Two-level checkpointing and verifications for linear task graphs

Anne Benoit*, Aurélien Cavelan*, Yves Robert*†, Hongyang Sun*
*Ecole Normale Supérieure de Lyon & Inria, France
†University of Tennessee Knoxville, USA

***Abstract*—Fail-stop and silent errors are omnipresent on large-scale platforms. Efficient resilience techniques must accommodate both error sources. To cope with the double challenge, a two-level checkpointing and rollback recovery approach can be used, with additional verifications for silent error detection. A fail-stop error leads to the loss of the whole memory content, hence the obligation to checkpoint on a stable storage (e.g., an external disk). On the contrary, it is possible to use in-memory checkpoints for silent errors, which provide a much smaller checkpointing and recovery overhead. Furthermore, recent detectors offer partial verification mechanisms that are less costly than the guaranteed ones but do not detect all silent errors. In this paper, we show how to combine all of these techniques for HPC applications whose dependency graph forms a linear chain. We present a sophisticated dynamic programming algorithm that returns the optimal solution in polynomial time. Simulation results demonstrate that the combined use of multi-level checkpointing and verifications leads to improved performance compared to the standard single-level checkpointing algorithm.**

***Index Terms*—resilience, fail-stop errors, silent errors, multi-level checkpoint, verification, dynamic programming.**

I. Introduction

Resilience is one of the major challenges for extreme-scale computing [12], [13]. In particular, several types of errors should be considered. In addition to the classical fail-stop errors (such as hardware failures), silent errors, also known as silent data corruptions, constitute another threat that cannot be ignored any longer [19], [20], [24], [25]. In order to deal with both types of errors, a traditional checkpointing and rollback recovery strategy can be used [15], coupled with a verification mechanism to detect silent errors [8], [16], [22]. Such a verification mechanism can be either general-purpose (e.g., based on replication [17] or even triplication [18]) or application-specific (e.g., based on algorithm-based fault tolerance (ABFT) [11], on approximate re-execution for ODE and PDE solvers [9], or on orthogonality checks for Krylov-based sparse solvers [16], [22]).

Because verification mechanisms can be costly, alternative techniques capable of rapidly detecting silent errors, with the risk of missing some errors, have been recently developed and studied [2], [3], [10], [14]. We call these verifications *partial verifications*, while perfect verifications (with no error missed) are referred to as *guaranteed verifications*. Furthermore, rather than checkpointing only on a stable storage (e.g., an external disk), a lightweight mechanism of in-memory checkpoints can be provided: one keeps a local copy of the data that has not been corrupted when a silent error strikes, and it can be used to perform a recovery rapidly. However, such local copies are lost if a fail-stop error occurs, and hence copies on stable storage must also be provided.

Designing resilience algorithms by combining all of these techniques is challenging. In this paper, we deal with a simplified, yet realistic, application framework, where a set of application workflows exchange data at the end of their execution. Such a framework can be modeled as a task graph whose dependencies form a linear chain. This scenario corresponds to an HPC application whose workflow is partitioned into a succession of (typically large) tightly-coupled computational kernels, each of which is identified as a task. At the end of each task, we can perform either a partial verification or a guaranteed verification of the task output; or, probably less frequently, we can perform a guaranteed verification followed by a memory checkpoint (we do not take the risk of storing a corrupted checkpoint, hence the need for a guaranteed verification); or again, probably even less frequently, we can perform a guaranteed verification, a memory checkpoint and a disk checkpoint in a row.

The main contribution of this paper is a sophisticated dynamic programming algorithm that returns the optimal solution, i.e., the solution that minimizes the expected execution time, in polynomial time. The originality is that we combine both types of verifications and both types of checkpoints. Furthermore, we present extensive simulations that demonstrate the usefulness of mixing these techniques, and, in particular, we demonstrate the gain obtained thanks to multi-level checkpointing.

To the best of our knowledge, the interplay of verification mechanisms with two types of checkpoints, in-memory and disk-based, has never been investigated for task graphs. Our previous work [6] considers linear chains with a single checkpoint type and guaranteed verification (for the record, the pioneering work [23] for linear chains only deals with a single checkpoint type and no verification). The closest work to this paper is our recent work [7] for divisible load applications, where we consider the same combined framework (with two error sources, two checkpoint types and two verification mechanisms); however, in [7], we target long-lasting executions, which are partitioned into periodic patterns that repeat over time, and we compute the best pattern up to first-order approximations. Here, we do not have the flexibility of divisible load applications, since we can insert resilience mechanisms only at the end of the execution of a task. We may well have a limited number of tasks, which prevents the use of any periodic strategy. Instead, we take a completely different approach and design (quite involved) dynamic programming algorithms that provide the optimal solution for any linear task graph.

The rest of this paper is organized as follows. We detail the model in Section II, before giving the dynamic programming algorithm in Section III. The simulation results are presented in Section IV. Finally, we conclude the paper in Section V with hints for future direction.

II. Model

We consider a linear chain of tasks $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$, where each task T_i has a weight w_i corresponding to the computational load. For notational convenience, we define $W_{i,j} = \sum_{k=i+1}^{j} w_k$ to be the time to execute tasks T_{i+1} to T_j for any $i < j$. Furthermore, we assume that hardware faults (*fail-stop* errors) and silent data corruptions (*silent* errors) coexist, as motivated in Section I. Since these two types of errors are caused by different sources, we assume that they are independent and that both occurrences follow *Poisson process* with arrival rates λ_f and λ_s, respectively. The probability of having at least a fail-stop error during the execution of tasks T_{i+1} to T_j is given by $p^f_{i,j} = 1 - e^{-\lambda_f W_{i,j}}$ and that of having at least a silent error during the same execution is $p^s_{i,j} = 1 - e^{-\lambda_s W_{i,j}}$.

To deal with both fail-stop and silent errors, resilience is provided through the use of a two-level checkpointing scheme coupled with an error detection (or verification) mechanism. When a fail-stop error strikes, the computation is interrupted immediately due to a hardware fault, so all the memory content is destroyed: we then recover from the last disk checkpoint or start again at the beginning of the application. On the contrary, when a silent error is detected, either by a partial verification or by a guaranteed one, we roll back to the nearest memory checkpoint, and recover from the memory copy there, which is much cheaper than recovering from the disk checkpoint.

We enforce that a memory checkpoint is always taken immediately before each disk checkpoint. This can be done with little overhead and it has been enforced in some practical multi-level checkpointing systems [5]. Also, a guaranteed verification is always taken immediately before each memory checkpoint, so that all checkpoints are valid (both memory and disk checkpoints), and hence only one memory checkpoint and one disk checkpoint need to be maintained at any time during the execution of the application. Furthermore, we assume that errors only strike the computations, while verifications, memory copies, and I/O transfers are protected from failures.

Let C_D denote the cost of disk checkpointing, C_M the cost of memory checkpointing, R_D the cost of disk recovery, and R_M the cost of memory recovery. Recall that when a disk recovery is done, we also need to restore the memory state. For simplicity, we assume that the cost R_M is included in the cost R_D. Also, let V^* denote the cost of guaranteed verification and V the cost of a partial verification. The partial verification is also characterized by its *recall*, which is denoted by r and represents the proportion of detected errors over all silent errors that have occurred during the execution. For notational convenience, we define $g = 1 - r$ to be the proportion of undetected errors. Note that the guaranteed verification can be considered as one with recall $r^* = 1$. Since a partial verification usually incurs a much smaller cost and yet has a reasonable recall [3], [10], it is highly attractive for detecting silent errors, and we make use of them between guaranteed verifications.

The objective is to determine where to place disk checkpoints, memory checkpoints, guaranteed verifications and partial verifications, in order to minimize the expected execution time (or makespan) of the application.

