined STAP on parallel computers," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 36, no. 2, pp. 655–662, Apr. 2000.
[24] T. D. R. Hartley, A. R. Fasih, C. A. Berdanier, F. Ozguner, and Ü. V. Çatalyürek, "Investigating the use of GPU-accelerated nodes for SAR image formation," in *Proc. IEEE Int. Conf. Cluster Comput. Workshops*, 2009, pp. 1–8.
[25] C. Chekuri, W. Hasan, and R. Motwani, "Scheduling problems in parallel query optimization," in *Proc. 14th ACM SIGACT-SIGMOD-SIGART Symp. Princ. Database Syst.*, New York, NY, USA: 1995, pp. 255–265.
[26] A. Rowe, D. Kalaitzopoulos, M. Osmond, M. Ghanem, and Y. Guo, "The discovery net system for high throughput bioinformatics," *Bioinformatics*, vol. 19, no. Suppl 1, pp. i225–i231, 2003.
[27] F. Guirado, A. Ripoll, C. Roig, A. Hernandez, and E. Luque, "Exploiting throughput for pipeline execution in streaming image processing applications," in *Proc. Euro-Par Parallel Process.*, 2006, pp. 1095–1105.
[28] E. Deelman, J. Blythe, Y. Gil, and C. Kesselman, "Workflow management in GriPhyN," *Grid Resource Management: State of the Art and Future Trends*, pp. 99–116, 2004.
[29] J. Kim, Y. Gil, and M. Spraragen, "A knowledge-based approach to interactive workflow composition," in *Proc. 14th Int. Conf. Automat. Planning Scheduling*, 2004.
[30] Z. Chen, "Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods," *ACM SIGPLAN Notices*, vol. 48, no. 8, pp. 167–176, 2013.

Authorized licensed use limited to: University of Glasgow. Downloaded on August 19,2021 at 17:53:12 UTC from IEEE Xplore. Restrictions apply.

[31] D. Tao *et al.*, “New-sum: A novel online ABFT scheme for general iterative methods,” in *Proc. 25th ACM Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2016, pp. 43–55.
[32] P. G. Bridges, K. B. Ferreira, M. A. Heroux, and M. Hoemmen, “Fault-tolerant linear solvers via selective reliability,” 2012, *arXiv:1206.1390*.
[33] J. Elliott, M. Hoemmen, and F. Mueller, “Evaluating the impact of SDC on the GMRES iterative solver,” in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, 2014, pp. 1193–1202.
[34] P. Sao and R. Vuduc, “Self-stabilizing iterative solvers,” in *Proc. Workshop Latest Adv. Scalable Algorithms Large-Scale Syst.*, 2013, pp. 1–8.
[35] M. E. Ozturk, G. Agrawal, Y. Li, and C.-S. Chou, “Handling soft errors in Krylov subspace methods by exploiting their numerical properties,” 2020. [Online]. Available: https://easychair.org/publications/preprint/BmZG
[36] Z. Chen, “Algorithm-based recovery for iterative methods without checkpointing,” in *Proc. 20th Int. Symp. High Perform. Distrib. Comput.*, 2011, pp. 73–84.
[37] D. Tao, S. Di, X. Liang, Z. Chen, and F. Cappello, “Improving performance of iterative methods by lossy checkponting,” in *Proc. 27th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2018, pp. 52–65.
[38] C. Pachajoa, M. Levonyak, and W. N. Gansterer, “Extending and evaluating fault-tolerant preconditioned conjugate gradient methods,” in *Proc. IEEE/ACM 8th Workshop Fault Tolerance for HPC at eXtreme Scale*, 2018, pp. 49–58.
[39] C. Pachajoa, M. Levonyak, W. N. Gansterer, and J. L. Träff, “How to make the preconditioned conjugate gradient method resilient against multiple node failures,” in *Proc. 48th Int. Conf. Parallel Process.*, 2019, pp. 1–10.
[40] C. Pachajoa, C. Pacher, M. Levonyak, and W. N. Gansterer, “Algorithm-based checkpoint-recovery for the conjugate gradient method,” in *Proc. 49th Int. Conf. Parallel Process.*, 2020, pp. 1–11.
[41] K. M. Chandy and L. Lamport, “Distributed snapshots: Determining global states of distributed systems,” *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, Jan. 1985.
[42] K. Ferreira *et al.*, “Evaluating the Viability of Process Replication Reliability for Exascale Systems,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, pp. 1–12.
[43] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, “Toward exascale resilience: 2014 update,” *Supercomput. Front. Innov.*, vol. 1, no. 1, pp. 5–18, 2014.
[44] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien, “Checkpointing strategies for parallel jobs,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, pp. 1–11.
[45] Y. Du, “Code for simulations,” 2020. [Online]. Available: https://github.com/Yishu0604/Optimal-Checkpointing-Strategies-for-Itera tive-Applications
[46] Y. Huo *et al.*, “3D whole brain segmentation using spatially localized atlas network tiles,” *NeuroImage*, vol. 194, pp. 105–119, Jul. 2019.
[47] A. Gainaru, B. Goglin, V. Honoré, and G. Pallez, “Profiles of upcoming HPC applications and their impact on reservation strategies,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 5, pp. 1178–1190, May 2021.
[48] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello, “VeloC: Towards high performance adaptive asynchronous checkpointing at large scale,” in *Proc. Int. Parallel Distrib. Process. Symp.*, 2019, pp. 911–920.
[49] L. Han, V. Le F èvre, L.-C. Canon, Y. Robert, and F. Vivien, “A generic approach to scheduling and checkpointing workflows,” *Int. J. High Perform. Comput. Appl.*, vol. 33, no. 6, pp. 1255–1274, Aug. 2019.

Yishu Du is currently working toward the dual PhD degrees with Computer Science Laboratory LIP, ENS Lyon and the School of Mathematics Sciences, Tongji University, Shanghai, China. His research interests include resilience and scheduling problems for large-scale platforms.

Loris Marchal graduated in computer sciences and the PhD from the École Normale Supérieure de Lyon, ENS Lyon, France, in 2006. He is currently a CNRS researcher with the LIP Laboratory of ENS Lyon. His research interests include parallel computing and scheduling for modern computing platforms, memory-aware and data-aware scheduling.

Guillaume Pallez is currently a tenured researcher with Inria Bordeaux – Sud-Ouest. His research interests include algorithm design and scheduling techniques for parallel and distributed platforms, such as data-aware scheduling and stochastic scheduling. He was the Technical Program vice-chair of SC’17, workshop chair of SC’18, and algorithm track vice-chair of ICPP’18. He was the recipient of 2019 IEEE TCHPC Early Career Researcher Award.

Yves Robert (Fellow, IEEE) is currently a full professor with Computer Science Laboratory LIP, ENS Lyon. He is a Senior Member of Institut Universitaire de France. He has been awarded the 2014 IEEE TCSC Award for Excellence in Scalable Computing, 2016 IEEE TCPP Outstanding Service Award, and 2020 IEEE CS Charles Babbage Award. He holds a Visiting Scientist position at the Innovative Computing Laboratory at University of Tennessee, Knoxville, since 2011. His research interests include scheduling techniques, parallel algorithms and resilient approaches for large-scale platforms. For more information please visit http://graal.ens-lyon.fr/~yrobert/

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: University of Glasgow. Downloaded on August 19,2021 at 17:53:12 UTC from IEEE Xplore. Restrictions apply.

