A Generic Approach to Scheduling and Checkpointing Workflows*

Journal Title
XX(X):1–18

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Li Han[1,2], Valentin Le Fèvre[2], Louis-Claude Canon[3], Yves Robert[2,4], and Frédéric Vivien[2]

Abstract

This work deals with scheduling and checkpointing strategies to execute scientific workflows on failure-prone large-scale platforms. To the best of our knowledge, this work is the first to target fail-stop errors for arbitrary workflows. Most previous work addresses soft errors, which corrupt the task being executed by a processor but do not cause the entire memory of that processor to be lost, contrarily to fail-stop errors. We revisit classical mapping heuristics such as HEFT and MINMIN and complement them with several checkpointing strategies. The objective is to derive an efficient trade-off between checkpointing every task (CKPTALL), which is an overkill when failures are rare events, and checkpointing no task (CKPTNONE), which induces dramatic re-execution overhead even when only a few failures strike during execution. Contrarily to previous work, our approach applies to arbitrary workflows, not just special classes of dependence graphs such as M-SPGS (Minimal Series-Parallel Graphs). Extensive experiments report significant gain over both CKPTALL and CKPTNONE, for a wide variety of workflows.

Keywords

workflow, checkpoint, fail-stop error, resilience

1 Introduction

This work deals with scheduling techniques to deploy scientific workflows on large parallel or distributed platforms. Scientific workflows are the archetype of HPC (High Performance Computing) applications, which are naturally partitioned into tasks that represent computational kernels. The tasks are partially ordered because the output of some tasks may be needed as input to some other tasks. Altogether, the application is structured as a DAG (Directed Acyclic Graph) whose nodes are the tasks and whose edges enforce the dependences. Nodes are weighted by the computational requirements (in flops) while edges are weighted by the size of communicated data (in bytes). Given a workflow and a platform, the problem of mapping the tasks onto the processors and to schedule them so as to minimize the total execution time, or makespan, has received considerable attention.

This classical mapping and scheduling problem has recently been revisited to account for the fact that errors and failures can strike during execution. Indeed, platform sizes have become so large that errors and failures are likely to strike at a high rate during application execution (Cappello et al. (2014)). More precisely, the MTBF (Mean Time Between Failures) μ_P of the platform decreases linearly with the number of processors P, since $\mu_P = \frac{\mu_{\text{ind}}}{P}$, where μ_{ind} is the MTBF of each individual component (see Proposition 1.2 in (Hérault and Robert (2015))). Take $\mu_{\text{ind}} = 10$ years as an example. If $P = 10^5$ then $\mu_P \approx 50$ minutes and if $P = 10^6$ then $\mu_P \approx 5$ minutes: from the point of view of fault-tolerance, scale is the enemy.

Several approaches (see Section 6 for a review) have been proposed to mitigate the simplest instance of the problem, that of soft and silent errors. Soft errors cause a task execution to fail but without completely losing the data present in the processor memory. Local checkpointing (or more precisely making a copy of all task input/output data), and task replication, are the most widely used technique to address soft errors. Silent errors represent a different challenge than soft errors, in that they do not interrupt the execution of the task but corrupt its output data. However, their net effect is the same, since a task must be re-executed whenever a silent error is detected. A silent error detector is applied at the end of a task's execution, and the task must be re-executed from scratch in case of an error. Again, local checkpointing (making copies of input/output data) or replicating tasks and comparing outputs, are two common techniques to mitigate the impact of silent errors.

Fail-stop errors, or failures, are much more difficult to deal with. In the case of a fail-stop error (e.g., a crash due to a power loss or some other hardware problem) the execution of the processor stops, all the content of its memory is lost, and the computations have to be restarted from scratch, either on the same processor once it reboots or on a spare. The de-facto approach to handle such failures is Checkpoint/Restart (C/R), by which application state is saved to stable storage, such as a shared file system, throughout execution. The common strategy used in practice is *checkpoint everything*, or CKPTALL: all output data of each task is saved onto stable

[1] East China Normal University, China
[2] Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, F-69342, LYON Cedex 07, France
[3] FEMTO-ST, Université de Bourgogne Franche-Comté, France
[4] University of Tennessee Knoxville, USA

Corresponding author:
Li Han, LIP, 46 Allée d'Italie 69364 Lyon, France
Email: li.han@ens-lyon.fr

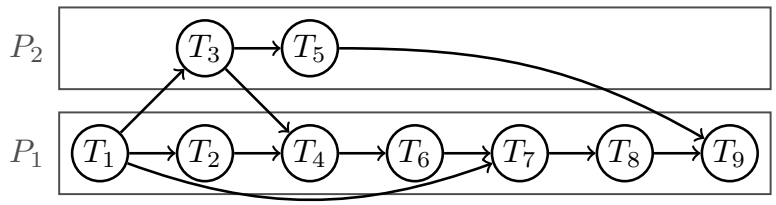

Figure 1. Schedule of a workflow with 9 tasks on 2 processors (each edge corresponds to a file dependence between tasks).

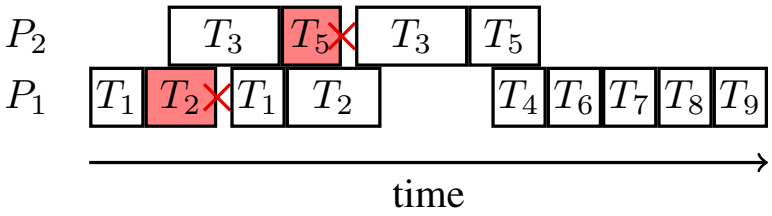

Figure 2. Sample execution of the workflow in Figure 1 without any checkpoint, with two failures striking during the execution of T_2 on P_1 and during that of T_5 on P_2.

storage (in which case we say "the task is checkpointed"). For instance, in production Workflow Management Systems (WMSs) (Deelman et al. (2015); Fahringer et al. (2007); Wilde et al. (2011); Wolstencroft et al. (2013); Altintas et al. (2004); Albrecht et al. (2012)), the default behavior is that all output data is saved to files and all input data is read from files, which is exactly the CKPTALL strategy. While this strategy leads to fast restarts in case of failures, its downside is that it maximizes checkpointing overhead. At the other end of the spectrum would be a *checkpoint nothing* strategy, or CKPTNONE, by which all output data is kept in memory (up to memory capacity constraints) and no task is checkpointed. This corresponds to "in-situ" workflow executions, which has been proposed to reduce I/O overhead (Zhang et al. (2012)). The downside is that, in case of a failure, a large number of tasks may have to be re-executed, leading to slow restarts. The objective of this work is to achieve a desirable trade-off between these two extremes. To the best of our knowledge, no general solution is available. We build upon our previous work (Han et al. (2018a)) that was restricted to M-SPGS (Minimal Series-Parallel Graphs) (Valdes et al. (1979)). In (Han et al. (2018a)), we took advantage of the recursive structure of M-SPGS and used proportional mapping (Pothen and Sun (1993)) for scheduling and checkpointing M-SPG workflows as sets of superchains. For general graphs, we have to resort to classical scheduling heuristics such as HEFT (Topcuoglu et al. (2002)) and MINMIN (Braun et al. (2001)), two reference scheduling algorithms widely used by the community. We provide extensions of HEFT and MINMIN that allow for a smaller subset of tasks to be checkpointed and lead to better makespans than the versions where each task (CKPTALL) or no task (CKPTNONE) is checkpointed.

The main contributions of this paper are the following:

- We deal with arbitrary dependence graphs, and require no graph transformation before applying our scheduling and checkpointing algorithms.
- We compare several mapping strategies and combine them with several checkpointing strategies.
- We design an event-based simulator to evaluate the makespan of the proposed solution. Indeed, computing the expected makespan of a solution is a difficult problem (Han et al. (2018a)), and simple Monte-Carlo based simulations cannot be applied to general DAGs unless all tasks are checkpointed: otherwise, sampling the weight distribution for each task independently is not enough to compute the makespan, since a failure may involve re-executing several tasks (as shown in Section 2).
- We report extensive experimental evaluation with both real-world and randomly generated workflows to quantify the performance gain achieved by the proposed approach.

The rest of the paper is organized as follows. First in Section 2, we work out an example to help understand the difficulty of the problem. Then we introduce the performance model in Section 3. We detail our scheduling and checkpointing algorithms in Section 4. We give experimental results in Section 5. Section 6 surveys the related work. Finally, we provide concluding remarks and directions for future work in Section 7.