III. Dynamic programming

In this section, we present a sophisticated multi-level dynamic programming algorithm to decide which tasks to verify, which tasks to checkpoint, as well as which type of verification or checkpoint to perform. Recall that we assumed a memory checkpoint always comes with a guaranteed verification to ensure the correctness of the results, and that a disk checkpoint always comes with a memory checkpoint, as motivated in Section II.

For convenience, we add a virtual task T_0, which is checkpointed on disk (and hence in memory), and whose recovery cost is zero. This accounts for the fact that it is always possible to restart the application from scratch at no extra cost.

We first describe in Section III-A a dynamic programming algorithm for the case where we use only guaranteed verifications, memory checkpoints and disk checkpoints. We then show how to extend this dynamic programming algorithm to include partial verifications in Section III-B.

A. Without partial verifications

Figures 1, 2, and 3 illustrate the idea of the algorithm without using partial verifications. The algorithm contains three dynamic programming levels, which are responsible for placing disk checkpoints (Figure 1), memory checkpoints (Figure 2), and guaranteed verifications (Figure 3), respectively. An additional step follows to compute the expected execution time between any two verifications. The following describes each step of the algorithm in detail.

Placing disk checkpoints. The first level focuses on placing disk checkpoints (see Figure 1). Let the function $E_{disk}(d_2)$ denote the expected time needed to successfully execute all the tasks from T_1 to T_{d_2}, where task T_{d_2} is verified and checkpointed both on disk and in memory.

In this function, we try all possible locations for the last checkpoint before T_{d_2}. For each possible location d_1, we call the function recursively on d_1 (to place disk checkpoints before T_{d_1}), and we add the expected time needed to execute the tasks from T_{d_1+1} to T_{d_2}. This is done through the $E_{mem}(d_1, d_2)$ function, which also decides where to place memory checkpoints, and accounts for the cost of memory checkpoints. The cost of the disk checkpoint C_D is finally added after T_{d_2}. Note that a location $d_1 = 0$ means that no

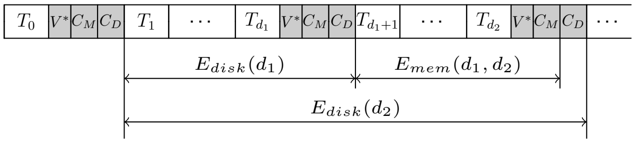

Figure 1. Placing disk checkpoints.

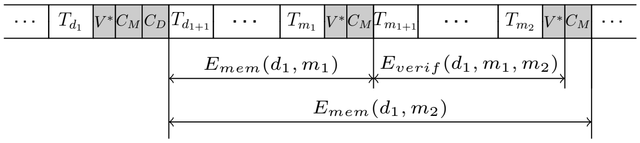

Figure 2. Placing memory checkpoints.

further disk checkpoints are added. In this case, we simply let $E_{disk}(0) = 0$, which initializes the dynamic program. We can express $E_{disk}(d_2)$ as follows:

$$E_{disk}(d_2) = \min_{0 \le d_1 < d_2} \big\{ E_{disk}(d_1) + E_{mem}(d_1, d_2) + C_D \big\}.$$

The total expected time needed to execute all the tasks T_1 to T_n is given by $E_{disk}(n)$.

Placing memory checkpoints. The second level aims at placing additional memory checkpoints between two disk checkpoints (see Figure 2). The function is first called from the first level between two disk checkpoints, each of which also comes with a memory checkpoint. We define $E_{mem}(d_1, m_2)$ as the expected time needed for successfully executing all the tasks from T_{d_1+1} to T_{m_2}, where there is a disk checkpoint at the end of task T_{d_1}, a memory checkpoint at the end of task T_{m_2}, and no other disk checkpoints. Note that there might be a disk checkpoint after T_{m_2}, for instance when we first call this function, but we do not account for the cost of this disk checkpoint in E_{mem}, only for the cost of the memory checkpoint (the cost of the disk checkpoint is already accounted for in E_{disk}).

As before, we try all possible locations for the last memory checkpoint between tasks T_{d_1} and T_{m_2}. For each possible location m_1, we call the function recursively on tasks T_{d_1} to T_{m_1}, and then call the function for the next level, $E_{verif}(d_1, m_1, m_2)$, which computes the expected time needed to execute the tasks from T_{m_1+1} to T_{m_2} (and decides where to place verifications). Finally, we add the cost of the memory checkpoint C_M following T_{m_2}. We can express $E_{mem}(d_1, m_2)$ as follows:

$$E_{mem}(d_1, m_2) = \min_{d_1 \le m_1 < m_2} \big\{ E_{mem}(d_1, m_1) + E_{verif}(d_1, m_1, m_2) + C_M \big\}.$$

If $m_1 = d_1$, there is no extra memory checkpoint between d_1 and m_2, and therefore we initialize the dynamic program with $E_{mem}(d_1, d_1) = 0$.

Placing additional verifications. The third level looks for where to insert additional verifications between two tasks with memory checkpoints (see Figure 3). The function is first called from the second level between two memory checkpoints, each of which also comes with a verification. Therefore, we define $E_{verif}(d_1, m_1, v_2)$ as the expected time needed for successfully executing all the tasks from T_{m_1+1} to T_{v_2}, knowing that the last memory checkpoint is after T_{m_1}, the last disk checkpoint is after T_{d_1}, and there are no checkpoints between T_{m_1+1} and T_{v_2}. Note that $E_{verif}(d_1, m_1, v_2)$ accounts only for the time required to execute and verify these tasks.

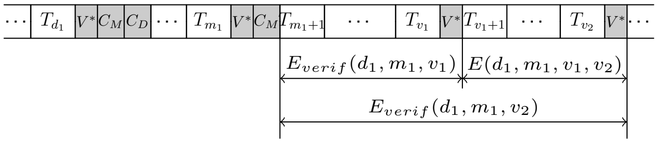

Figure 3. Placing verifications.

As before, we try all possible locations for the last verification between T_{m_1} and T_{v_2}, and for each possible location v_1, we call the function recursively on tasks T_{m_1} to T_{v_1}. Furthermore, we add the expected time needed to successfully execute the tasks T_{v_1+1} to T_{v_2}, denoted by $E(d_1, m_1, v_1, v_2)$, knowing the position of the last disk checkpoint d_1 and the position of the last memory checkpoint m_1. We express $E_{verif}(d_1, m_1, v_2)$ as follows:

$$E_{verif}(d_1, m_1, v_2) = \min_{m_1 \le v_1 < v_2} \big\{ E_{verif}(d_1, m_1, v_1) + E(d_1, m_1, v_1, v_2) \big\}. \quad (1)$$

Again, the case $v_1 = m_1$ means that no further verifications are added, so we initialize the dynamic program with $E_{verif}(d_1, m_1, m_1) = 0$. The verification cost at the end of T_{v_2} is accounted for in the function $E(d_1, m_1, v_1, v_2)$.

Computing the expected execution time between two verifications. Finally, to compute the expected time needed for successfully executing several tasks between two verifications, we need the position of the last disk checkpoint d_1, the position of the last memory checkpoint m_1, and the positions of the two verifications v_1 and v_2.

On the one hand, if a fail-stop error occurs with probability $p^f_{v_1,v_2}$, then the execution stops and we must recover from the last disk checkpoint. In this case, we lose $T^{\text{lost}}_{v_1,v_2}$ time, pay the cost of recovery R_D (set to 0 if $d_1 = 0$), and re-execute the tasks starting from T_{d_1}. The re-execution is done in three steps. First, we call $E_{mem}(d_1, m_1)$ to compute the expected time needed to re-execute the tasks from the last disk checkpoint after T_{d_1} to the last memory checkpoint after T_{m_1}. Then, we call the function $E_{verif}(d_1, m_1, v_1)$ to account for the time needed to re-execute the tasks between the last memory checkpoint after T_{m_1} to the next verification after T_{v_1}. Finally, we re-execute tasks T_{v_1+1} to T_{v_2} with $E(d_1, m_1, v_1, v_2)$.