WEB SUPPLEMENTARY MATERIAL Optimal Checkpointing Strategies for Iterative Applications

Yishu Du, Loris Marchal, Guillaume Pallez (Aupy) and Yves Robert

1 THE GCR APPLICATION

The second real-life application scenario is GCR, a Krylov Subspace method [?EES83] solving the m-dimensional sparse linear system $A\boldsymbol{x} = \boldsymbol{b}$. Each iteration of the method is divided into n sub-iterations, whose computational and memory requirements increase from one sub-iteration to the next. The common way to control the number of iterative steps within an acceptable range is to adopt a restart strategy [?Young1980,?Simoncini2000,?Saad2003], that is, to fix a small value n (usually much less than m, such as 10, 20, etc.). If the last n-th sub-iteration does not lead to convergence, then the approximate solution $\boldsymbol{x}^n$ is used as the initial value of a new iteration, and the GCR method is restarted. The process is repeated until a satisfactory approximate solution is found, as detailed in Algorithm 1.

Algorithm 1 GCR(n)

1: $\boldsymbol{x}^0$, $\boldsymbol{r}^0 = A\boldsymbol{x}^0 - \boldsymbol{b}$, $\boldsymbol{p}^0 = P^{-1}\boldsymbol{r}^0$, $\boldsymbol{q}^0 = A\boldsymbol{p}^0$
2: **for** $k = 1, 2, \dots,$ until convergence **do**
3: **for** $i = 0, 1, \dots, n-1$ **do**
4: $\beta = \frac{(\boldsymbol{r}^i, \boldsymbol{q}^i)}{(\boldsymbol{q}^i, \boldsymbol{q}^i)}$ ▷ $4m - 1$
5: $\boldsymbol{x}^{i+1} = \boldsymbol{x}^i + \beta \boldsymbol{p}^i$ ▷ $2m$
6: $\boldsymbol{r}^{i+1} = \boldsymbol{r}^i + \beta \boldsymbol{q}^i$ ▷ $2m$
7: **if** $\|\boldsymbol{r}^{i+1}\| \le \varepsilon$ **then**
8: exit
9: $\boldsymbol{e} = P^{-1}\boldsymbol{r}^{i+1}$ ▷ $3m - 1$
10: $\tilde{\boldsymbol{e}} = A\boldsymbol{e}$ ▷ $2nz(A) - 1$
11: **for** $l = 0, 1, \dots, i$ **do**
12: $\alpha_l = \frac{(\tilde{\boldsymbol{e}}, \boldsymbol{q}^l)}{(\boldsymbol{q}^l, \boldsymbol{q}^l)}$ ▷ $(i+1)(4m-1)$
13: $\boldsymbol{p}^{i+1} = \boldsymbol{e} + \sum_{l=0}^{i} \alpha_l \boldsymbol{p}^l$ ▷ $m + (i+1)m$
14: $\boldsymbol{q}^{i+1} = \tilde{\boldsymbol{e}} + \sum_{l=0}^{i} \alpha_l \boldsymbol{q}^l$ ▷ $m + (i+1)m$
15: $[\boldsymbol{x}^0, \boldsymbol{r}^0, \boldsymbol{p}^0, \boldsymbol{q}^0] \leftarrow [\boldsymbol{x}^n, \boldsymbol{r}^n, \boldsymbol{p}^n, \boldsymbol{q}^n]$

• *Yishu Du is with Tongji University, Shanghai, China. Loris Marchal and Yves Robert are with LIP, École Normale Supérieure de Lyon, CNRS & Inria, France. Guillaume Pallez (Aupy) is with Inria & Université de Bordeaux, France. Yishu Du is also with LIP, ENS Lyon. Yves Robert is also with University of Tennessee Knoxville, USA. Contact: Yves.Robert@ens-lyon.fr.*

We consider an iterative application composed of $N = 10^3$ iterations, and each iteration (outer loop k) has either $n = 10$ or $n = 20$ tasks. Each task corresponds to one sub-iteration of the loop on i. The number of non-zero elements of sparse matrix A is denoted as *nz(A)*. We assume that $m = 100000$, $nz(A) = 27m$, and the preconditioner matrix P is a diagonal matrix in the simulation. We pick (somewhat arbitrarily) 27 because it is the size of a $3 \times 3 \times 3$ cube for a neighborhood of interactions, so the matrix has 27 diagonals (3D-stencil for Jacobi or Gauss-Seidel, typically). The number of floating-point operations for task i is $f_i = (6m-1)i + 19m - 4 + 2nz(A)$, see Table 1.

We use incremental checkpointing [?Agarwal04, ?Naksinehaboon08] for GCR(n). The vectors that need to be saved if we checkpoint after task i and the corresponding size c_i of the checkpoint are detailed in Table 1. Similarly, the vectors that need to be recovered if we experiment a failure task a_i and the corresponding size r_i of the recovery are also detailed in Table 1. We observe that c_i remains constant and small for all i, owing to the incremental checkpointing technique. Of course, the larger i, the more vectors to recover, and r_i is increasing. Since the checkpoint cost of each task is constant, the CKPTYDPER heuristic chooses the task with minimum recovery size $r_{\min}$. Only the result of this task will (possibly) be checkpointed. Then it also uses the Young-Daly formula to compute how many iterations to include in between two checkpoints, namely $\max\left(1, \text{round}\left(\frac{w_c}{T}\right)\right)$.

We consider here that the computing platform has unit speed $s = 1$, so that $t_i = f_i/s = f_i$. In order to test different scenarios for the relative cost of checkpoint compared to computations, we define the Communication-to-Computation Ratio (CCR) as ratio between the cost of communicating one byte to the cost of computing one flop. With the choice $s = 1$, the CCR is exactly the inverse of the bandwidth. Hence, from the size of the memory to checkpoint M_i, we compute the time for a checkpoint: $c_i = M_i \times$ CCR. Similarly, we let $r_i = M'_i \times$ CCR, where the size of the memory to recover is M'_i.

We conducted experiments with CCR $\in \{0.1, 0.2, 1, 5, 10\}$, thereby covering a wide range of scenarios (respectively low, balanced and high communication cost).

Results for the GCR application are reported in Figures 1 to 10, with two values of n and five values of CCR.

Task	Floating point operations f_i	Vectors to checkpoint	M_i	Vectors to recover	M'_i
a_0	$19m-4+2nz(A)$	$\boldsymbol{p}^1,\boldsymbol{q}^1,\boldsymbol{r}^1,\boldsymbol{x}^1$	$4m$	$\boldsymbol{p}^0,\boldsymbol{p}^1,$ $\boldsymbol{q}^0,\boldsymbol{q}^1,$ $\boldsymbol{r}^1,\boldsymbol{x}^1$	$6m$
a_1	$(6m-1)+19m-4+2nz(A)$	$\boldsymbol{p}^2,\boldsymbol{q}^2,\boldsymbol{r}^2,\boldsymbol{x}^2$	$4m$	$\boldsymbol{p}^0,\boldsymbol{p}^1,\boldsymbol{p}^2,$ $\boldsymbol{q}^0,\boldsymbol{q}^1,\boldsymbol{q}^2,$ $\boldsymbol{r}^2,\boldsymbol{x}^2$	$8m$
...	...	...	...	...	...
a_{n-2}	$(6m-1)(n-2)+19m-4+2nz(A)$	$\boldsymbol{p}^{n-1},\boldsymbol{q}^{n-1},\boldsymbol{r}^{n-1},\boldsymbol{x}^{n-1}$	$4m$	$\boldsymbol{p}^0,\ldots,\boldsymbol{p}^{n-1},$ $\boldsymbol{q}^0,\ldots,\boldsymbol{q}^{n-1},$ $\boldsymbol{r}^{n-1},\boldsymbol{x}^{n-1}$	$(2n+2)m$
a_{n-1}	$(6m-1)(n-1)+19m-4+2nz(A)$	$\boldsymbol{p}^0,\boldsymbol{q}^0,\boldsymbol{r}^0,\boldsymbol{x}^0$	$4m$	$\boldsymbol{p}^0,$ $\boldsymbol{q}^0,$ $\boldsymbol{r}^0,\boldsymbol{x}^0$	$4m$

Table 1: Tasks composing the GCR application.