2 Example

In this section, we illustrate the difficulty of deciding where to place checkpoints in a workflow. Consider the example of Figure 1 with 9 tasks, T_i, $1 \leq i \leq 9$, that have been mapped on 2 processors as shown on the figure. Note that this DAG cannot be reduced to an M-SPG and our previous approach (Han et al. (2018a)) cannot be applied for this graph. While most tasks are assigned to processor P_1, some tasks are assigned to the second processor, P_2, to exploit the parallelism of the DAG. Any dependence between two tasks represents a file that is required to start the execution of the successor task; hence, $T_1 \rightarrow T_2$ represents a file produced by task T_1 that is required for the execution of task T_2 to start. Because T_1 and T_2 are both executed on processor P_1, this file is kept in the memory of P_1 after T_1 completes. However, for the dependence $T_1 \rightarrow T_3$, because the tasks T_1 and T_3 are executed on different processors, the corresponding file must

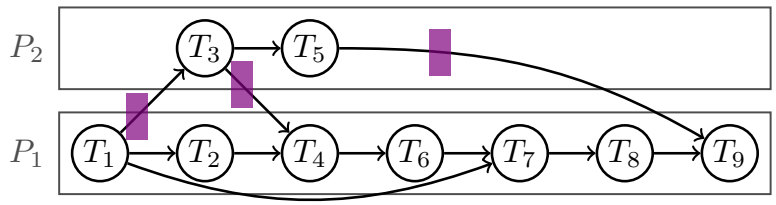

Figure 3. A purple *crossover* checkpoint is performed for each file produced by one processor and used by another one.

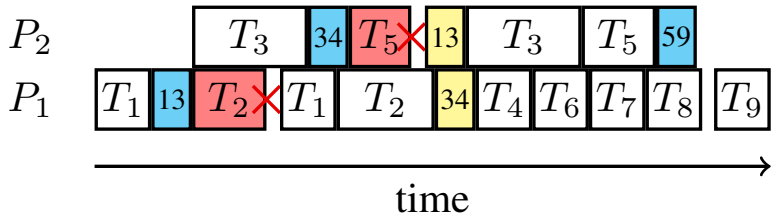

Figure 4. Sample execution of the application in Figure 3 with two failures striking during the execution of T_2 on P_1 and that of T_5 on P_2, with crossover checkpoints. Label ij indicates the file from T_i to T_j. Now T_4 can start before the re-execution of T_3 since its output was checkpointed.

be retrieved by P_2. Such a dependence between two tasks assigned to two different processors is called a *crossover dependence*.

In a first scenario, let us suppose that no task is checkpointed as showed in Figure 1: then if no failure strikes, the makespan will be the shortest possible, consisting only of the execution time of each task and of retrieving the necessary input files. However, as soon as a failure happens, we may need to restart the whole application from the very beginning. To study such a scenario, we need to explicit the memory management. Let us assume that once a processor has sent a file to another processor, then this file is deleted from the memory of the producing processor. For instance, as soon as P_2 has received from P_1 the file corresponding to the dependence $T_1 \rightarrow T_3$, this file is erased from the memory of P_1. Remember that a failure wipes out the whole content of the memory of the struck processor. Thus, if a failure strikes during the execution of T_5, to be able to re-attempt to execute T_5, T_3 will need to be re-executed before (because the file $T_3 \rightarrow T_5$ is no longer available), which requires T_1 to be re-executed first (because the file $T_1 \rightarrow T_3$ is no longer available). Hence, a single failure in a part of the graph may require the re-execution of most of the workflow, which is known as the domino effect in fault tolerance protocols and needs to be mitigated. Figure 2 shows an example of execution of the DAG when no task is checkpointed. To execute T_4, we need both T_2 and T_3 to finish successfully, and that no fault strikes neither P_1 nor P_2 between the completion of these tasks and the start of T_4. Here, T_2 does not finish so T_1 is re-executed. When P_2 fails, we need to re-execute T_3, which requires input from T_1. Luckily (!), P_1 already suffered from a failure, so T_1 has already been re-executed. Otherwise, we would have had to restart the execution of the whole workflow because of the failure of P_2.

To avoid rolling back to the beginning in case of failures, we can try to place some checkpoints inside the workflow. As commonly assumed in workflow management systems (Deelman et al. (2015); Fahringer et al. (2007); Wilde et al. (2011); Wolstencroft et al. (2013); Altintas et al. (2004); Albrecht et al. (2012)), any file produced by one processor and required by another processor is necessarily saved to stable storage. Thus, an error on one processor will not lead to re-execution on another processor. In the second scenario shown in Figure 3, we decide to checkpoint every crossover dependence (from T_1 to T_3, T_3 to T_4, and T_5 to T_9). An execution of that schedule is shown in Figure 4. Cyan boxes represent checkpoints while yellow boxes represent data being read. We can see that thanks to the crossover checkpoints, T_4 does not need to wait for the completion of the second execution of T_3 anymore, as T_3 output data has already been checkpointed. Moreover, if only a failure on P_2 happened, instead of rolling back to task T_1 to re-execute T_3 as it was the case before, T_3 could have restarted directly (although the entire content of the processor memory is lost, so all inputs of T_3 must be recovered from stable storage after a downtime before the execution of T_3 can restart). Note that we start executing the target task of a crossover dependence after checkpointing, to make sure that all files needed are available in the stable storage in case of failure. The motivation to checkpoint all files involved in crossover dependences is to isolate the processors. Indeed, if all crossover files are checkpointed, a failure on a processor will never lead to the re-execution of a task successfully executed on another processor. Overall, we will lose less time recomputing tasks or waiting for their second completion. However, reading from stable storage and checkpointing also take time. Finding the right trade-off is the main focus of this paper: deciding which tasks should be checkpointed, so that the overhead added by the checkpointing and reading of files is not more expensive than the re-execution of tasks.

We conclude by informally introducing examples of checkpointing strategies that achieve desirable trade-offs (see Section 4.2 for details). Two additional checkpoints, in blue, called *induced* checkpoints, have been added in Figure 5. Their role is to secure the fast re-execution of tasks that are the target of a crossover dependence, namely T_4 and T_9. The blue checkpoint after T_2 isolates the execution of the task sequence $S_1 = \{T_4, T_6, T_7, T_8\}$ on P_1. To this purpose, it is necessary to checkpoint all intermediate results that may

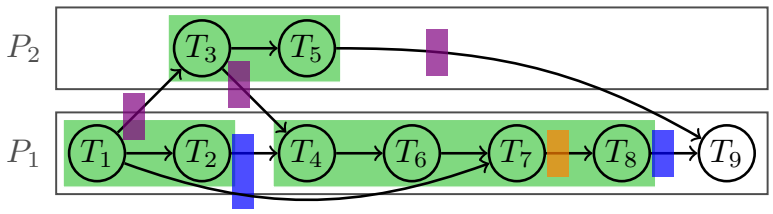

Figure 5. Blue *induced* checkpoints are used to isolate task sequences on a processor (labeled in green, such as the sequence T_4, T_6, T_7 and T_8 on P_1). Finally, additional checkpoints can be added inside an idle-free task sequence through a dynamic programming algorithm: the orange checkpoint corresponds to such an addition.

be used after the execution of T_2: these are the files generated by previous tasks, namely $T_1 \rightarrow T_7$ and $T_2 \rightarrow T_4$, This way, when a failure strikes, previous tasks do not have to be restarted and the computation may be restarted directly from T_4. This way, tasks in the sequence S_1 may be sequentially executed without idle time. It would not have been possible to include T_1 and T_2 in S_1 because T_4 could have waited for the completion of T_3 leading to idle time in some scenarios. Similarly, the second blue checkpoint isolates the execution of T_9.

Finally, once the four tasks T_4, T_6, T_7, and T_8 of the sequence S_1 have been "isolated" from other tasks, it is possible to use a dynamic programming algorithm similar to that used in (Han et al. (2018a)) in order to introduce additional checkpoints. In the example of Figure 5, a single additional checkpoint, in orange, is inserted after T_7.

3 Model

This section details the execution and fault-tolerance models used to compare scheduling and checkpointing algorithms.

3.1 Execution Model

The execution model for a task workflow on a homogeneous system is represented as a Directed Acyclic Graph (DAG), $G = (V, E)$, where V is the set of nodes corresponding to the tasks, and E is the set of edges corresponding to the dependences between tasks. In a DAG, a node without any predecessor is called an *entry* node, while a node without any successor is an *exit* node. For a task T in G, $pred(T)$ and $succ(T)$ represent the set of its immediate predecessors and successors respectively. We say that a task T is *ready* if either it does not have any predecessor in the dependence graph, or if all its predecessors have been executed.

In this model, the execution time of a task $T_i \in V$ is w_i, i.e., its execution time in a failure-free execution. Each dependence $(T_i, T_j) \in E$ is associated with the cost $c_{i,j}$ to store/read the data onto/from stable storage. In some real-world instances from (Pegasus (2014)), a task may generate multiple files for different successors or generate the same file for more than one successor task. Direct communications between processors take place through a communication network which is much faster than writing/reading from the stable storage. Before the execution of T_j on processor P_k, all input files needed by T_j must be present in the local memory of P_k. Suppose that T_j needs a file from T_i executed on processor P_l, it will first try to retrieve the file from the local memory of P_l. The start time of T_j is $\max\{t_{c_{ij}}, (t_{avail_k} + t_c)\}$, where $t_{c_{ij}}$ is the finish time of checkpointing the file transferred from T_i to T_j, t_{avail_k} donates the available time on processor P_k, and t_c is the communication time. We start executing the task after checkpointing the crossover dependence, to make sure that all files needed are available in the stable storage in case of failure. If the file is absent from local memory of P_l, then it must be *read* from the stable storage.