On the other hand, with probability $1 - p^f_{v_1,v_2}$, there is no fail-stop error. In this case, we pay W_{v_1,v_2} by executing all the tasks from T_{v_1+1} to the next verification after T_{v_2}. Then we add the cost of the guaranteed verification V^*. After the verification, there is a probability $p^s_{v_1,v_2}$ of detecting a silent error, in which case we recover from the last memory checkpoint with a cost R_M (set to 0 if $m_1 = 0$) and only re-execute the tasks from there by calling the function $E_{verif}(d_1, m_1, v_1)$ followed by $E(d_1, m_1, v_1, v_2)$ as before. Therefore, we have:

$$\begin{aligned}
&E(d_1, m_1, v_1, v_2) = \\
&\quad p^f_{v_1,v_2}\Big(T^{\text{lost}}_{v_1,v_2} + R_D + E_{mem}(d_1, m_1) \\
&\qquad + E_{verif}(d_1, m_1, v_1) + E(d_1, m_1, v_1, v_2)\Big) \\
&\quad + \left(1 - p^f_{v_1,v_2}\right)\Big(W_{v_1,v_2} + V^* + p^s_{v_1,v_2}\big(R_M \\
&\qquad + E_{verif}(d_1, m_1, v_1) + E(d_1, m_1, v_1, v_2)\big)\Big). \qquad (2)
\end{aligned}$$

In order to compute the expected execution time, we need to compute $T^{\text{lost}}_{v_1,v_2}$, which is the expected time loss due to a fail-stop error occurring during the execution of tasks T_{v_1+1} to T_{v_2}. We can derive:

$$\begin{aligned}
T^{\text{lost}}_{v_1,v_2} &= \int_0^\infty x\mathbb{P}(X = x | X < W_{v_1,v_2})dx \\
&= \frac{1}{\mathbb{P}(X < W_{v_1,v_2})}\int_0^{W_{v_1,v_2}} x\mathbb{P}(X = x)dx \ ,
\end{aligned}$$

where $\mathbb{P}(X = x)$ denotes the probability that a fail-stop error strikes at time x. By definition, we have $\mathbb{P}(X = x) = \lambda_f e^{-\lambda_f x}$ and $\mathbb{P}(X < W_{v_1,v_2}) = 1 - e^{-\lambda_f W_{v_1,v_2}}$. Integrating by parts, we get:

$$T^{\text{lost}}_{v_1,v_2} = \frac{1}{\lambda_f} - \frac{W_{v_1,v_2}}{e^{\lambda_f W_{v_1,v_2}} - 1} \ . \qquad (3)$$

Now, substituting $T^{\text{lost}}_{v_1,v_2}$ into Equation (2) and simplifying, we obtain:

$$\begin{aligned}
E(d_1, m_1, v_1, v_2) &= e^{\lambda_s W_{v_1,v_2}}\left(\frac{e^{\lambda_f W_{v_1,v_2}} - 1}{\lambda_f} + V^*\right) \\
&+ e^{\lambda_s W_{v_1,v_2}}\left(e^{\lambda_f W_{v_1,v_2}} - 1\right)\left(R_D + E_{mem}(d_1, m_1)\right) \\
&+ \left(e^{(\lambda_s+\lambda_f)W_{v_1,v_2}} - 1\right)E_{verif}(d_1, m_1, v_1) \\
&+ \left(e^{\lambda_s W_{v_1,v_2}} - 1\right)R_M. \qquad (4)
\end{aligned}$$

Complexity. The complexity is dominated by the computation of the table $E_{verif}(d_1, m_1, v_2)$, which contains $O(n^3)$ entries, and each entry depends on at most n other entries that are already computed. All tables are computed in a bottom-up fashion, from the left to the right of the intervals. Hence, the overall complexity of the algorithm is $O(n^4)$.

B. With partial verifications

It may be beneficial to further add partial verifications between two guaranteed verifications. The intuitive idea would be to add yet another level to the dynamic programming algorithm, and to replace $E(d_1, m_1, v_1, v_2)$ in Equation (1) by a call to a function $E^{(intuitive)}_{partial}(d_1, m_1, v_1, p_2, v_2)$, with $p_2 = v_2$, which would compute the expected time needed to execute all the tasks from T_{v_1+1} to T_{p_2} and add further partial verifications (computed from the left to the right).

However, while the dynamic programming approach was rather intuitive without partial verifications, the problem becomes much harder with partial verifications. The main reason is that when computing an interval between two partial verifications, there is a probability g that the error remains undetected after the partial verification. When this happens, we need to account for the time lost executing the following tasks until the error is eventually detected (by the subsequent partial verifications or in the worst case by the guaranteed verification) or until the execution is interrupted by a fail-stop error. This is only possible if we know the optimal positions of the partial verifications after the interval up to the next guaranteed verification. This requires the dynamic programming algorithm to first compute the values at the right of the current interval, hence progressing the opposite way as what was done so far. Therefore, the function becomes $E_{partial}(d_1, m_1, v_1, p_1, v_2)$ (expected time needed to execute all the tasks from T_{p_1+1} to T_{v_2}), and it tries all positions p_2 for the next partial verification. But then, it also requires to remove some terms that account for re-executed work from the intervals on the left of the current interval (because we do not have this information yet), and to re-inject them later in the computation. In the end, we have quite a complicated algorithm.

Expected lost time in case of silent error. First, we compute $E_{right}(d_1, m_1, v_1, p_1, v_2)$, the expected time lost executing the tasks T_{p_1+1} to T_{v_2}, assuming that there was a silent error in this interval. This computation uses p_2, the optimal position of the verification immediately following p_1, which is computed with the dynamic programming. Indeed, T_{p_2} may detect the error or not. If the error is detected by T_{p_2}, we lose $W_{p_1,p_2} + V + R_M$ work, while we use $E_{right}(d_1, m_1, v_1, p_2, v_2)$ if the error remains undetected. Also, we consider fail-stop errors only between T_{p_1+1} and T_{p_2}, because fail-stop errors between T_{p_2+1} and T_{v_2} will be accounted for in $E_{right}(d_1, m_1, v_1, p_2, v_2)$. Note that even if we know that there is a silent error in the interval, we may need to recover from a fail-stop error if it strikes before the silent error is detected. Altogether, we have:

$$\begin{aligned}
&E_{right}(d_1, m_1, v_1, p_1, v_2) = \\
&\quad p^f_{p_1,p_2}\left(T^{\text{lost}}_{p_1,p_2} + R_D + E_{mem}(d_1, m_1)\right) \\
&\qquad + (1 - p^f_{p_1,p_2})\Big(W_{p_1,p_2} + V + (1-g)R_M \\
&\qquad\qquad + gE_{right}(d_1, m_1, v_1, p_2, v_2)\Big) \\
&\quad = (1 - e^{-\lambda_f W_{p_1,p_2}})\Big(\frac{1}{\lambda_f} - \frac{W_{p_1,p_2}}{e^{\lambda_f W_{p_1,p_2}} - 1} \\
&\qquad\qquad + R_D + E_{mem}(d_1, m_1)\Big) \\
&\qquad + e^{-\lambda_f W_{p_1,p_2}}\Big(W_{p_1,p_2} + V + (1-g)R_M \\
&\qquad\qquad + gE_{right}(d_1, m_1, v_1, p_2, v_2)\Big).
\end{aligned}$$

The initialization is $E_{right}(d_1, m_1, v_1, v_2, v_2) = R_M$. Indeed, in this case, there is no task to execute, and if there was a silent error, it is therefore immediately detected by v_2 (a guaranteed