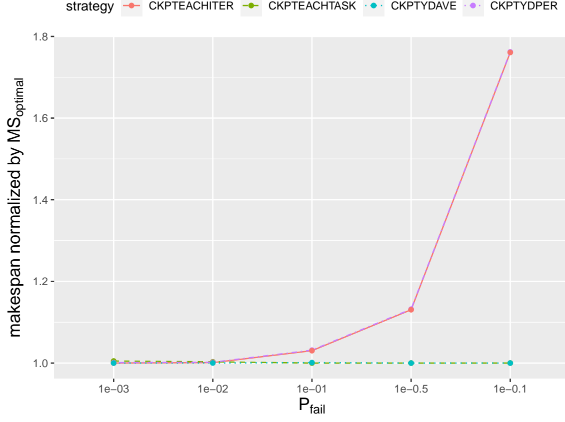


Figure 1: Performance overhead with different failure probabilities for GCR(n), with n=10 and CCR = 0.1.

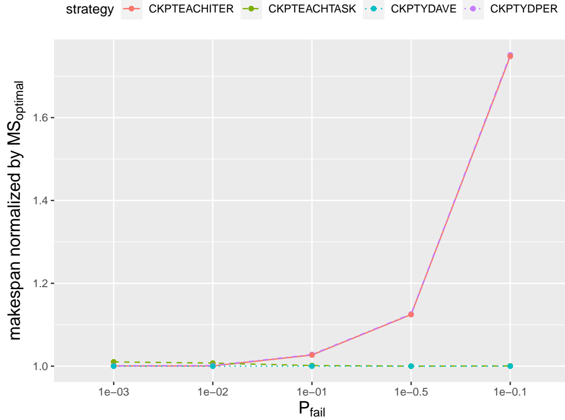


Figure 2: Performance overhead with different failure probabilities for GCR(n), with n=10 and CCR = 0.2.

When the CCR increases, CKPTEACHITER and CKPTYDPER are further away from the optimal strategy when p_{fail} is small (for 10^{-3} and 10^{-2}), while both strategies are closer to the optimal strategy when p_{fail} is large (for 10^{-1}, $10^{-0.5}$ and $10^{-0.1}$); CKPTEACHTASK and CKPTYDAVE are further away from the optimal strategy for all p_{fail} values.

In addition, when n increases from 10 to 20, CKPTEACHITER, CKPTYDPER and CKPTYDAVE are closer to the optimal strategy, while CKPTEACHTASK keeps a high overhead (with a ratio up to 1.5 to the optimal.

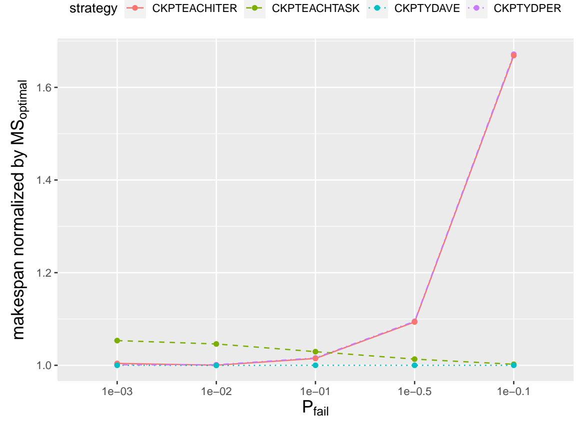


Figure 3: Performance overhead with different failure probabilities for GCR(n), with n=10 and CCR = 1.

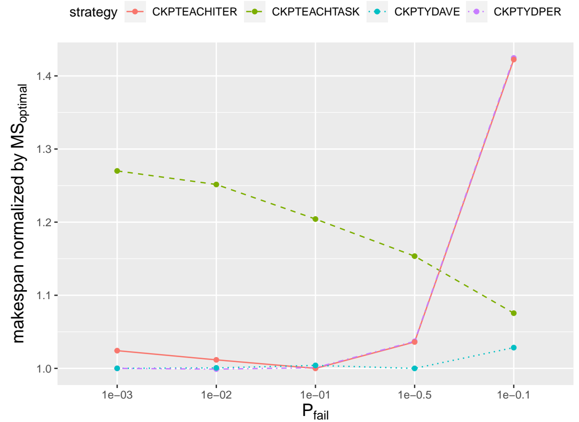


Figure 4: Performance overhead with different failure probabilities for GCR(n), with n=10 and CCR = 5.

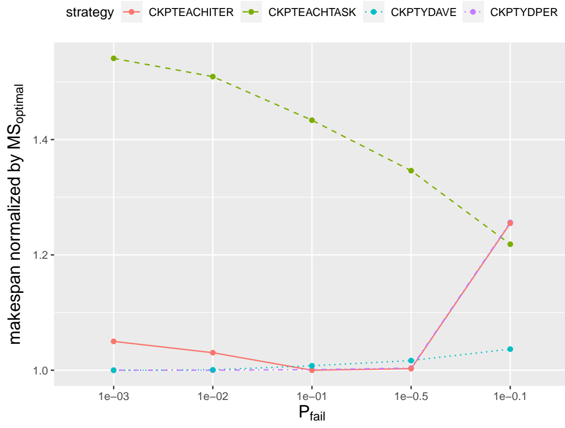


Figure 5: Performance overhead with different failure probabilities for GCR(n), with n=10 and CCR = 10.

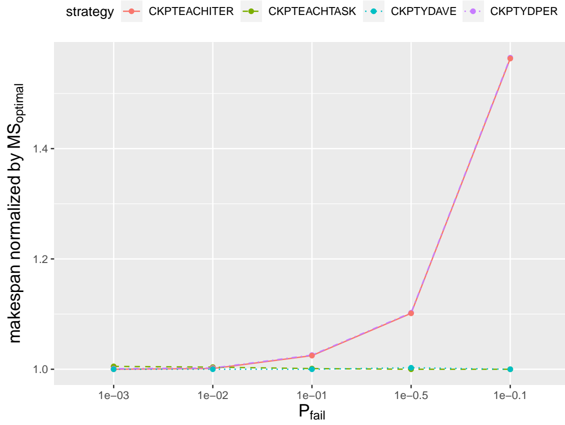


Figure 6: Performance overhead with different failure probabilities for GCR(n), with n=20 and CCR = 0.1.