3.2 Fault-Tolerance Model

In this work, each processor is a processing element that is subject to its own individual failures. Failures can strike a processor at any time, during either task execution or waiting time. Failure inter-arrival times are assumed to be Exponentially distributed. These failure-prone processors stop their execution once a failure strikes, i.e., we have fail-stop errors. When a fail-stop error strikes a processor, the whole content of its memory is lost and the computation it was performing must be restarted, either on the same processor after a reboot, or on a spare processor (e.g., taken from a pool of spare processors either specifically requested by the job submitter, or maintained by the resource management infrastructure).

Consider a single task T, with weight w, scheduled on such a processor, and whose input is stored on stable storage. The total execution time W of T is a random variable, because several execution attempts may be needed before the task succeeds. We assume that failures are i.i.d. (independent and identically distributed) across the processors and that the failure inter-arrival times at each processor is Exponentially distributed with Mean Time Between Failures (MTBF) $\mu = 1/\lambda$. Let $\lambda \ll 1$ be the Exponential failure rate of the processor. With probability $e^{-\lambda w}$, no failure occurs, and W is equal to w. With probability $(1 - e^{-\lambda w})$, a failure occurs. For Exponentially distributed failures, the expected time to failure, knowing that a failure occurs during the task execution (i.e., in the next w seconds), is $1/\lambda - \frac{w}{(e^{\lambda(r+w)}-1)}$ (Hérault and Robert (2015)). After this failure, there is a downtime d, which is (an upper bound of) the time needed to reboot the processor or migrate to a spare. Then we start the execution again, first with the recovery r, the time to read data from stable storage, and then the work w. With a general model where an unbounded number of failures can occur during recovery and work, the expected time W to execute task T is given by $W = \left(\frac{1}{\lambda} + d\right) e^{\lambda r} \left(e^{\lambda w} - 1\right)$ (Hérault and Robert (2015)). Now if the output data of task T is checkpointed, with a time c to write all of its output files onto stable storage, the total time

becomes:

$$W = \left(\frac{1}{\lambda} + d\right) e^{\lambda r} \left(e^{\lambda(w+c)} - 1\right) . \quad (1)$$

Equation (1) assumes that failures can also occur during checkpoints, which is the most general model for failures. We also assume that failures may strike during the idle time (i.e., waiting time) of the processor (e.g., the power supply may fail). In the case of a sequence of non-checkpointed tasks to be executed on a processor P, the output data of each task resides in the memory of P for use by subsequent tasks. When a failure strikes P, the entire memory content is lost and the whole task sequence must be re-executed from scratch.

3.3 Problem Formulation

Given a DAG and a set of processors on which fail-stop failures strike with Exponentially distributed inter-arrival times, the objective is to schedule the task executions and potential checkpoints such that the expected completion time (or makespan) is minimized. Due to delays resulting from the faults, the schedule of the tasks consists of an assignment to processors and of a task ordering. Each processor executes tasks as soon as possible and resumes their processing when a failure strikes. Finally, the schedule of the checkpoints is the (possibly empty) list of files that must be checkpointed after each task execution.

4 Scheduling and checkpointing algorithms

In this section, we first present heuristics to map tasks to processors. Then we propose three different checkpointing strategies that can be used simultaneously.

4.1 Scheduling heuristics

We map tasks to processors and schedule them using two classical scheduling heuristics, HEFT (Topcuoglu et al. (2002))[†] and MINMIN (Braun et al. (2001)). We run these heuristics as if the platforms were not subject to failures, that is, without considering checkpoints. Therefore, we decide first on which processor a task will be executed, and the order in which a processor will execute tasks, before deciding when and what to checkpoint (see Section 4.2). However, we present variants of HEFT and MINMIN, named HEFTC and MINMINC, that are specifically designed for our failure-prone framework.

Heterogeneous Earliest Finish Time first (HEFT) is presented as the HEFTC variant in Algorithm 1. The original HEFT algorithm comprises two phases. In a first *task prioritizing phase*, the bottom-level of all tasks is computed and tasks are ordered by non-increasing bottom-levels. The bottom-level of a task is the maximum length of any path starting at the task and ending in an exit task, considering that all communications take place (Darte et al. (2000)). In the second *processor selection phase*, the first unscheduled task is scheduled as early as possible on a processor that minimizes its completion time. In all cases, ties are broken arbitrarily. To these original two phases, we add a third one, the *chain mapping phase* (lines 7 and 8 of Algorithm 1). If the newly mapped task T is the head of a chain in the task graph, which means T only has one successor whose only predecessor is T, then this whole chain is mapped on the same processor as T, and the tasks will be executed consecutively. Ensuring that entire chain of tasks are scheduled on the same processor decreases the number of crossover dependences and thus, the time to checkpoint them. HEFTC has a complexity of $O(n^2)$ for a workflow with n tasks. During the processor selection phase, the earliest finish time of a task is computed in HEFTC while assuming that the newly mapped task must start *after* all tasks previously scheduled on that processor have completed. On the contrary, the original HEFT heuristic is allowed to perform backfilling following a classical insertion-based policy, as long as the completion time of no task is delayed. Allowing backfilling is more expensive at scheduling time but should lower the execution time (the complexity of HEFT with backfilling is also $O(n^2)$ with homogeneous processors). We do not allow backfilling for HEFTC because it could be antagonistic to the chain mapping phase if it led to backfill the head of the chain, but not the whole chain.

Algorithm 1: HEFTC

```
1 Compute the bottom-level of all tasks by traversing
    the graph from the exit tasks
2 Sort the tasks by non-increasing values of their
    bottom-levels
3 while there are unscheduled tasks do
4     Select the first task T_i
5     k ← arg min_{1≤k≤p} EarliestFinishTime(T_i, P_k)
6     Schedule task T_i on processor P_k
7     if T_i is the head of a chain of tasks then
8         Schedule the whole chain continuously on P_k
```

The MINMIN scheduling algorithm is presented in the MINMINC variant in Algorithm 2. The original MINMIN algorithm is a simple loop which, at each step, schedules the task that can finish the earliest among unscheduled tasks. Therefore, at each step it considers all ready tasks and, for each of them, all the processors. We (try to) improve this heuristic by adding a *chain mapping phase* exactly as previously (lines 5 and 6 of Algorithm 2). MINMINC has a complexity of $O(n^2p)$ for a workflow with n tasks and p processors.

Algorithm 2: MINMINC

```
1 ReadyTasks ← entry tasks
2 while there are unscheduled tasks do
3     Pick a task T ∈ ReadyTasks and a processor P
        such that the completion time of T on P is
        minimum among the Earliest Finish Times of all
        ready tasks
4     Schedule task T on processor P
5     if T is the head of a chain of tasks then
6         Schedule the whole chain continuously on P
7     Update ReadyTasks
```

4.2 Checkpointing strategies

While the previous scheduling algorithms provide mappings of tasks to processors, it remains to decide which files must be checkpointed and when. This section introduces finer strategies than the two extremes solutions that consist of checkpointing no task or all tasks. These two extreme solutions, CKPTNONE and CKPTALL, are denoted with the suffixes NONE and ALL, respectively.

The minimum strategy that is required to isolate processors consists in checkpointing all files that must be transferred between any pair of processors, i.e., exactly the files corresponding to crossover dependences. In this case, any failure on a processor will not require any re-execution on other processors. The strategy is denoted with a "C" in the checkpoint suffix.

For the next two additional strategies, we introduce a new type of checkpoints: *task checkpoints*. While a simple file checkpoint consists of writing to stable storage a file that corresponds to a dependence between two tasks, a task checkpoint consists of writing all files that (i) reside in memory on a processor; (ii) will be used later by tasks assigned to the same processor; and (iii) have not already been checkpointed. In the example in Section 2, for each crossover dependence we did a simple file checkpoint rather than a full task checkpoint. A task checkpoint after task T_3 would have also checkpointed the file corresponding to the dependence $T_3 \to T_5$. A non-trivial task checkpoint for the example of Section 2 would be a task checkpoint for task T_2. This checkpoint would require checkpointing the files corresponding to the dependences $T_2 \to T_4$ and $T_1 \to T_7$.

When a task checkpoint is performed after the execution of a task, multiple files may be checkpointed "at the same time" (either newly created files or previously created ones that will later be used). If several files are checkpointed, they are all checkpointed after the task completion, one after the other (in any order). When absent from memory (following a failure), input files are read from stable storage as late as possible, just before the execution of the task that needs them.