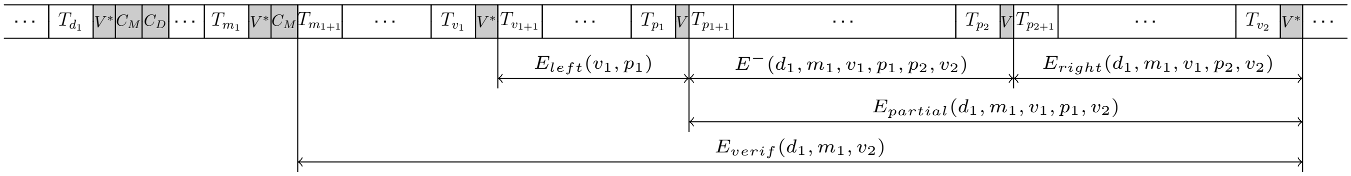

Figure 4. Placing partial verifications.

verification), and we just pay R_M. Knowing p_2, we are therefore able to compute all values of E_{right}. We will see later how we use this knowledge in E_{right}. Note that the time to re-execute the tasks after a recovery is omitted here, since it will be accounted for when computing $E(d_1, m_1, v_1, p_1, p_2, v_2)$, the expected time needed to successfully execute all the tasks between two partial verifications (from T_{p_1+1} to T_{p_2}).

Expected time to compute tasks T_{p_1+1} to T_{p_2}. Figure 4 shows all the tasks involved in the computation of an interval consisting of several tasks between two partial verifications at p_1 and p_2. Let $E(d_1, m_1, v_1, p_1, p_2, v_2)$ denote the expected time needed to successfully execute all the tasks from T_{p_1+1} to T_{p_2}, knowing that the last disk checkpoint is after T_{d_1}, the last memory checkpoint is after T_{m_1}, the last guaranteed verification is after T_{v_1}, and the next guaranteed verification is after T_{v_2}.

On the one hand, if a fail-stop error occurs with probability $p^f_{p_1,p_2}$, then the task stops and we must recover from the last disk checkpoint. We lose $T^{\text{lost}}_{p_1,p_2}$ time, we pay the cost for the disk recovery R_D, and we need to re-execute the tasks starting from T_{d_1}. This is done in three steps: first we call $E_{mem}(d_1, m_1)$ to compute the expected time needed to re-execute the tasks from the last disk checkpoint after T_{d_1} to the last memory checkpoint after T_{m_1}. Then we call the function $E_{verif}(d_1, m_1, v_1)$ to account for the time needed to re-execute the tasks between the last memory checkpoint after T_{m_1} to the next guaranteed verification after T_{v_1}, and finally we are left with the remaining tasks between T_{v_1+1} and T_{p_1}. Let $E_{left}(v_1, p_1)$ denote the expected time needed to re-execute all the tasks from T_{v_1+1} to T_{p_1}. Finally, we can re-execute tasks T_{v_1+1} to T_{v_2} by calling $E(d_1, m_1, v_1, p_1, p_2, v_2)$.

On the other hand, there is a probability $(1 - p^f_{p_1,p_2})$ of having no fail-stop errors. In that case, we execute all the tasks from T_{p_1+1} to the next verification after T_{p_2} and we pay W_{p_1,p_2}. Then we add the cost V for the verification. After the partial verification, there is a probability $p^s_{p_1,p_2}$ of having a silent error. In this case, we pay a recovery from the last memory checkpoint (R_M) and re-executed the tasks from there: we call $E_{verif}(d_1, m_1, v_1)$, followed by $E_{left}(v_1, p_1)$ and $E(d_1, m_1, v_1, p_1, p_2, v_2)$. Furthermore, if the error was not detected (with probability g), we use $E_{right}(d_1, m_1, v_1, p_2, v_2)$ to compute the expected time lost executing the tasks following T_{p_2}, knowing that there is an undetected silent error (as explained earlier). Therefore, we have:

$$\begin{aligned}
&E(d_1, m_1, v_1, p_1, p_2, v_2) = \\
&\quad p^f_{p_1,p_2}\Big(T^{\text{lost}}_{p_1,p_2} + R_D + E_{mem}(d_1, m_1) + E_{verif}(d_1, m_1, v_1) \\
&\qquad\qquad + E_{left}(v_1, p_1) + E(d_1, m_1, v_1, p_1, p_2, v_2)\Big) \\
&+ (1 - p^f_{p_1,p_2})\Big(W_{p_1,p_2} + V + p^s_{p_1,p_2}\big(E_{verif}(d_1, m_1, v_1) \\
&\qquad\qquad + E_{left}(v_1, p_1) + E(d_1, m_1, v_1, p_1, p_2, v_2) \\
&\qquad\qquad + (1 - g)R_M + gE_{right}(d_1, m_1, v_1, p_2, v_2)\big)\Big).
\end{aligned}$$

Substituting $T^{\text{lost}}_{p_1,p_2}$ into the equation above and simplifying, we obtain:

$$\begin{aligned}
&E(d_1, m_1, v_1, p_1, p_2, v_2) = e^{\lambda_s W_{p_1,p_2}}\left(\frac{e^{\lambda_f W_{p_1,p_2}} - 1}{\lambda_f} + V\right) \\
&+ e^{\lambda_s W_{p_1,p_2}}\left(e^{\lambda_f W_{p_1,p_2}} - 1\right)\left(R_D + E_{mem}(d_1, m_1)\right) \\
&+ \left(e^{(\lambda_s+\lambda_f) W_{p_1,p_2}} - 1\right)\left(E_{verif}(d_1, m_1, v_1) + E_{left}(v_1, p_1)\right) \\
&+ \left(e^{\lambda_s W_{p_1,p_2}} - 1\right)\left((1 - g)R_M + gE_{right}(d_1, m_1, v_1, p_2, v_2)\right).
\end{aligned}$$

Finally, because we do not know at this point how to compute $E_{left}(v_1, p_1)$, we remove the term

$$\left(e^{(\lambda_s+\lambda_f) W_{p_1,p_2}} - 1\right) E_{left}(v_1, p_1)$$

from $E(d_1, m_1, v_1, p_1, p_2, v_2)$. This corresponds to the amount of time needed to re-execute all the tasks from T_{v_1+1} to T_{p_1} when there is an error between T_{p_1+1} and T_{p_2}. This time will be added back when computing $E_{partial}$, as explained below. Therefore, we introduce the modified expression of E, denoted E^-, as follows:

$$\begin{aligned}
&E^-(d_1, m_1, v_1, p_1, p_2, v_2) = e^{\lambda_s W_{p_1,p_2}}\left(\frac{e^{\lambda_f W_{p_1,p_2}} - 1}{\lambda_f} + V\right) \\
&+ e^{\lambda_s W_{p_1,p_2}}\left(e^{\lambda_f W_{p_1,p_2}} - 1\right)\left(R_D + E_{mem}(d_1, m_1)\right) \\
&+ \left(e^{(\lambda_s+\lambda_f) W_{p_1,p_2}} - 1\right)\left(E_{verif}(d_1, m_1, v_1)\right) \\
&+ \left(e^{\lambda_s W_{p_1,p_2}} - 1\right)\left((1 - g)R_M + gE_{right}(d_1, m_1, v_1, p_2, v_2)\right).
\end{aligned}$$

Computing $E_{partial}(d_1, m_1, v_1, p_1, v_2)$, the expected time needed to execute all the tasks from T_{p_1+1} to T_{v_2} (and placing extra partial verifications). Finally, we need to compute $E_{partial}$ and to decide when to use additional partial verifications on tasks that are not yet verified. The function is first called from the third level between two guaranteed verifications, and p_1 is originally set to v_1. Therefore, $E_{partial}(d_1, m_1, v_1, p_1, v_2)$ denotes the expected time needed to execute all the tasks from T_{p_1+1} to T_{v_2}, where T_{p_1} is followed by a partial verification (with the exception of the first call) and T_{v_2} is followed by a guaranteed verification, knowing the position of the last disk checkpoint d_1, the position of the last memory checkpoint m_1 and the position of the last guaranteed verification v_1.