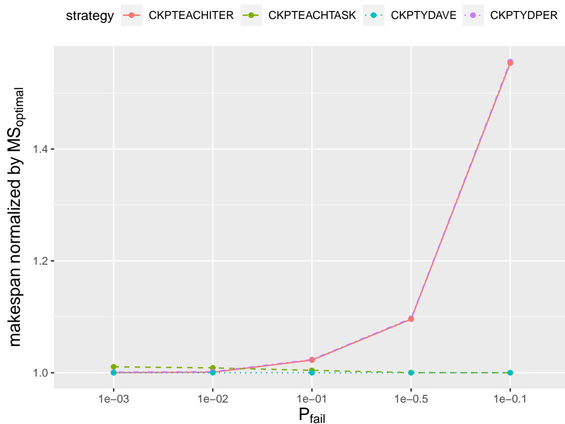


Figure 7: Performance overhead with different failure probabilities for GCR(n), with n=20 and CCR = 0.2.

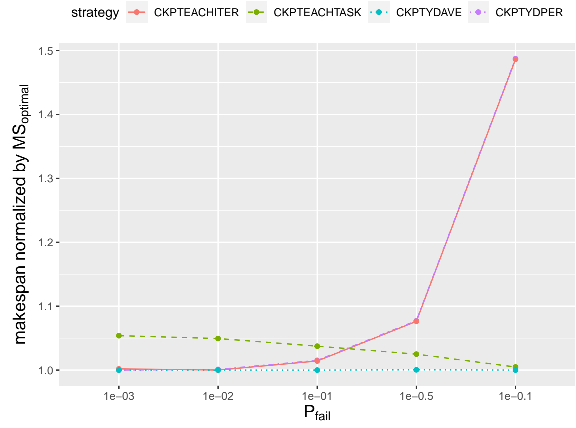


Figure 8: Performance overhead with different failure probabilities for GCR(n), with n=20 and CCR = 1.

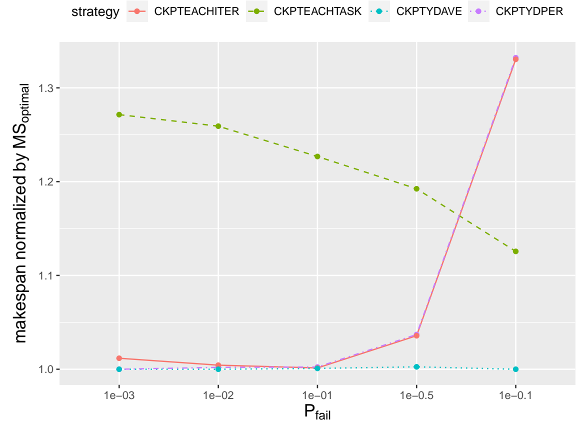


Figure 9: Performance overhead with different failure probabilities for GCR(n), with n=20 and CCR = 5.

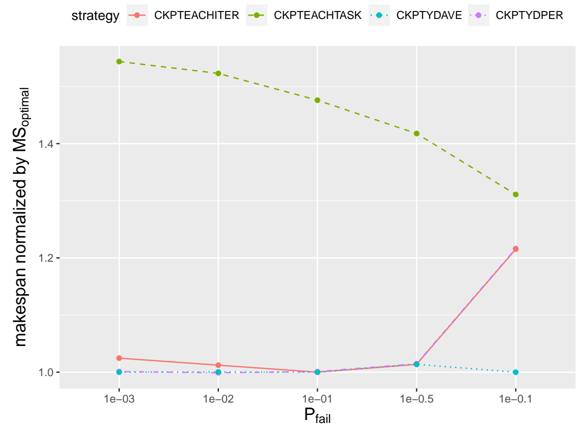


Figure 10: Performance overhead with different failure probabilities for GCR(n), with n=20 and CCR = 10.

2 Additional experiments with synthetic applications

2.1 Impact of Communication-to-Computation Ratio (CCR)

An important factor that influences the performance of checkpointing strategies, and more precisely of the checkpointing and recovery overheads, is the data-intensiveness of the application. For the synthetic application, in order to test the impact of the correlation between checkpoint costs and task running times on the strategies, we let the checkpoint time move from dependent to independent of the task running time (see Figures 11 and 12).

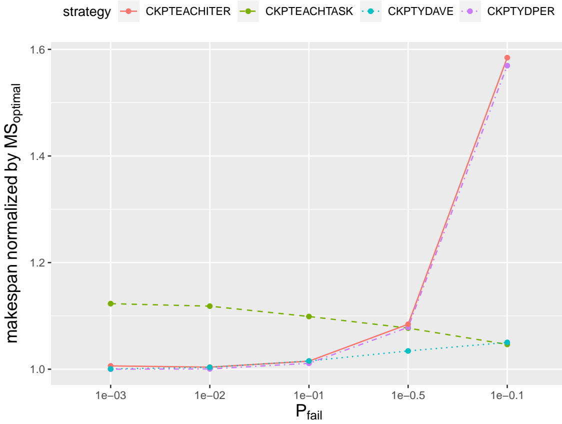


Figure 11: Performance overhead with different failure probabilities for the synthetic application with $n = 10$ and c_i drawn in Uniform$[10, 100]$.

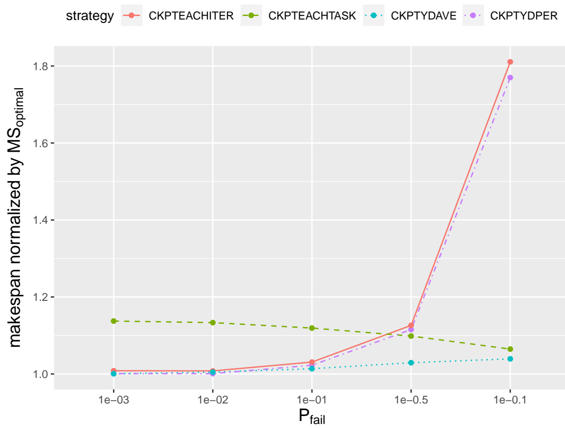


Figure 12: Performance overhead with different failure probabilities for the synthetic application with $n = 20$ and c_i drawn in Uniform$[10, 100]$.

2.2 Impact of checkpoint time

In this section, we vary the checkpointing cost of each task in order to study its influence on the results. We have two settings for the checkpointing cost. First, we consider that the checkpointing time is proportional to the task execution time, as assumed in the main paper: $C_i = \eta t_i$. We have previously considered $\eta = 0.1$, and we conduct here experiments with $\eta \in \{0.01, 0.05, 0.10, 0.15, 0.20\}$ thereby covering a wide range of scenarios (respectively low, balanced and high checkpointing cost). Lower checkpoint costs can be achieved with state-of-the-art in-memory or hierarchical checkpoint protocols [?veloc2019], while larger checkpoint costs correspond to traditional protocols that save application data on remote disks.

Second, as in the previous section of this supplementary material, we set checkpoint times taken uniformly at random in some interval. We now set checkpoint times taken in Uniform$[100\eta, 1000\eta]$ to also cover wider range of scenarios (recall that the average task length is 550 seconds).

Results for varying the value of η are reported in Figures 13 to 17 with the five p_{fail} values and the two checkpoint settings (proportional or uniform).

We first observe that the distribution of checkpoint times (proportional to the task execution time or uniform) has very little impact of the results. However, the heuristics do behave very differently when varying checkpoint time, and their behavior also depends on the failure probability.