Checkpointing crossover dependences enable to isolate processors, in that there is no re-execution propagation from a processor to another. However, when a task is the target of a crossover dependence, its starting time is the maximum of the availability times of all its input files, and these files come from different processors. Therefore, its starting time may be delayed by failures occurring on other processors. Because failures can strike during idle time, it may be beneficial to try to use the potential waiting time by performing a task checkpoint of the task preceding the target task. This way, the whole content of the memory will be preserved, the cost of the checkpoint may be offset by some waiting time, and if a failure strikes during the remaining waiting time all input files remain available. Therefore, we propose a new checkpointing strategy denoted with "I" in the checkpoint suffix. This strategy consists of checkpointing all *induced* dependences. A dependence $T_i \to T_j$ is an *induced* dependence if T_i and T_j are scheduled on the same processor P and there exists a crossover dependence $T_k \to T_l$ such that T_l is scheduled on P after T_i and before T_j (or $T_l = T_j$). Checkpointing these induced dependences is done by performing a task checkpoint of the task preceding T_l on P. In the example of Section 2, the dependences $T_2 \to T_4$ and $T_1 \to T_7$ are both induced dependences because of the crossover dependence $T_3 \to T_4$. As a side note, we point out that our *induced* checkpoints are not related to those introduced in (Baldoni et al. (1997)) to track consistency among execution traces of general applications: we deal with simple task graphs and can easily regenerate missing data by following the dependences.

So far, we have only introduced checkpoints to isolate processors, either to avoid failure propagation or to try to minimize the impact of processors having to wait from each other. We further consider checkpoints that more directly optimize expected total execution time. We present an additional strategy, denoted by the suffix "DP", which adds additional checkpoints through a $O(n^2)$ dynamic programming algorithm, which is a transposition of that of (Han et al. (2018a)). This dynamic program considers a maximal sequence of consecutive tasks that are all assigned to the same processor, and that are isolated from other tasks: the sequence contains no checkpoint and none of its tasks is the target of a crossover dependence, except for its first task. Let $T_1, ..., T_k$ be such a sequence of tasks. By definition, all input data produced by some previous tasks have been checkpointed. Then, the optimal expected time to execute this sequence is given by $Time(k)$ where $Time$ is defined as follows:

$$Time(j) = \min\left(T(1,j), \min_{1\leq i<j} Time(i) + T(i+1,j)\right)$$

where $T(i,j)$ is the expected time to execute tasks T_i to T_j provided that two task checkpoints are performed: one right before task T_i and one right after task T_j. Using the same reasoning as in Section 3.2, we can provide an upper bound on $T(i,j)$ as follows:

$$T(i,j) = \left(\frac{1}{\lambda} + d\right)\left(e^{\lambda(R_i^j + W_i^j + C_i^j)} - 1\right)$$

where R_i^j (resp. W_i^j and C_i^j) is the sum of the recovery (resp. execution and checkpointing) costs of tasks T_i to T_j. The recovery costs concern all input files of these tasks that are on the stable storage, while the checkpointing costs concern all files that will be checkpointed when a task checkpoint is done after T_j. This is an upper bound, because when no failure strikes, some input files of tasks T_i to T_j may already be present in memory and will not be read from stable storage. Because we have no simple mean to know whether some failures had previously struck, we have to resort to this upper bound. This is a necessary condition to be able to reuse, in some way, the dynamic programming approach of (Han et al. (2018a)). This algorithm requires, by construction, that induced dependences be checkpointed. However, we heuristically use it even when this condition is not satisfied. In this case, we take a maximal sequence while allowing tasks to be the target of crossover dependences, and behave as if these crossover dependences were not existing: we discard any potential waiting time that may be due to these crossover dependences (because we have no means to estimate them).

5 Experiments

In this section, we describe the experiments conducted to assess the efficiency of the checkpointing strategies.

In Subsection 5.1, we describe the parameters and applications used during our experimental campaign, then in Subsection 5.2 we present the simulator used to run the applications and simulate the behavior of large-scale platforms. Finally, we present our results in Subsection 5.3.

5.1 Experimental methodology

We consider workflows from real-world applications, namely representative workflow applications generated by the Pegasus Workflow Generator (PWG) (da Silva et al. (2014); Bharathi et al. (2008); Juve et al. (2013)), as well as the three most classical matrix decomposition algorithms (LU, QR, and Cholesky) (Choi et al. (1996)), and randomly generated DAGs from the Standard Task Graph Set (STG) (Tobita and Kasahara (2002)).

Pegasus workflows. PWG uses the information gathered from actual executions of scientific workflows as well as domain-specific knowledge of these workflows to generate representative and realistic synthetic workflows (the parameters of which, e.g., the total number of tasks, can be chosen). We consider all of the five workflows (Pegasus (2014)) generated by PWG, including three M-SPGs (GENOME, LIGO, and MONTAGE) that are used to compare our new general approach with PROPCKPT, the strategy for M-SPGs proposed in (Han et al. (2018a)).

- MONTAGE: The NASA/IPAC Montage application stitches together multiple input images to create custom mosaics of the sky. The average weight of a MONTAGE task is 10s. Structurally, MONTAGE is a three-level graph (Deelman et al. (2005)). The first level (reprojection of input image) consists of a bipartite directed graph. The second level (background rectification) is a bottleneck that consists of a join followed by a fork. Then, the third level (co-addition to form the final mosaic) is simply a join.
- LIGO: LIGO's Inspiral Analysis workflow is used to generate and analyze gravitational waveforms from data collected during the coalescing of compact binary systems. The average weight of a LIGO task is 220s. Structurally, LIGO can be seen as a succession of Fork-Joins meta-tasks, that each contains either fork-join graphs or bipartite graphs.
- GENOME: The epigenomics workflow created by the USC Epigenome Center and the Pegasus team automates various operations in genome sequence processing. The average weight of a GENOME task depends on the total number of tasks and is greater than 1000s. Structurally, GENOME starts with many parallel fork-join graphs, whose exit tasks are then both joined into a new exit task, which is the root of fork graphs.
- CYBERSHAKE: The CYBERSHAKE workflow is used by the Southern Calfornia Earthquake Center to characterize earthquake hazards in a region. The average weight of a CYBERSHAKE task is 25s. Structurally, the CYBERSHAKE workflow starts with several forks. Then each of the forked tasks has two dependences: one to a single task (join) and one to a specific task for each of the tasks. Finally, all these new tasks are joined without another dependence this time.
- SIPHT: The SIPHT workflow, from the bioinformatics project at Harvard, is used to automate the search for untranslated RNAs (sRNAs) for bacterial replicons in the NCBI database. The average weight of a SIPHT task is 190s. Structurally, the SIPHT workflow is composed of two different parts that are joined at the end: the first one is a series of join/fork/join, while the other is made of a giant join.

We generate these workflows with 50, 300, and 700 tasks (these are the number of tasks given to the generator, the actual number of tasks in the generated workflows depend on the workflow shape). The task weights and file sizes are generated by PWG. In some instances, a single file may be used by more than one task and a dependence may represent multiple files to transfer between two tasks. In the first case, whenever a file is common to multiple dependences, the file is only saved once. In the second case, files are aggregated into a single one.

Matrix factorizations. We consider the three most classical factorizations of a $k \times k$ tiled matrix: LU, QR, and Cholesky factorizations.

- The LU decomposition is the factorization of any matrix into a product of one lower-triangular (L) and one upper-triangular (U) matrices. Structurally, the DAG is made of k steps, with at step i, one task having two sets of $k-i-1$ children, and each pair of tasks between the two sets having another child.
- The QR decomposition is the decomposition of a matrix into a product of an orthogonal matrix (Q) and upper-triangular matrix (R), i.e., $A = QR$ with $QQ^T = Id$. Structurally, the QR decomposition looks like the LU decomposition but it has more complex dependences between the $k-i-1$ children at step i.
- Cholesky is a factorization of a positive and definite matrix into the product of a triangular matrix and its transpose, i.e., $A = BB^T$ where B is lower-triangular and has non-zero values of the diagonal. The Cholesky decomposition DAG is the representation of a panel algorithm and can be constructed recursively by removing the first row and the first column of submatrices, to keep factorizing the trailing matrix.

For each factorization, we perform experiments with $k = 6$, 10, and 15, for a total of $3 \times 3 = 9$ DAGs with up to 1240 tasks. The number of vertices in the DAG depends on k as follows: the Cholesky DAG has $\frac{1}{3}k^3 + O(k^2)$ tasks, while the LU and QR DAGs have $\frac{2}{3}k^3 + O(k^2)$ tasks. There are 4 types of tasks in LU, QR, and Cholesky, which are labeled by the corresponding BLAS kernels (Choi et al. (1996)), and their weights are based on actual kernel execution times as reported in (Augonnet et al. (2011)) for an execution on Nvidia Tesla M2070 GPUs with tiles of size $b = 960$.

Random graphs. The STG benchmark (Tobita and Kasahara (2002)) includes 180 instances for each size of DAGs (from 50 to 5 000). This set is often used in the literature to compare the performance of scheduling strategies. Instead of choosing part of the instances for each size, we did experiments on all instances of size 300 and 750. For each instance, one of the four DAG generators specifies the structure of the dependences (e.g., layer-by-layer) and one of the six cost generators provides the distribution of the processing times (e.g., uniform).