Contrarily to the expressions derived in Section III-A, note that partial verifications are placed from the left to the right. We use the expression of E^-, trying all possible positions p_2 for the partial verification following p_1, and we account for the fact that tasks between T_{p_1+1} and T_{p_2} may be re-executed several times (because we removed E_{left} from E^-). In fact, for any number of partial verifications between p_2 and v_2, we can show that $E^-(d_1, m_1, v_1, p_1, p_2, v_2)$ is re-executed $e^{(\lambda_s+\lambda_f)W_{p_2,v_2}}$ times, and hence we obtain:

$$\begin{aligned}&E_{partial}(d_1, m_1, v_1, p_1, v_2) = \\ &\quad \min_{p_1 < p_2 \le v_2} \Big\{ E^-(d_1, m_1, v_1, p_1, p_2, v_2) \cdot e^{(\lambda_s+\lambda_f)W_{p_2,v_2}} \\ &\qquad\qquad + E_{partial}(d_1, m_1, v_1, p_2, v_2) \Big\}.\end{aligned}$$

The initialization case is for $p_2 = v_2$. Then, there is no re-execution of the interval induced by errors to the right of p_2 (within an E_{left}), and therefore we compute only once $E^-(d_1, m_1, v_1, p_1, v_2, v_2)$. Furthermore, the interval is ended by a guaranteed verification, and therefore we add the corresponding verification cost:

$$\begin{aligned}&E_{partial}(d_1, m_1, v_1, v_2, v_2) = \\ &\quad E^-(d_1, m_1, v_1, p_1, v_2, v_2) + e^{(\lambda_s+\lambda_f)W_{p_1,v_2}}(V^* - V).\end{aligned}$$

Because partial verifications are placed from the left to the right, when implementing the algorithm, we first compute all values of $E_{partial}$ on the right of the interval, which are needed to progress towards the left. This is why we always have the values of the next p_2 when computing E_{right}, which correspond to the minimum value selected by $E_{partial}$. However, it was not possible to derive the values of the re-execution for the left part of the interval, hence the trick to compute the number of times each interval is re-executed, due to a failure on the right (the term E_{left} that is removed from the initial expression of E).

Accounting for re-executions *on the left*. Finally, let us show that for any number of partial verifications between p_2 and v_2, the function $E^-(d_1, m_1, v_1, p_1, p_2, v_2)$ re-executes an $e^{(\lambda_s+\lambda_f)W_{p_2,v_2}}$ amount of work. If there are no partial verifications after p_2, then it is executed once when progressing within the computation, and we also need to account for the $e^{(\lambda_s+\lambda_f)W_{p_2,v_2}} - 1$ re-executed work due to the term E_{left} that was suppressed from $E^-(d_1, m_1, v_1, p_2, v_2, v_2)$.

With one intermediate partial verification p_3 between p_2 and v_2, the same reasoning shows that there is an amount of $e^{(\lambda_s+\lambda_f)W_{p_2,p_3}}$ re-executed work coming from the initial execution and the E_{left} term suppressed from $E^-(d_1, m_1, v_1, p_2, p_3, v_2)$. Furthermore, there is an amount of $\left(e^{(\lambda_s+\lambda_f)W_{p_3,v_2}} - 1\right)$ re-executed work coming from the E_{left} term of $E^-(d_1, m_1, v_1, p_3, v_2, v_2)$. In turn, this re-executed work incurs $e^{(\lambda_s+\lambda_f)W_{p_2,p_3}}$ re-executed work (initial execution and re-executions due to the E_{left} term coming from $E^-(d_1, m_1, v_1, p_2, p_3, v_2)$). Overall, the number of re-executed work is finally

$$e^{(\lambda_s+\lambda_f)W_{p_2,p_3}} + \left(e^{(\lambda_s+\lambda_f)W_{p_2,p_3}}\right)\left(e^{(\lambda_s+\lambda_f)W_{p_3,v_2}} - 1\right),$$

and therefore there is a total amount of $e^{(\lambda_s+\lambda_f)W_{p_2,v_2}}$ re-executed work. It is easy to extend this reasoning to any number of intervals by induction, assuming that it is true for i intermediate partial verifications $pv_i, \ldots, pv_1$, and adding a partial verification pv_{i+1} between p_2 and pv_i. The same reasoning holds.

Complexity. Clearly, the complexity is now dominated by the computation of the table $E_{partial}(d_1, m_1, v_1, p_1, v_2)$, which contains $O(n^5)$ entries, and each entry depends on at most n other entries that are already computed. Hence, the overall complexity of the algorithm is $O(n^6)$.

IV. Performance evaluation

In this section, we conduct a set of simulations to assess the relative efficiency of our approach under realistic scenarios. We instantiate the model with actual parameters from the literature and we compare the performance of three algorithms: (i) a single-level algorithm A_{DV^*} with only disk checkpoints (and additional guaranteed verifications), (ii) a two-level algorithm combining memory and disk checkpoints A_{DMV^*} (as in Section III-A), and (iii) the complete algorithm using additional partial verifications A_{DMV} (as in Section III-B). The optimal positions of verifications and disk checkpoints can be easily derived for A_{DV^*}, using a simplification of the proposed dynamic programming algorithm in Section III-A with no additional memory checkpoints.

Simulation setup. We make several assumptions on the input parameters. First, we assume that recovery costs and checkpoint costs are similar; following [19], [21], we set recovery costs to be the same as checkpoint costs, i.e., $R_D = C_D$ and $R_M = C_M$. Then, we assume that a guaranteed verification must check all the data in memory, making its cost in the same order as that of a memory checkpoint, i.e., $V^* = C_M$. Furthermore, we assume partial verifications similar to those proposed in [3], [4], [10], with very low cost while offering good recalls. In the following, we set $V = V^*/100$ and $r = 0.8$. The total computational weight is set to be 25000 seconds and it is distributed uniformly among up to $n = 50$ tasks in three different patterns shown as follows.

(1) *Uniform*: all tasks share the same weight W/n, as in matrix multiplication or in some iterative stencil kernels.

(2) *Decrease*: task T_i has weight $\alpha(n+1-i)^2$, where $\alpha \approx 3W/n^3$; this quadratically decreasing function resembles some dense matrix solvers, e.g., by using LU or QR factorization.

(3) *HighLow*: a set of tasks with large weight is followed by a set of tasks with small weight. In the simulations, we set 10% of the tasks to be large and let them contain 60% of the total computational weight.

We point out that all these choices are somewhat arbitrary and can easily be modified in the evaluations; we believe they represent reasonable values for current and next-generation HPC applications. The code is publicly available at http://graal.ens-lyon.fr/~yrobert/chain2levels.zip for the interested readers to experiment with their own parameters.

Platform settings. Table I presents the four platforms used in the simulations and their main parameters. These platforms have been used to evaluate the Scalable Checkpoint/Restart (SCR) library by Moody et al. [19], who provide accurate measurements for λ_f, λ_s, C_D and C_M using real applications. Note that the Hera platform has the worst error rates, with a platform MTBF of 12.2 days for fail-stop errors and 3.4 days for silent errors. In comparison, and despite its higher number of nodes, the Coastal platform features a platform MTBF of 28.8 days for fail-stop errors and 5.8 days for silent errors. In addition, the last platform uses SSD technology for memory checkpointing, which provides more data space, at the cost of higher checkpointing costs.