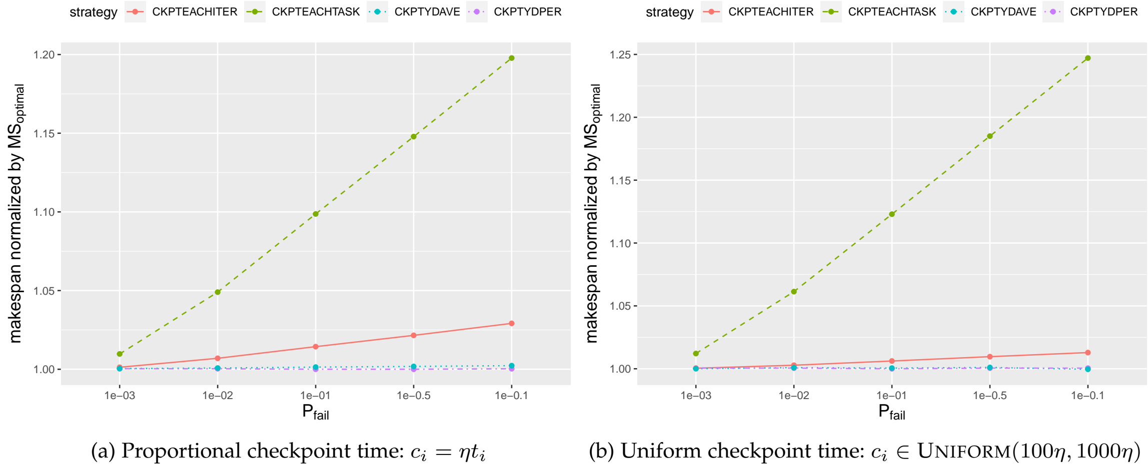
When p_{fail} is small (10^{-3} or 10^{-2}), we note that the CkptEachIter heuristic (in red) performs worse and further away from the optimal strategy when η gets larger; for larger values of p_{fail}, its performance is slightly improved when η gets larger.

The performance of the CkptEachTask (in green) heuristic becomes worse and further away from the optimal strategy when η gets larger for all p_{fail} values.

When p_{fail} is very small (10^{-3}), the performance CkptYDAve heuristic (in blue) is quite close with the optimal strategy for all values of η; when p_{fail} is small (10^{-2} or 10^{-1}), it is improved when η gets smaller. Finally, when p_{fail} is large ($10^{-0.5}$ or $10^{-0.1}$), the performance of CkptYDAve slightly varies with η, but it is almost optimal for small values of η.

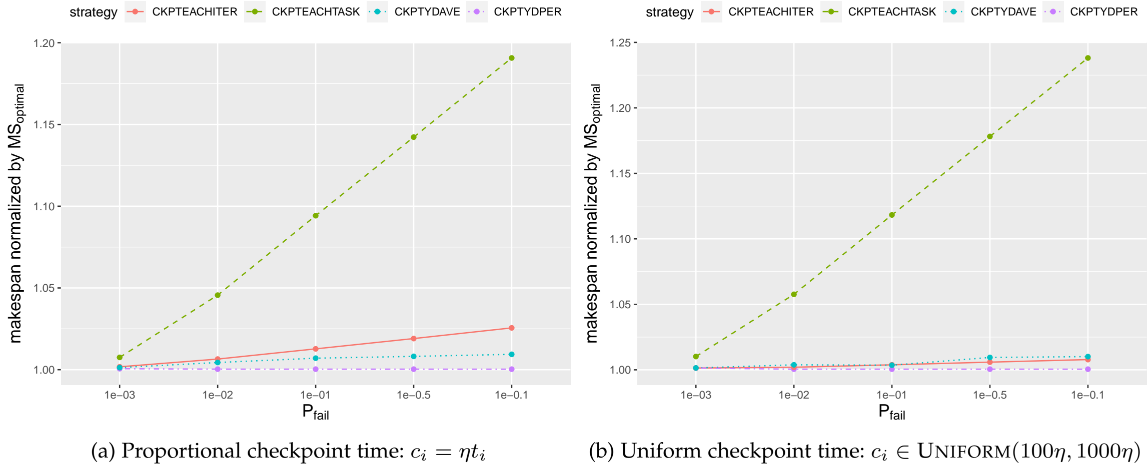
The performance of the CkptYDPer heuristic (in purple) is very close to the optimal strategy when p_{fail} is small (10^{-3} or 10^{-2}). For larger failure probabilities, its performance is improved when η gets larger.

Overall, heuristics CkptEachTask and CkptYDAve benefits from a very small checkpoint time: when the overhead due to checkpointing is negligible, these heuristics reach an optimal makespan.



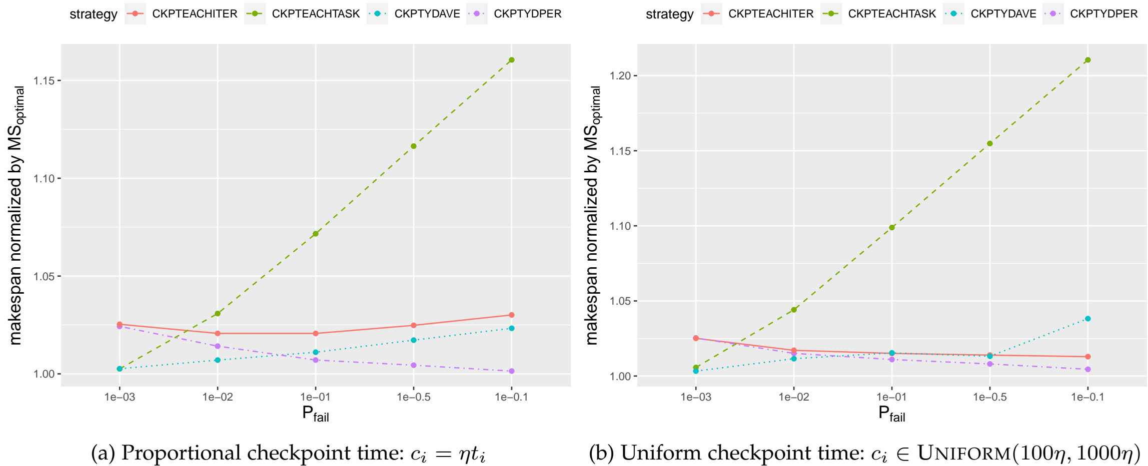
(a) Proportional checkpoint time: $c_i = \eta t_i$ (b) Uniform checkpoint time: $c_i \in \text{UNIFORM}(100\eta, 1000\eta)$

Figure 13: Performance overhead with different values of η for the synthetic application with $n = 10$ and $p_{\text{fail}} = 10^{-3}$.