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

Checkpointing costs. 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. We define the Communication-to-Computation Ratio (CCR) as the time needed to store all the files handled by a workflow (input, output, and intermediate files) divided by the time needed to perform all the computations of that workflow on a single processor. For Pegasus workflows, LU, QR, and Cholesky, we vary the CCR by scaling file sizes by a factor. As STG only provides task weights, we compute the average communication cost as $\bar{c} = \bar{w} \times CCR$. Communication costs are generated with a lognormal distribution with parameters $\mu = \log(\bar{c}) - 2$ and $\sigma = 2$ to ensure an expected value of $\bar{c}$. This distribution with parameter $\sigma = 2$ has been advocated to model file sizes (Downey (2001)). This allows considering and quantifying the data-intensiveness of all workflows in a coherent manner across experiments and workflow classes and configurations.

Reference strategies. In the experiments, we compare our strategies to the two extreme approaches CKPTALL and CKPTNONE. We use the simulator described in Subsection 5.2. For each parameter setting of each workflow, we run 10,000 random simulations and approximate the makespan by the observed average makespan.

5.2 Simulator

In order to evaluate the performance of our strategies, we implemented a discrete event simulator. The C++ code for the simulator is available at `http://github.com/vlefevre/task-graph-simulation`. To simulate the execution of applications on large-scale platforms, we operate in three steps:

1. We first read an input file describing the task-graph and the scheduling/mapping strategy;
2. Then we generate a set of fail-stop error times for each processor during a time `horizon` (that is set by the user);
3. Finally, we execute ready tasks by mapping them to a processor and we keep doing this until all tasks are executed.

The first part is basically reading a file that describes the following important elements for the simulation:

- For each task,
  - its ID,
  - its weight (i.e., duration),
  - the ID of the processor it has been mapped to,
  - several booleans indicating whether the task has to be checkpointed or not, one for each checkpointing strategy.
- For each dependence between two tasks,
  - the ID of the parent,
  - the ID of the child,
  - the list of files with their time to be loaded/written that creates the dependence (i.e., there are some of the output files of the parent and some of the input files of the child).
- For each processor, its schedule: a list of tasks that have been mapped to it and that respects the causal order of the task-graph.

The second part is done by using the inversion sampling method: we generate error times according to a random variable that follows an exponential distribution, and this exponential distribution is generated from an (assumed) uniform distribution between 0 and 1 obtained by calling the C function `rand()`, and dividing its result by the C constant `RAND_MAX`. In our case, if U is a random variable following a uniform distribution between 0 and 1, then $-\frac{\log U}{\lambda}$ follows an exponential distribution of parameter λ. We generate errors on each processor, until the time of one error is greater than the `horizon` parameter. In the experiments, it was set to at least 2 times the expected makespan we have with the CKPTALL strategy, which we computed using the Monte-Carlo method. In practice, most of the simulations were done before the horizon was reached except for NONE with large p_{fail}.

For the last step, we keep a global time t on all the processors, and we generate events happening on each processor (either a failure or the successful completion of a task). Each processor holds the time of its last event in a variable t_i. At each moment of the simulation, we have $t_i \geq t, \forall i$. The algorithm repeats these steps until all tasks are marked executed:

- For each processor p_i,
  - we look at the next task to be executed on p_i (following the list scheduling given as input) if the current task is finished at time t;
  - if it is ready, we compute its full execution time by computing the time of reading the necessary input files, the weight of the task (given as input) and potentially some writing (in case of crossover dependences or if the checkpoint strategy requires this task to be checkpointed);
  - we look at the next error happening after time t: if it is before the end of the task then we set t_i to be the time of that failure, otherwise t_i is set to the time when the task ends and the task is marked executed.
- We set $t = \min_i t_i$.

There are two more things to detail: the computation of reading times and how we rollback when there is a failure. For the first problem, we keep a set of all files loaded on each processor. Before reading an input file, we check if it is already loaded (i.e., belongs to that set). If it is already loaded, we count a cost of 0, otherwise we add the communication/reading time for that file that is given as input. Files are added to the set whenever they are loaded

or written (not necessarily a checkpoint). The set is cleared whenever a fail-stop error strikes on the processor.

When there is a failure, the rollback is easy because we always checkpoint crossover dependences. This implies that a failure on a processor p_i will only impact the tasks that have been executed on p_i since the last checkpointed task that was mapped to p_i. To rollback we explore the list of tasks backward from the current task to the last checkpointed one (we keep two pointers on these two tasks at each time to access them instantaneously), we mark each task unexecuted, we clear the set of loaded files and we can start simulating again from the last checkpointed task as if nothing happened. In the case of CKPTNONE, the simulation is rolled back from the first task anytime an execution or communication is interrupted.

Finally, the simulator computes the following measures: the number of file checkpoints taken, the number of task checkpoints taken, the number of failures, the total time spent checkpointing data and the execution time of the application.

5.3 Results

In this section, we first compare the expected makespan of our proposed checkpointing strategies (CDP and CIDP) over two baseline strategies (ALL and NONE) with the same task mapping and scheduling strategy. Remind that "C", "I" and "DP" stands for checkpointing crossover dependences, checkpointing induced dependences and dynamic programming respectively. Then, we compare the solutions (different task mapping and scheduling heuristics combined with several checkpointing strategies) from this work with the method PROPCKPT proposed in (Han et al. (2018a)) for M-SPGS.

In Figures 6-10, we compare the four considered task mapping and scheduling strategies: HEFT and MINMIN, with their chain-mapping variants HEFTC and MINMINC using boxplots[‡]. On these figures, the lower the better and the baseline at 1 is the performance of HEFT. The chain-mapping variants have the same performance or improve that of their basic counterparts, especially when communications are expensive (rightmost parts of the graphs). The other conclusion is that MINMIN (resp. MINMINC) almost always achieves same or worse performance than HEFT (resp. HEFTC). This is easily explained by the fact that HEFT and HEFTC take into account the critical path of workflows. These trends are representative of the trends that can be observed for all considered graphs and workflows, but suffer from some exceptions. The chain-mapping variants can be superceded by their basic counterparts for workflows that do not include any chains (like LU in Figure 7), because the basic variants can use backfilling. However, backfilling sometimes backfires, even in the absence of chains, like for SIPHT in Figure 9 where HEFTC can decrease the expected makespan by more than 30% with respect to HEFT. Overall, of the four considered task mapping and scheduling heuristics, HEFTC never achieves significantly bad performance, and most of the time achieves the best performance. This is the reason why we focus on it in the remainder of this section.

Figures 11 through 18 present the expected makespans achieved by CDP, CIDP and NONE divided by that of ALL when the Communication-to-Computation Ratio increases. Therefore, the lower the better and data points below the $y = 1$ line denote cases in which these strategies outperform the competitor ALL (i.e., achieve a lower expected makespan). Each figure shows results for workflows with different number of tasks, ranging from 50 to 1240 tasks (each line of subfigure is for a different size, the number of tasks being reported on the rightmost column), for various number of processors P (different line styles), and for the three p_{fail} values (0.0001, 0.001, 0.01). We report on these figures the average number of failures that occur for the 10,000 random trials for each setting. These numbers are reported in black above the horizontal axis in each figure. The other two lines of numbers are the number of checkpointed tasks for the CDP and CIDP strategies, each number is printed with the same color as the curve of the corresponding strategy.

A clear observation is that CIDP never achieves worse performance than ALL: either it achieves a similar performance or it outperforms ALL, especially when communications, and thus checkpoints, are expensive (in the rightmost parts of graphs). It should be noted that when checkpoints come for free (leftmost parts of graphs), ALL and CIDP have the same performance as they do the same thing: they checkpoint all tasks. When the number of failures rises, the optimal solution is to checkpoint more tasks, potentially all of them, and the gain of CIDP with respect to ALL therefore decreases. This can be seen, for instance, on Figure 12 when $p_{\text{fail}} = 0.01$, $n = 385$ and there are 385 tasks checkpointed.

In the majority of cases, CDP also achieves similar or better performance than ALL. As we explained in Section 4, the dynamic programming algorithm is well-defined for CIDP, which checkpoints all induced dependences. However, CDP tries to save some checkpointing overhead by not systematically checkpointing induced dependences. As a consequence, the dynamic programming algorithm estimations of expected execution times may be inaccurate, which explains the sometimes bad performance of CDP. There are only a couple of CCR values for CYBERSHAKE for which CDP achieves a significantly worse performance than ALL. On the contrary, CDP often has better performance than CIDP when checkpointing cost is high. In all scenarios, CDP checkpoints less or the same number of tasks than CIDP. Depending on the checkpointing cost and failure rate, CDP can lead to significant improvement over ALL. For workflows as dense as LU, we save more than 10% when $CCR = 1$ for both strategies, and CDP even achieves 35% saving for SIPHT. As the CCR decreases, the ratio converges to 1. As already pointed out, this is because both strategies decide to checkpoint most, if not all, tasks, when checkpointing becomes cheaper.