Table I
PLATFORM PARAMETERS.

platform	#nodes	λ_f	λ_s	C_D	C_M
Hera	256	9.46e-7	3.38e-6	$300s$	$15.4s$
Atlas	512	5.19e-7	7.78e-6	$439s$	$9.1s$
Coastal	1024	4.02e-7	2.01e-6	$1051s$	$4.5s$
Coastal SSD	1024	4.02e-7	2.01e-6	$2500s$	$180.0s$

Impact of number of tasks. The first column of Figure 5 presents, for each platform, the normalized makespan with respect to the error-free execution time for different numbers of tasks with the *Uniform* weight distribution. Note that varying the number of tasks has an impact on both the size of the tasks and the maximum number of checkpoints and verifications an algorithm can place. When the number of tasks is small (e.g., less than 5), the probability of having an error during the execution (either a fail-stop or a silent) increases quickly (more than 10% on Hera) for a single task. As a result, the application experiences more recoveries and re-executions with larger tasks, which increases the execution overhead. However, when the number of tasks is large enough, the size of the tasks becomes small and the probability of having an error during the execution of one task drops significantly, reducing the recovery and re-execution costs at the same time.

Single-level algorithm $\mathbf{A}_{DV^*}$. The second column of Figure 5 shows the numbers of disk checkpoints (with associated memory checkpoints) and guaranteed verifications used by the A_{DV^*} algorithm on the four platforms and for different numbers of tasks. We observe that a large number of guaranteed verifications is placed by the algorithm while the number of checkpoints remains relatively small (i.e., less than 5 for all the platforms). This is because checkpoints are costly, and verifications help to reduce the amount of time lost due to silent errors. Since verifications are cheaper, the algorithm tends to place as many of them as possible, except when their relative costs also become high (e.g., on Coastal SSD).

Two-level algorithm $\mathbf{A}_{DMV^*}$. The third column of Figure 5 presents the numbers of disk checkpoints, memory checkpoints and guaranteed verifications used by the A_{DMV^*} algorithm on the four platforms and for different number of tasks. We observe that the number of guaranteed verifications remains similar to that placed by the A_{DV^*} algorithm. However, the two-level algorithm uses additional memory checkpoints, which drastically reduces the amount of time lost in re-execution when a silent error is detected. In particular, we observe that the algorithm A_{DMV^*} always leads to a better makespan compared to the single-level algorithm A_{DV^*}, with an improvement of 2% on Hera and 5% on Atlas, as shown in the first column of Figure 5. This demonstrates the usefulness of the multi-level checkpointing approach.

Two-level algorithm with partial verifications $\mathbf{A}_{DMV}$. The last column of Figure 5 presents the numbers of disk checkpoints, memory checkpoints, guaranteed verifications and additional partial verifications used by the A_{DMV} algorithm on the four platforms and for different numbers of tasks. Although partial verifications are always more cost-effective than guaranteed ones, due to the imperfect recall, they are only worth it if one can use a lot of them, which is only possible when the number of tasks is large enough. Therefore, the algorithm only starts to use partial verifications when the number of tasks is greater than 30 on Hera, 40 on Coastal and 50 on Atlas, where the silent error rate is the highest among the four platforms. In our setting, adding partial verifications has a limited impact on the makespan, with the exception of the Coastal SSD platform, where the cost of checkpoints and verifications are much higher than on the other platforms. Partial verifications, being 100 times cheaper than guaranteed verifications, remain the only affordable resilience tool on this platform. In this case, we observe an improved makespan (around 1% with 50 tasks) compared to the A_{DMV^*} algorithm, as shown in the first column of Figure 5.

Distribution of checkpoints and verifications. Figure 6 shows the positions of the disk checkpoints, memory checkpoints, verifications and partial verifications obtained by running the A_{DMV} algorithm on each of the four platforms and for 50 tasks with the uniform distribution. For all platforms, the algorithm does not perform any additional disk checkpoints. These being costly, the algorithm rather uses more memory checkpoints and verifications. On most platforms, the optimal solution is a combination of equi-spaced memory checkpoints and guaranteed verifications, with additional partial verifications in-between. However, on the Coastal SSD platform, the cost of checkpoints and verifications is substantially higher, which leads the algorithm to choose partial verifications over guaranteed ones.

***Decrease* pattern.** In the following, we focus on the platforms Hera and Coastal SSD, which represent both extremes in terms of size (number of processors) and hardware used for memory checkpointing (RAM and SSD, respectively). The first column of Figure 7 presents the performance of

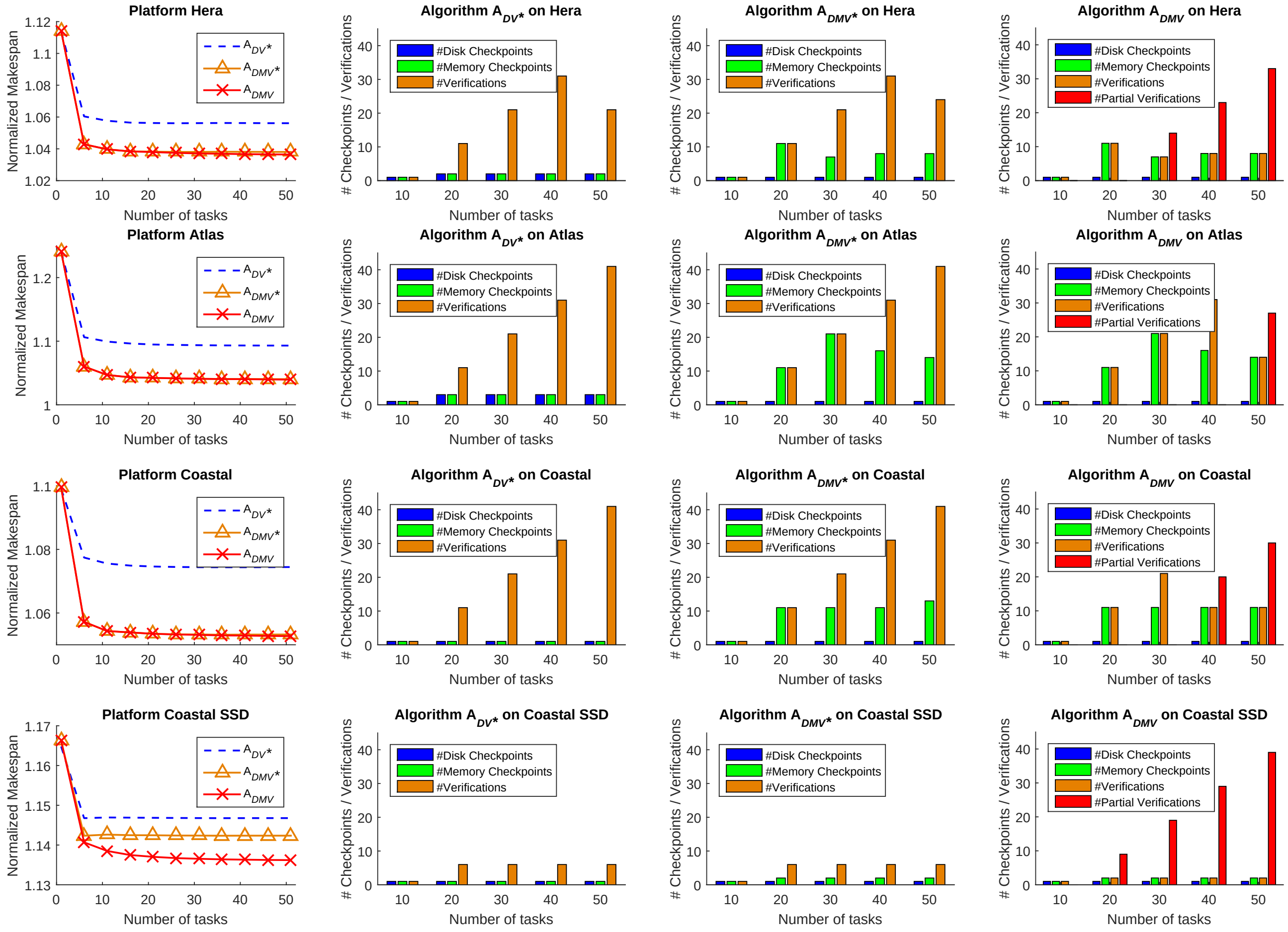

Figure 5. Performance of the three algorithms on each platform with the *Uniform* pattern. Each row corresponds to one platform.