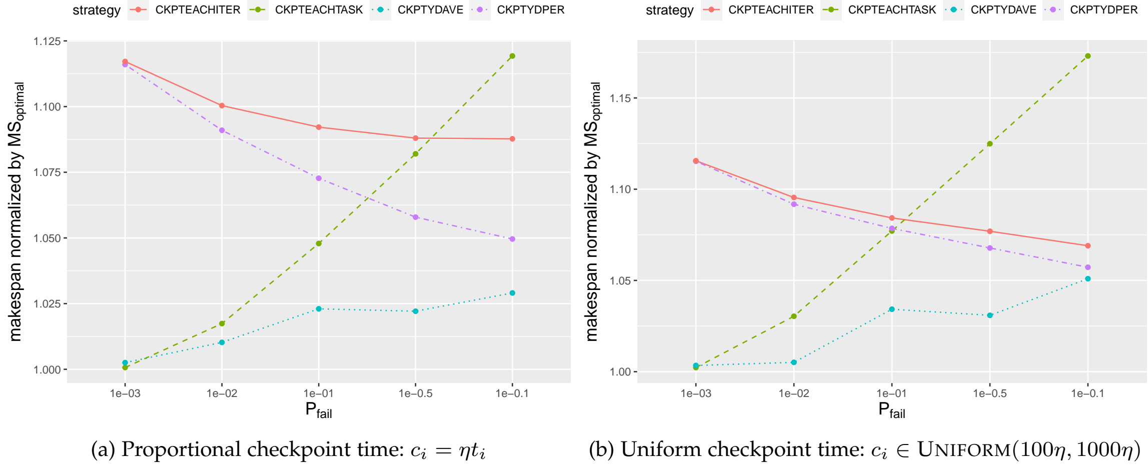
(a) Proportional checkpoint time: $c_i = \eta t_i$ (b) Uniform checkpoint time: $c_i \in \text{UNIFORM}(100\eta, 1000\eta)$

Figure 14: Performance overhead with different values of η for the synthetic application with $n = 10$ and $p_{\text{fail}} = 10^{-2}$.



(a) Proportional checkpoint time: $c_i = \eta t_i$ (b) Uniform checkpoint time: $c_i \in \text{UNIFORM}(100\eta, 1000\eta)$

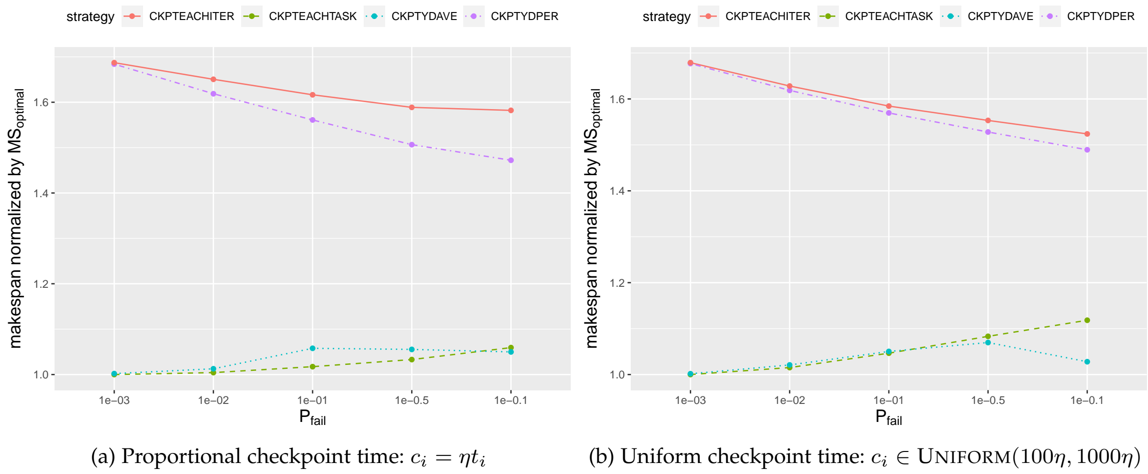
Figure 15: Performance overhead with different values of η for the synthetic application with $n = 10$ and $p_{\text{fail}} = 10^{-1}$.



(a) Proportional checkpoint time: $c_i = \eta t_i$

(b) Uniform checkpoint time: $c_i \in \text{UNIFORM}(100\eta, 1000\eta)$

Figure 16: Performance overhead with different values of η for the synthetic application with $n = 10$ and $p_{\text{fail}} = 10^{-0.5}$.



(a) Proportional checkpoint time: $c_i = \eta t_i$

(b) Uniform checkpoint time: $c_i \in \text{UNIFORM}(100\eta, 1000\eta)$

Figure 17: Performance overhead with different values of η for the synthetic application with $n = 10$ and $p_{\text{fail}} = 10^{-0.1}$.

3 ABSOLUTE OVERHEAD FOR ALL SCENARIOS

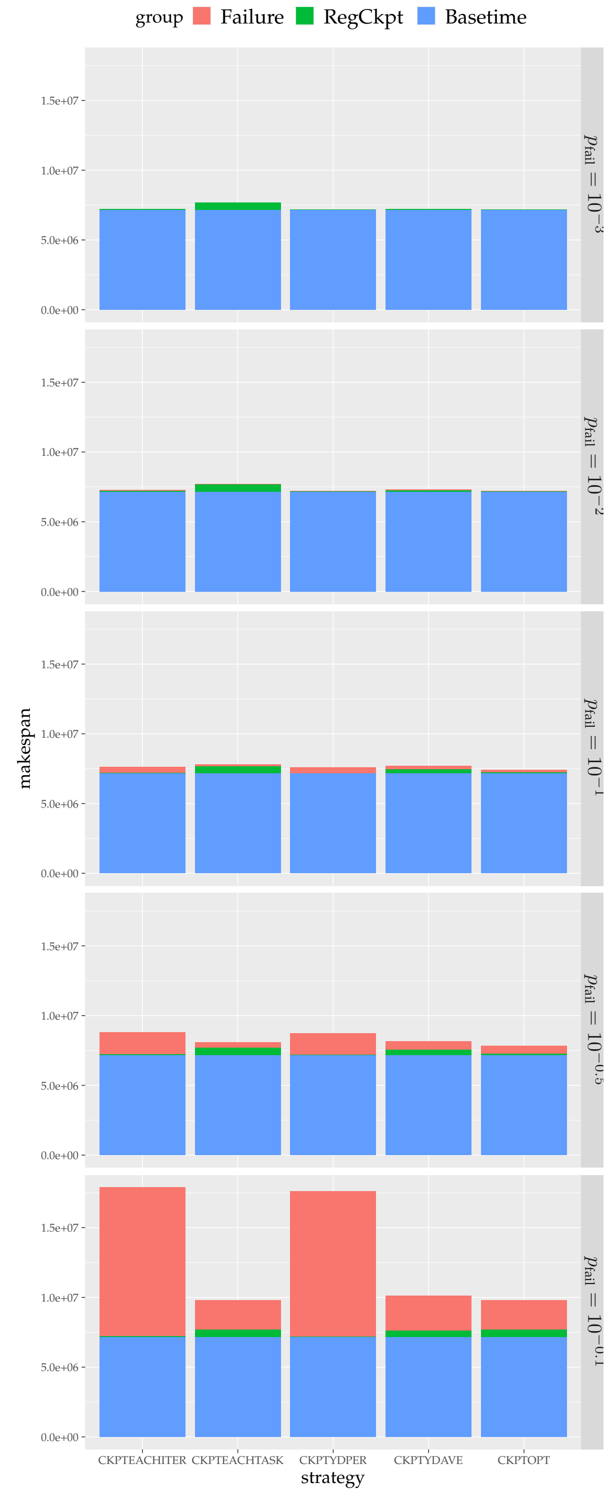


Figure 18: Bar plots for absolute overhead (neuroscience).