CDP and CIDP achieve better results than NONE except when (i) checkpoints are expensive (high CCR) and/or (ii) failures are rare (low p_{fail}). In these cases, checkpointing is a losing proposition, and yet our strategies, by design, always checkpoints some files (they checkpoint all crossover files and even induced dependences for CIDP). In practice, in such cases, the optimal approach is to bet that no failure will happen and to restart the whole workflow execution from scratch upon the very rare occurrence of a failure. NONE becomes worse whenever there are more failing tasks, i.e., when the failure rate increases (going from the leftmost

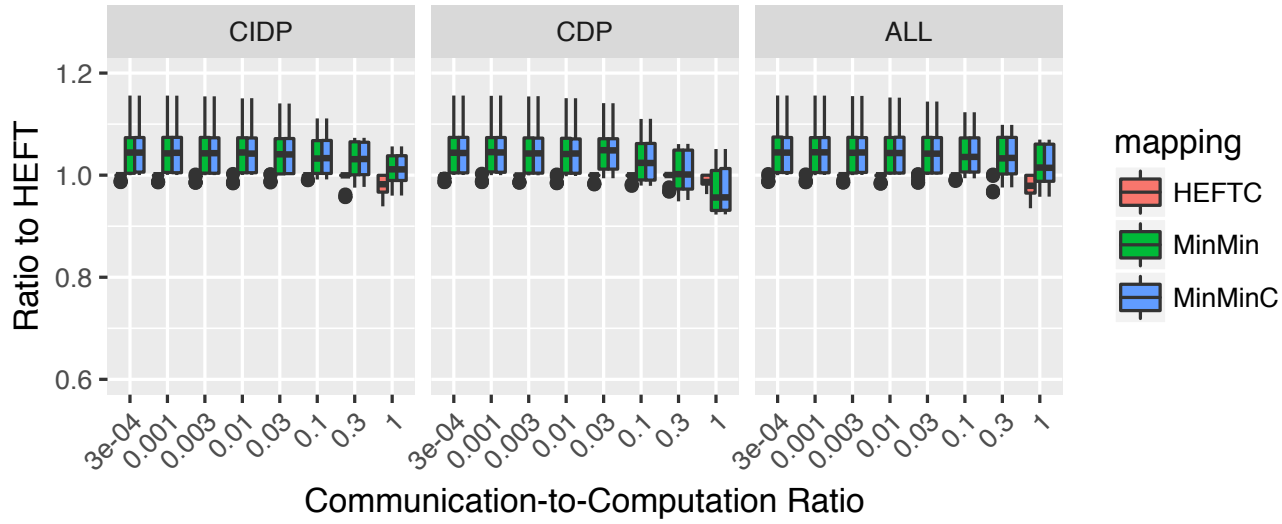

Figure 6. Relative performance of the four task mapping and scheduling strategies for Cholesky.

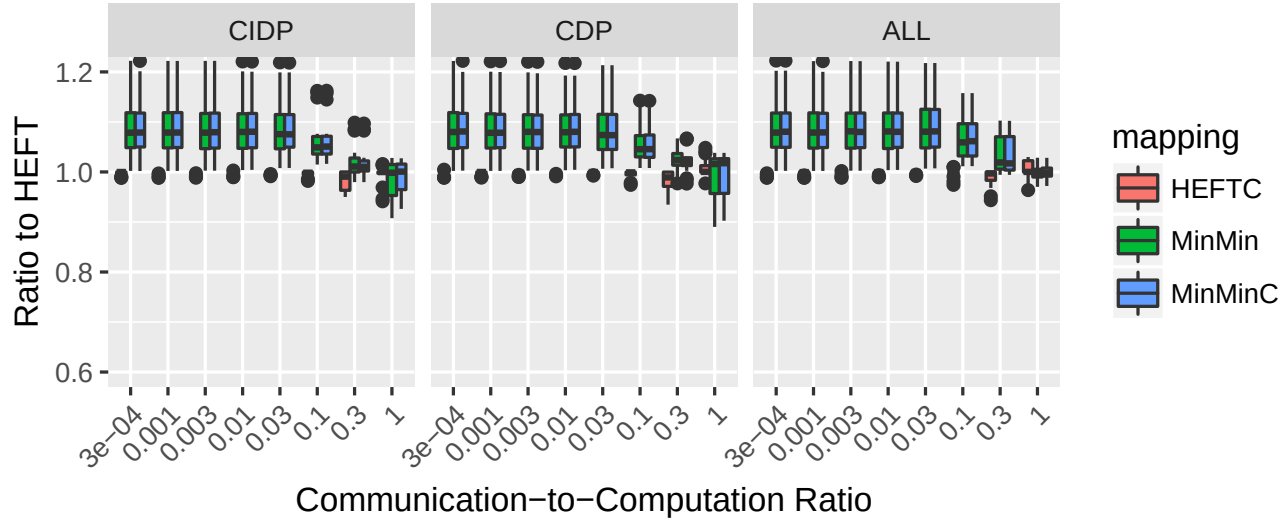

Figure 7. Relative performance of the four task mapping and scheduling strategies for LU.

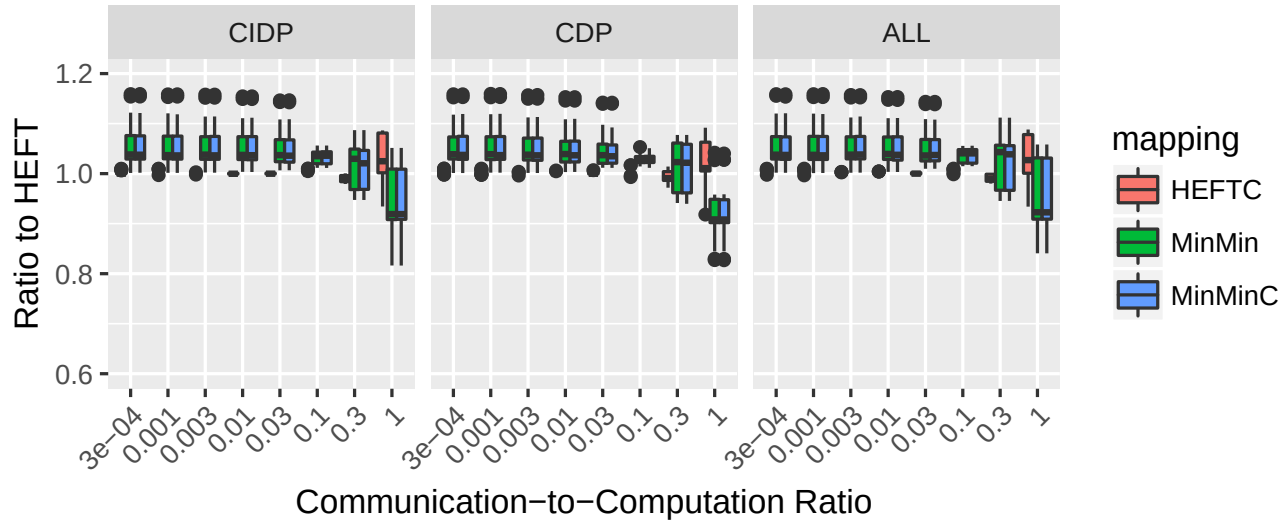

Figure 8. Relative performance of the four task mapping and scheduling strategies for QR.

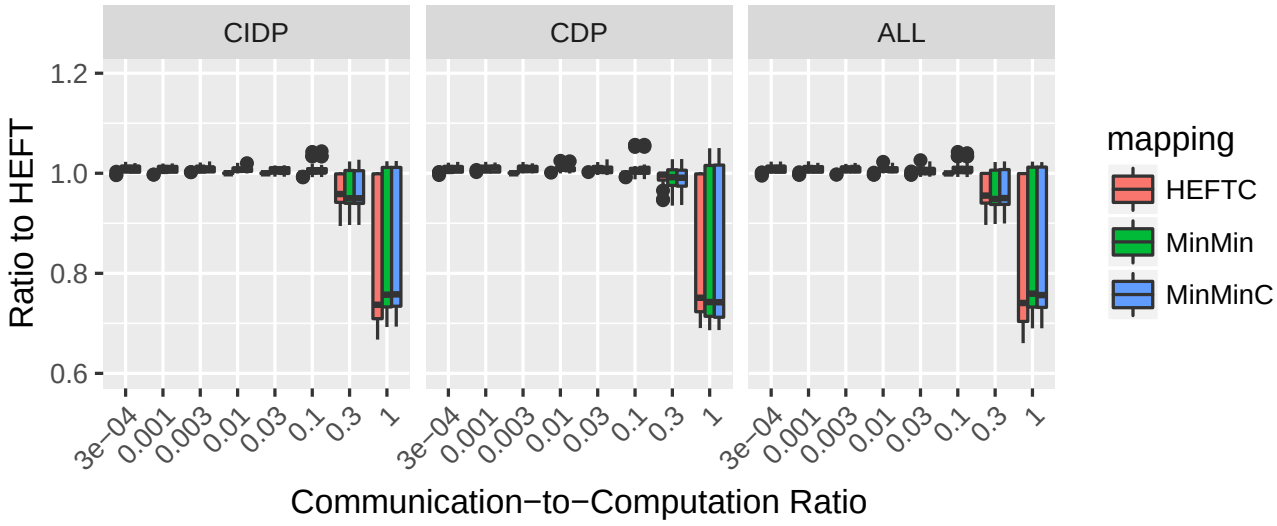

Figure 9. Relative performance of the four task mapping and scheduling strategies for Sipht.