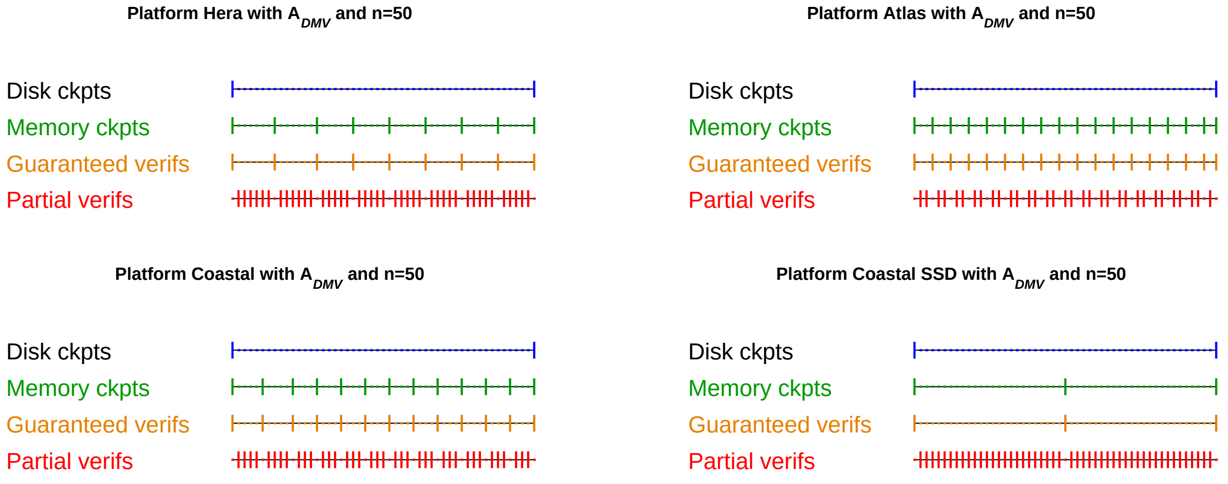
Platform Hera with A_{DMV} and n=50
Platform Atlas with A_{DMV} and n=50
Platform Coastal with A_{DMV} and n=50
Platform Coastal SSD with A_{DMV} and n=50
Disk ckpts
Memory ckpts
Guaranteed verifs
Partial verifs

Figure 6. Distribution of disk checkpoints, memory checkpoints and verifications for the A_{DMV} algorithm on each platform with the *Uniform* pattern.

the three algorithms for different number of tasks and for the *Decrease* pattern. The second column shows the number of disk checkpoints, memory checkpoints, guaranteed and partial verifications given by the A_{DMV} algorithm. The third column is a visual representation of the corresponding solution obtained for 50 tasks and with the same configuration. We observe that the makespan obtained is very similar for all three algorithms (with a slight advantage for A_{DMV}). Since

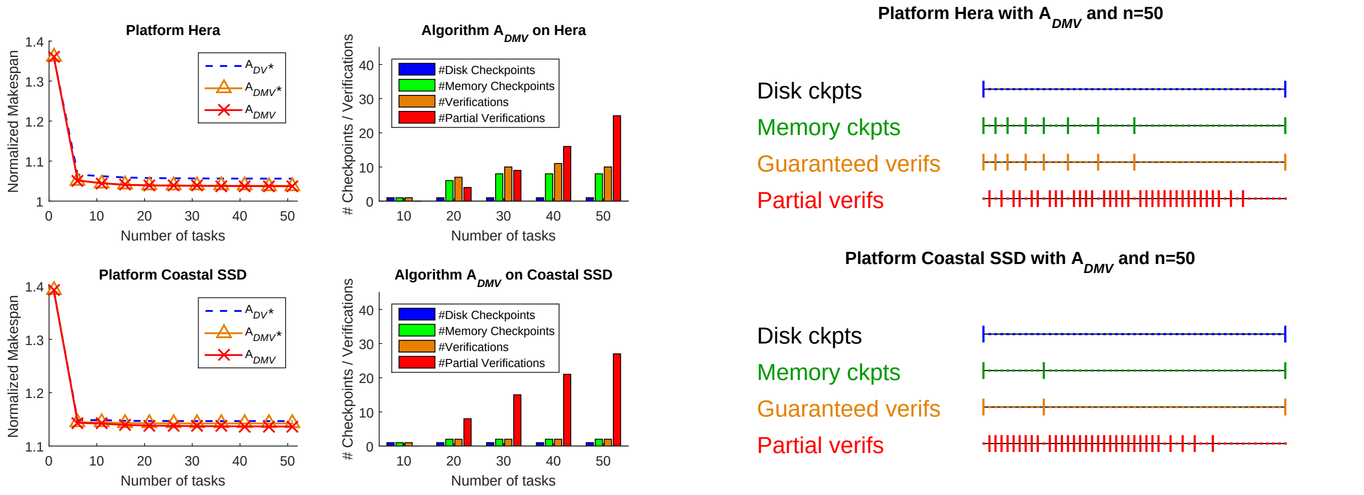

Figure 7. Performance of the three algorithms, and distribution of disk checkpoints, memory checkpoints and verifications (for the A_{DMV} algorithm) on platforms Hera and Coastal SSD with the *Decrease* pattern.

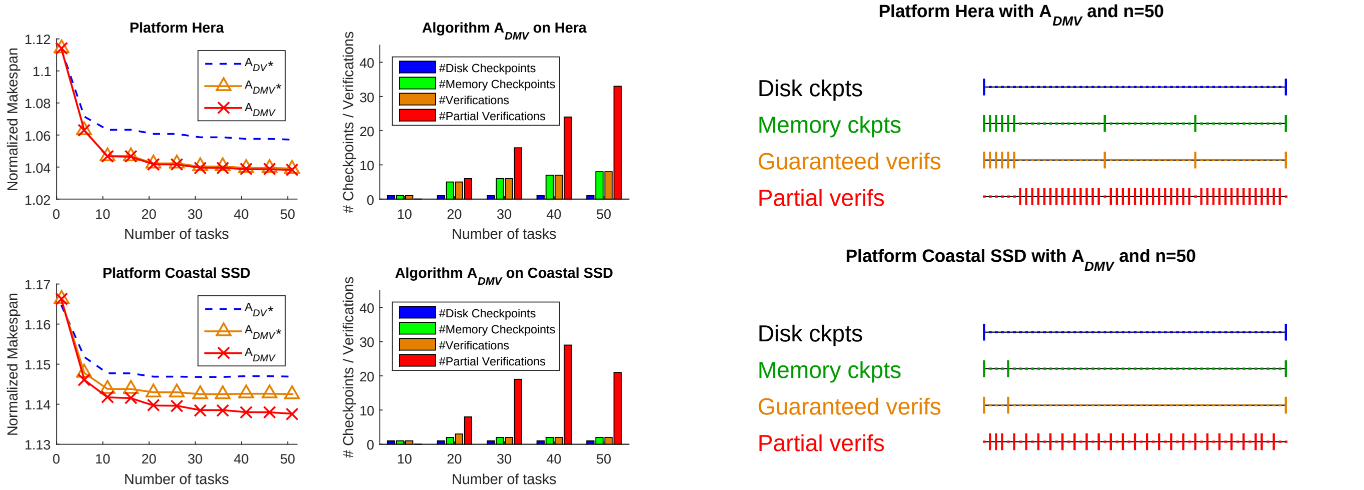

Figure 8. Performance of the three algorithms, and distribution of disk checkpoints, memory checkpoints and verifications (for the A_{DMV} algorithm) on platforms Hera and Coastal SSD with the *HighLow* pattern.

the large tasks at the beginning of the chain are more likely to fail, they will be checkpointed more often, as opposed to the small tasks at the end, which the algorithm does not even consider worth verifying.