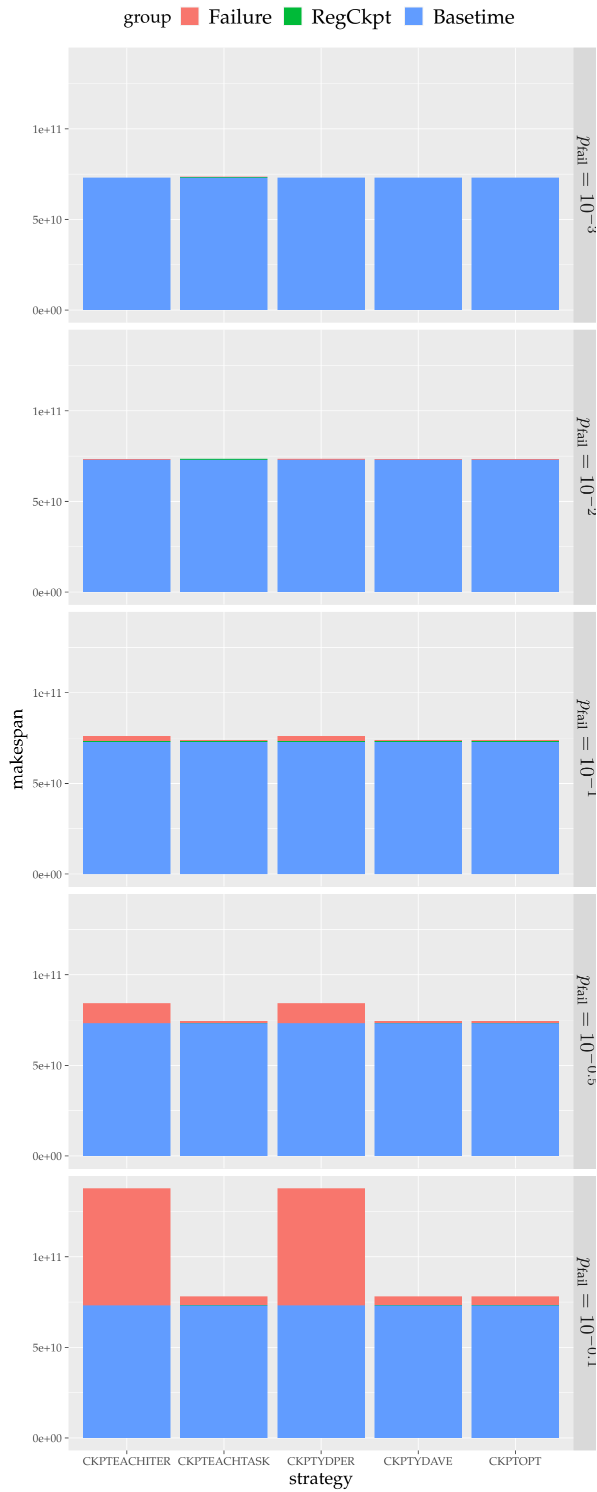


Figure 19: Bar plots for absolute overhead (GCR, $n = 10$ and CCR $= 0.1$).

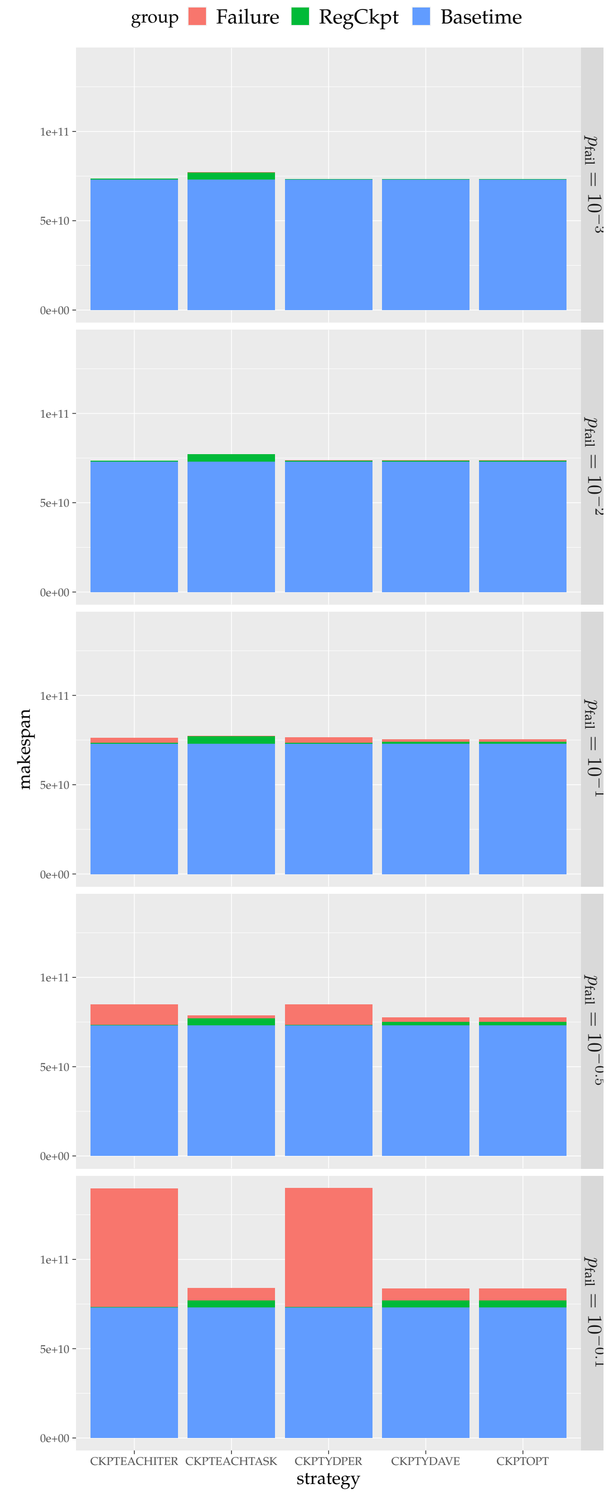


Figure 20: Bar plots for absolute overhead (GCR, $n = 10$ and CCR $= 1$).