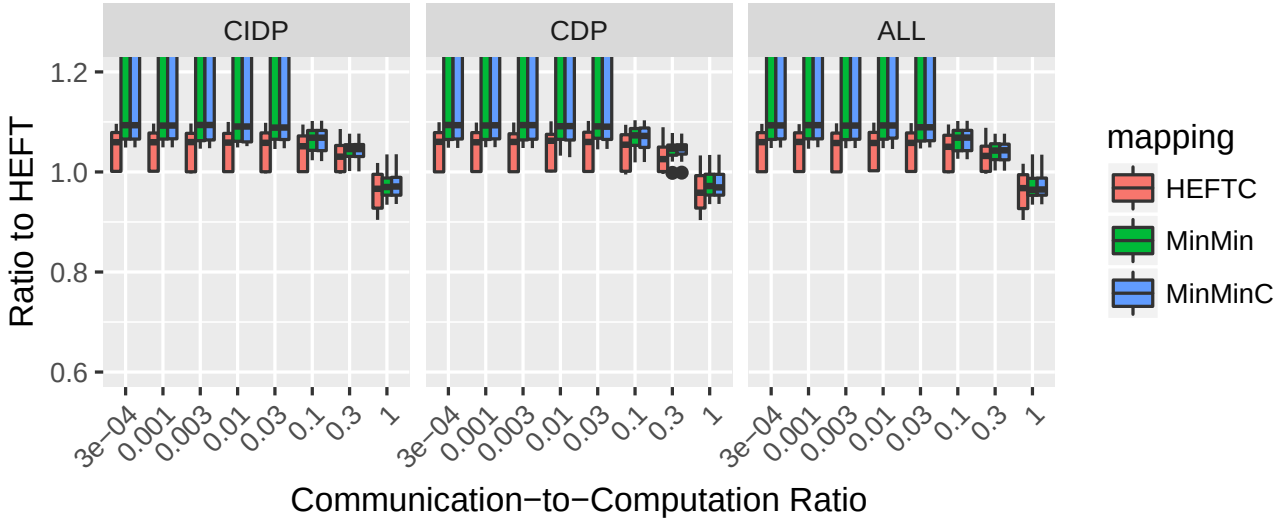

Figure 10. Relative performance of the four task mapping and scheduling strategies for CyberShake.

column to the rightmost one in the figures), and/or when the number of tasks increases (going from the topmost row to the bottom one in the figures). When the failure rate is high and the workflows are large (the bottom right corner of the figures), the relative expected makespan of NONE is so high that it does not appear in the plots. The above results, and our experimental methodology in general, make it possible to identify these cases so as to select which approach to use in practical situations.

Figure 19 presents the aggregated results for the 180 STG random DAGs with boxplots. The trends on these graphs are the same as already reported. This confirms the generality of our conclusions.

Finally, we compare our new general approach with PROPCKPT, the approach specific to M-SPGS that we proposed in (Han et al. (2018a)). Figures 20-22 present this comparison for Montage, Ligo and Genome, which are the three M-SPGS presented in (Han et al. (2018a)). Overall, the new approaches perform better than PROPCKPT.

6 Related work

Checkpointing workflows has received considerable attention in the recent years, but no satisfactory solution has yet been proposed for fail-stop failures and general DAGs.

Many authors (Cao et al. (2015); Jin et al. (2009); Kail et al. (2016)) have considered soft errors, by which a task execution fails but does not lead to completely losing the data present in the processor memory. Fail-stop errors have far more drastic consequences than soft errors as they induce the loss of all data present in memory. Therefore they require different solutions.

As discussed in Section 1, silent errors represent do not interrupt the execution of the task but corrupt its output data. Their net effect is the same, since a task must be re-executed whenever a silent error is detected. Their detection requires the use of some silent error detectors at the end of a task's execution. Two well-known examples of fault detectors are Algorithm-Based Fault Tolerance (ABFT) (Huang and Abraham (1984); Bosilca et al. (2009); Shantharam et al. (2012)) and silent error detectors based on domain-specific

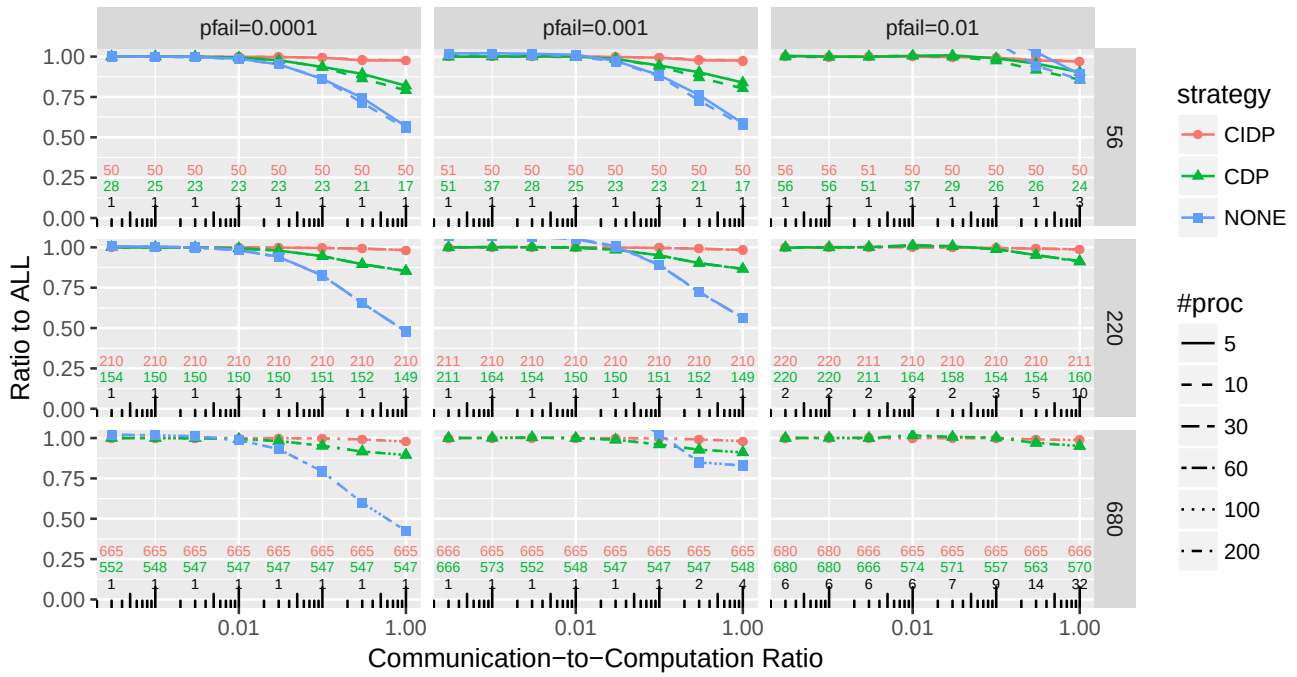

Figure 11. Performance of the different checkpointing strategies for Cholesky using HEFTC for task mapping and scheduling.

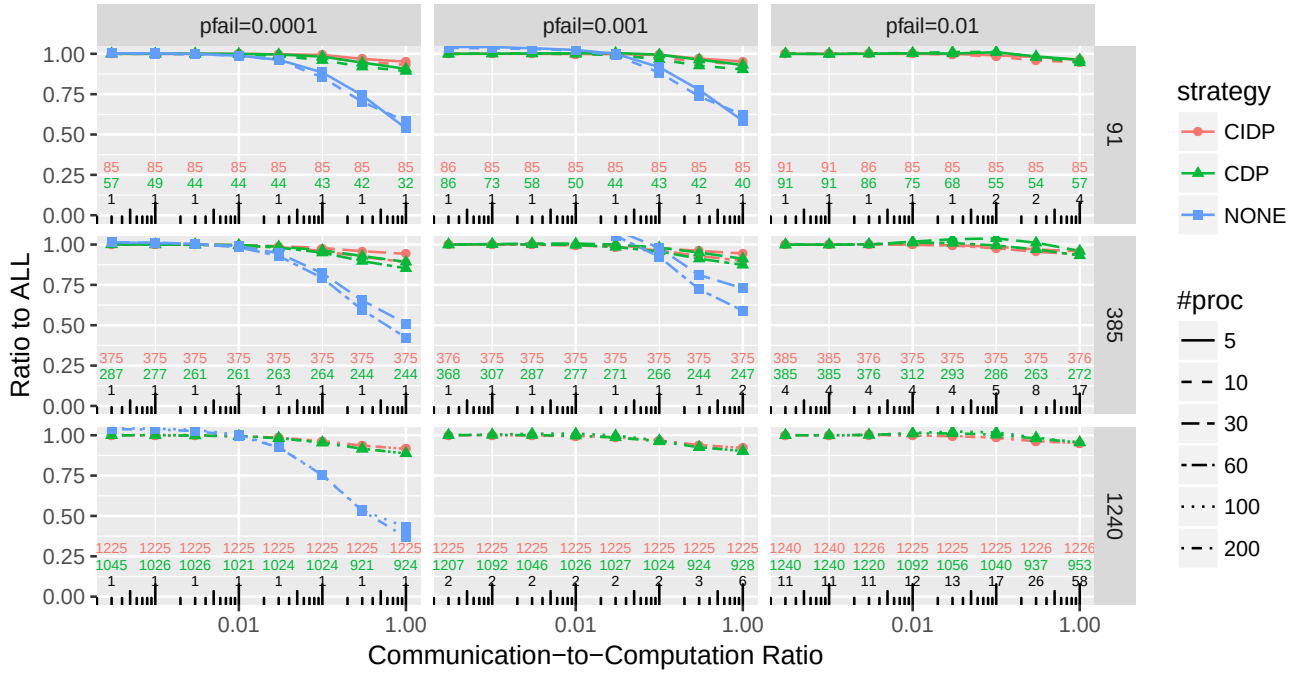

Figure 12. Performance of the different checkpointing strategies for LU using HEFTC for task mapping and scheduling.

data analytics (Berrocal et al. (2015); Bautista Gomez and Cappello (2014, 2015)). As we only consider fail-stop errors we do not need to use fault detectors.