***HighLow* pattern.** Once again, we focus on platforms Hera and Coastal SSD. Similarly to Figure 7, Figure 8 assesses the impact of the *HighLow* pattern on the performance of the three algorithms as well as on the numbers and the positions of checkpoints and verifications. Recall that we set the first 10% of the tasks to contain 60% of the total computational weight, while the rest of the tasks contain the remaining 40%. With 50 tasks and a total computational weight of $25000s$, the first 5 tasks have a weight of $3000s$ each, while the remaining tasks have a weight of around $222s$ each. Under this configuration, an error occurring during the execution of a large task would cost $T^{\text{lost}} \approx 1500s$ time loss on average for fail-stop errors (see Equation (3)) and $3000s$ for silent errors, plus an additional $3000s$ time loss for each preceding task that has not been checkpointed. With the MTBF on Hera, a large task will fail with probability 1.3%, as opposed to the probability of 0.096% for small tasks. As a result, the disk checkpoint, which takes $300s$, turns out to be still too expensive, but the memory checkpoint, which takes only $15.4s$ on Hera, becomes mandatory: on average an error will occur way before the total accumulated cost of our preventive memory checkpoints even adds up to the cost of one task. On Coastal SSD, however, the memory checkpoint is still quite expensive, so that only one of the first 5 tasks is marked for verification and memory checkpointing. On both platforms, since the rest of the tasks are small, the solution is similar to the one we observed for the *Uniform* pattern, except that memory checkpoints and verifications are less frequent.

Summary of results. Overall, we observe that the combined use of disk checkpoints and memory checkpoints allows us to decrease the makespan, for the three task patterns and the four platforms. The use of partial verifications further decreases the makespan, especially on the Coastal SSD platform where the checkpointing costs are high. To give some numbers, our approach saves 2% of execution time on Hera and 5% on Atlas. These percentages may seem small, but they correspond to saving half an hour a day on Hera, and more than one hour a day on Atlas, with no further overhead.

V. Conclusion and Future Work

In this paper, we proposed a two-level checkpointing scheme to cope with both fail-stop errors and silent data corruptions on large-scale platforms. Although numerous studies have dealt with either error source, few studies have dealt with both, while it is mandatory to address both sources simultaneously at scale. By combining standard disk checkpointing technique with in-memory checkpoints and verification mechanisms (partial or guaranteed), we have designed a multi-level dynamic programming algorithm that computes the optimal solution for a linear application workflow in polynomial time. Simulations based on realistic parameters on several platforms show consistent results, and confirm the benefit of the combined approach. While the most general algorithm has a high complexity of $O(n^6)$, where n is the number of tasks, it executes within a few seconds for $n = 50$ tasks, and therefore can be readily used for real-life linear workflows whose sizes rarely exceed tens of tasks.

One interesting future direction is to assess the usefulness of this approach on general application workflows. The problem gets much more challenging, even in the simplified scenario where each task requires the entire platform to execute. In fact, in this simplified scenario, it is already NP-hard to decide which task to checkpoint in a simple join graph ($n-1$ source tasks and a common sink task), with only fail-stop errors striking (hence a single level of checkpoint and no verification at all) [1]. Still, heuristics are urgently needed to address the same problem as in this paper, with two error sources, two checkpoint types and two verification mechanisms, if we are to deploy general HPC workflows efficiently at scale.

Acknowledgments

This research was funded in part by the European project SCoRPiO, by the LABEX MILYON (ANR-10-LABX-0070) of Université de Lyon, within the program "Investissements d'Avenir" (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR), and by the PIA ELCI project. Yves Robert is with Institut Universitaire de France. A short version of this paper was presented at PMBS'15 (without proceedings).

References

[1] G. Aupy, A. Benoit, H. Casanova, and Y. Robert. Scheduling computational workflows on failure-prone platforms. In *17th Workshop on Advances in Parallel and Distributed Computational Models APDCM 2015*, 2015.

[2] L. Bautista-Gomez, A. Benoit, A. Cavelan, S. K. Raina, Y. Robert, and H. Sun. Which verification for soft error detection? In *Proc. HiPC*, 2015. Full version available as INRIA RR-8741.

[3] L. Bautista Gomez and F. Cappello. Detecting silent data corruption through data dynamic monitoring for scientific applications. *SIGPLAN Notices*, 49(8):381–382, 2014.

[4] L. Bautista Gomez and F. Cappello. Detecting and correcting data corruption in stencil applications through multivariate interpolation. In *Proc.1st Int. Workshop on Fault Tolerant Systems (FTS)*, 2015.

[5] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. FTI: High performance fault tolerance interface for hybrid systems. In *Proc. SC'11*, 2011.

[6] A. Benoit, A. Cavelan, Y. Robert, and H. Sun. Assessing general-purpose algorithms to cope with fail-stop and silent errors. In *Proc. PMBS'14*, 2014. Full version available as INRIA-8599.

[7] A. Benoit, A. Cavelan, Y. Robert, and H. Sun. Optimal resilience patterns to cope with fail-stop and silent errors. In *Proc. IPDPS'16*, 2016. Full version available as INRIA RR-8786.

[8] A. Benoit, Y. Robert, and S. K. Raina. Efficient checkpoint/verification patterns. *Int. J. High Performance Computing Applications*, 2015.

[9] A. R. Benson, S. Schmit, and R. Schreiber. Silent error detection in numerical time-stepping schemes. *Int. J. High Performance Computing Applications*, 2014.

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

[11] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou. Algorithm-based fault tolerance applied to high performance computing. *J. Parallel Distrib. Comput.*, 69(4):410–416, 2009.

[12] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward Exascale Resilience. *Int. Journal of High Performance Computing Applications*, 23(4):374–388, 2009.

[13] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1), 2014.

[14] A. Cavelan, S. K. Raina, Y. Robert, and H. Sun. Assessing the impact of partial verifications against silent data corruptions. In *Proc. ICPP*, 2015.

[15] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[16] Z. Chen. Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *Proc. PPoPP*, pages 167–176, 2013.

[17] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proc. SC'12*, page 78, 2012.

[18] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM J. Res. Dev.*, 6(2):200–209, 1962.

[19] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *Proc. SC'10*, 2010.

[20] T. O'Gorman. The effect of cosmic rays on the soft error rate of a DRAM at ground level. *IEEE Trans. Electron Devices*, 41(4):553–557, 1994.

[21] F. Quaglia. A cost model for selecting checkpoint positions in time warp parallel simulation. *IEEE Trans. Parallel Dist. Syst.*, 12(4):346–362, 2001.

[22] P. Sao and R. Vuduc. Self-stabilizing iterative solvers. In *Proc. ScalA '13*, 2013.

[23] S. Toueg and O. Babaoğlu. On the optimum checkpoint selection problem. *SIAM J. Comput.*, 13(3), 1984.

[24] J. Ziegler, M. Nelson, J. Shell, R. Peterson, C. Gelderloos, H. Muhlfeld, and C. Montrose. Cosmic ray soft error rates of 16-Mb DRAM memory chips. *IEEE Journal of Solid-State Circuits*, 33(2):246–252, 1998.

[25] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, and B. Chin. IBM experiments in soft fails in computer electronics. *IBM J. Res. Dev.*, 40(1):3–18, 1996.