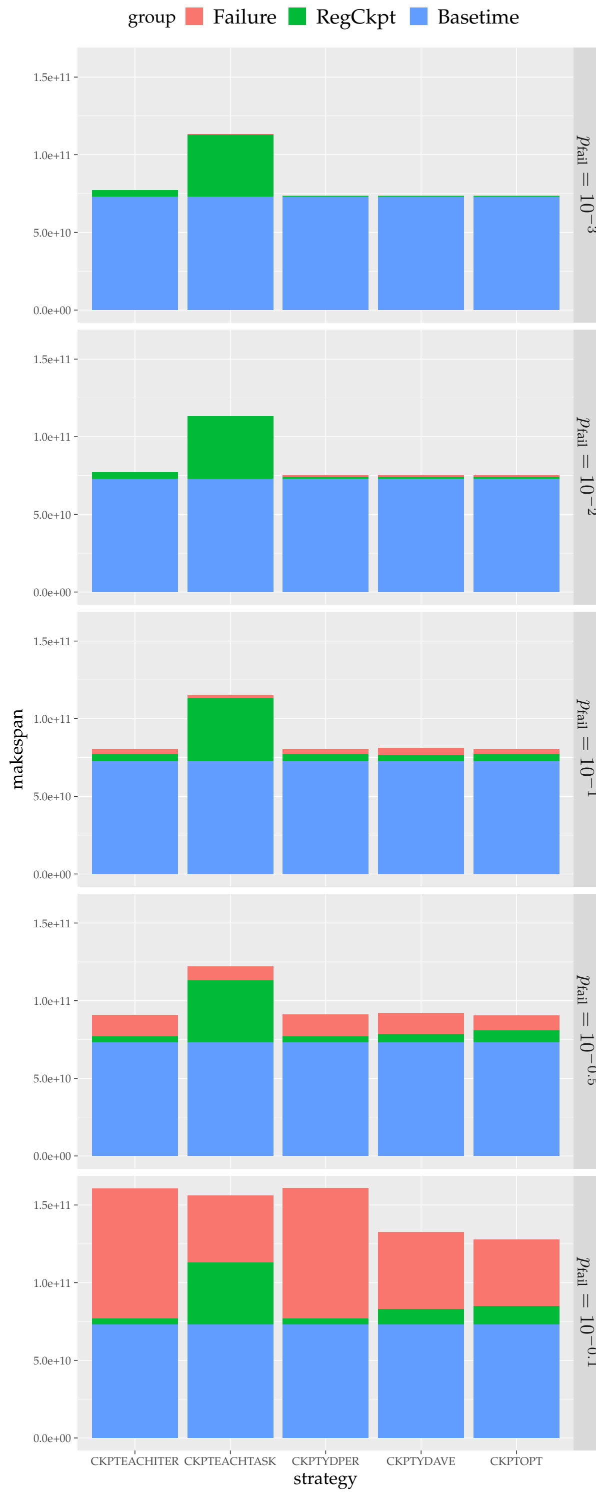


Figure 21: Bar plots for absolute overhead (GCR, $n = 10$ and CCR $= 10$).

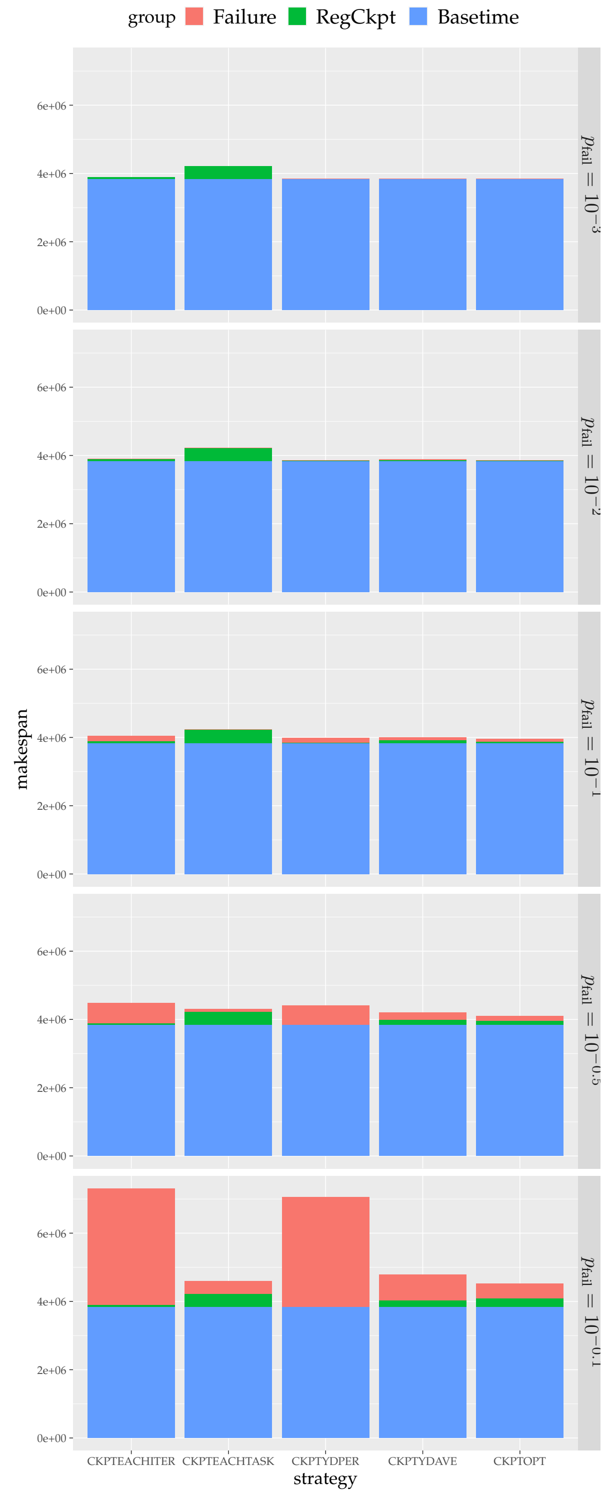


Figure 22: Bar plots for absolute overhead (synthetic, $n = 10$ and $c_i = 0.1 * t_i$).

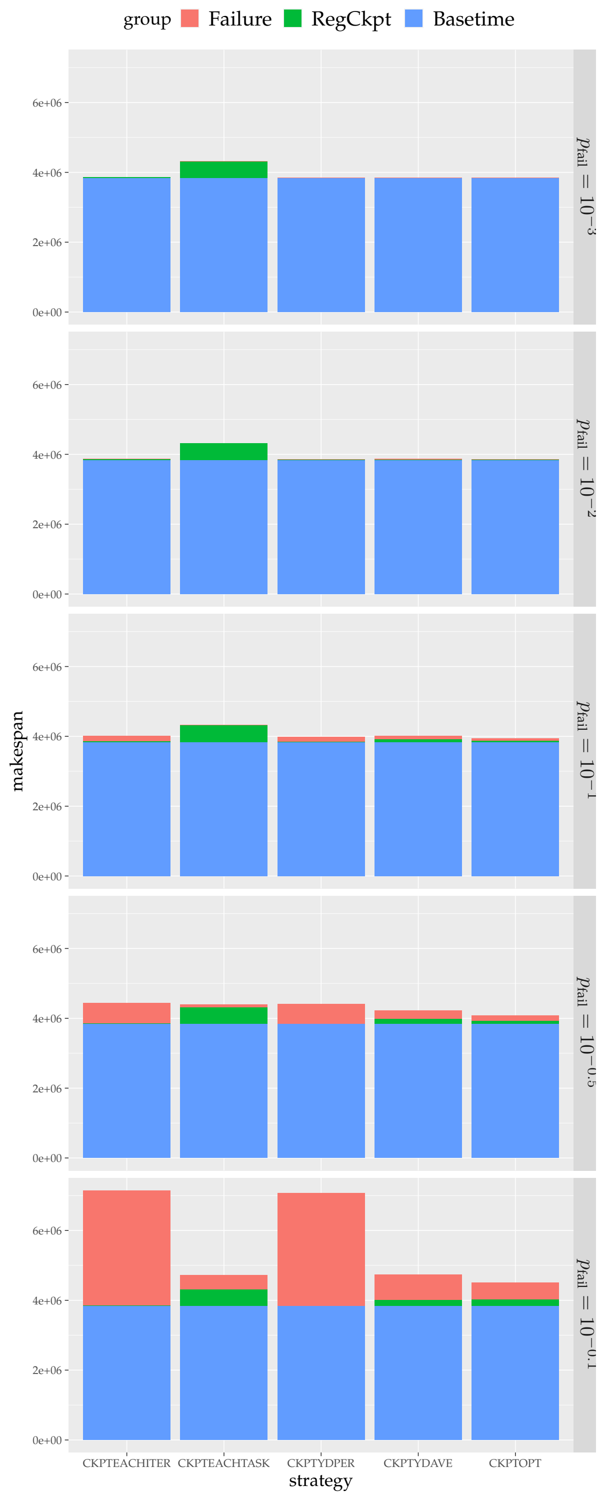


Figure 23: Bar plots for absolute overhead (synthetic, $n = 10$ and c_i drawn in UNIFORM$[10, 100]$.