Relatively few published works have studied fail-stop failures, rather than soft and silent errors, in the context of workflow applications. In (Zhu et al. (2016)), Zhu et al. proposed an automatic checkpoint algorithm on Spark to solve the long lineage problem. Because of the lazy feature of transformation operation, one could get the logical graph (lineage) before an action (submitting a job). Their solution is just tracing back the lineage, find and keep all the RDDs which are created in the job with direct parents in the previous job so they could recompute from these RDDs to get all the RDDs in this job. This method is based on specific domain knowledge. In (Lin et al. (2013)), they designed a checkpointing mechanism specifically for Map tasks, which creates a checkpoint at a specific percentage of a task in progress. Checkpoints are created when the progress reaches 0.5 or 0.25 by calculation progress rate and estimated task execution time. Hwang et al. (Hwang and Kesselman (2003)) presented a flexible framework for handling Grid failures. They divided workflow failure handling techniques into

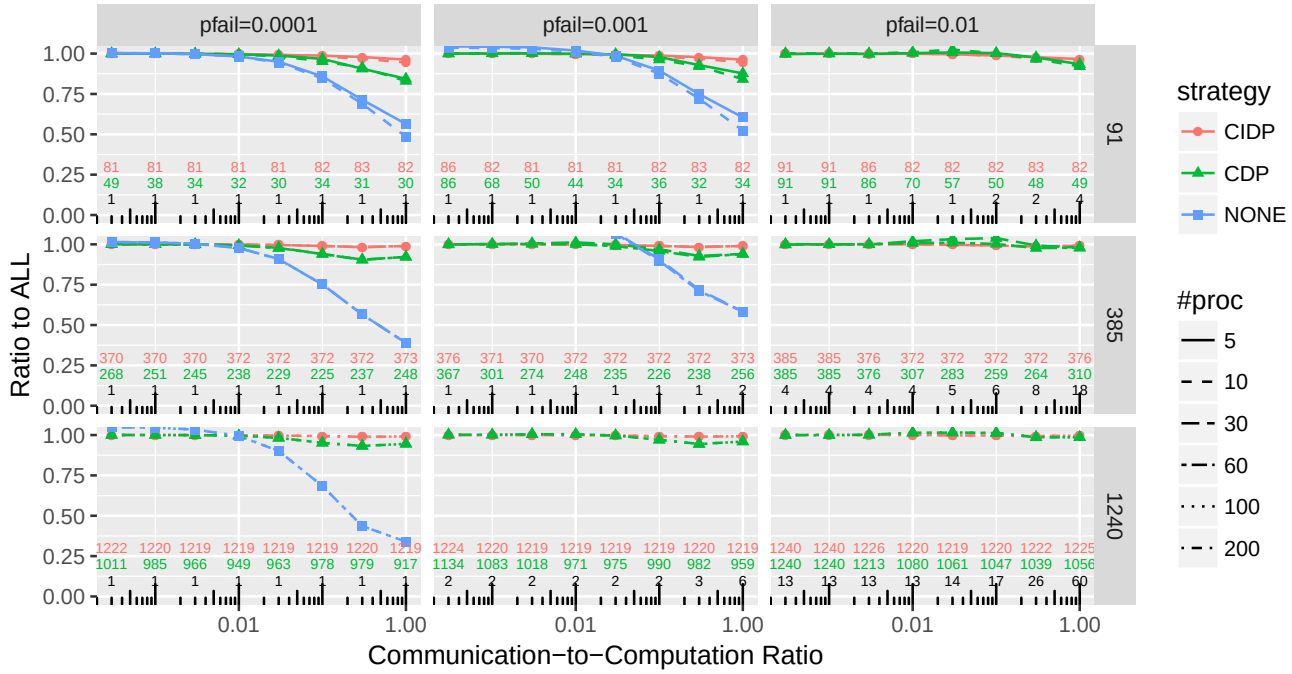

Figure 13. Performance of the different checkpointing strategies for QR using HEFTC for task mapping and scheduling.

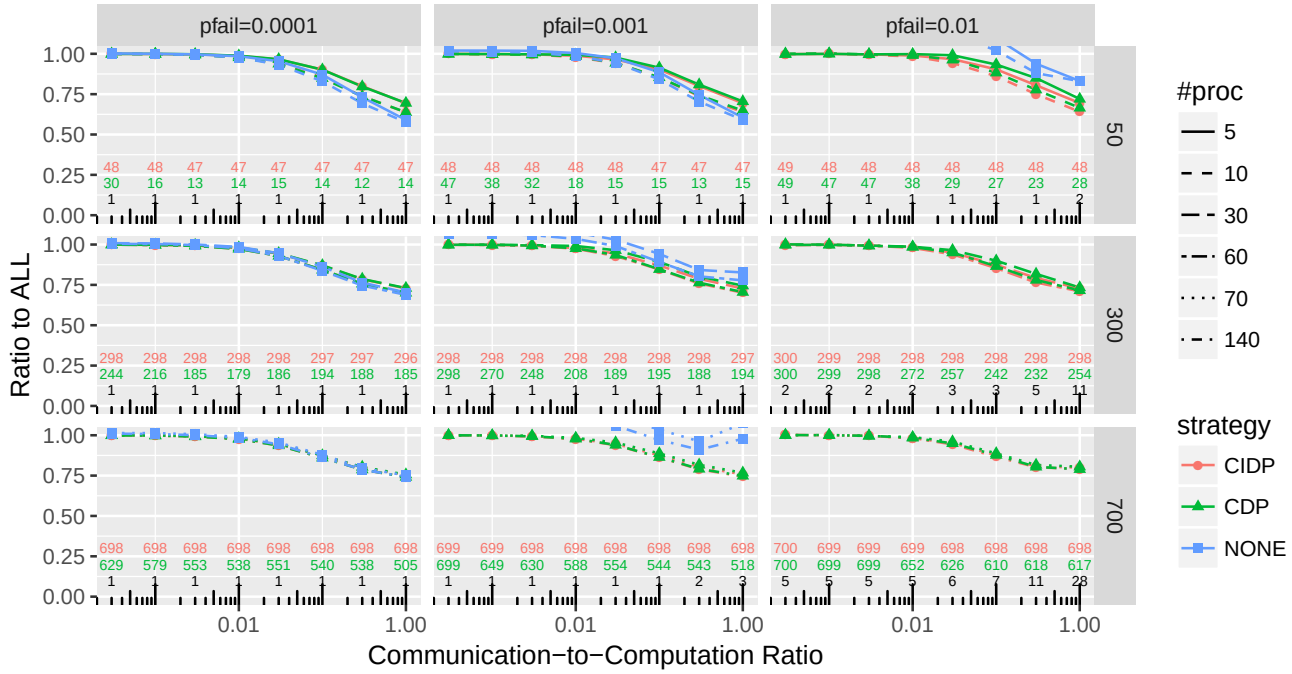

Figure 14. Performance of the different checkpointing strategies for Montage using HEFTC for task mapping and scheduling.

two different levels, namely task-level and workflow-level. Duan et al. (Duan et al. (2005)) presented the Distributed workflow Enactment Engine (DEE) of the ASKALON application.Their approach focuses on application-level checkpoints which triggered by precise checkpointing event defined. In (Tolosana-Calasanz et al. (2010)), they described an alternative workflow-level chekpointing scheme and every workflow node accomplishes its own local checkpoint. They employed the Petri nets to model the workflows which is a hierarchical non-DAG structure.

When the workflow consists of a linear chain of tasks, the problem of finding the optimal checkpoint strategy, i.e., determining which tasks to checkpoint, has been solved by Toueg and Babaoglu (Toueg and Babaoğlu (1984)) using a dynamic programming algorithm. The algorithm of (Toueg and Babaoğlu (1984)) was later extended in (Benoit et al. (2016)) to cope with both fail-stop and silent errors simultaneously. When the workflow is general but comprised of parallel tasks that each executes on the whole platform, the problem of placing checkpoints is NP-complete for simple join graphs (Aupy et al. (2016)) (this is because the original

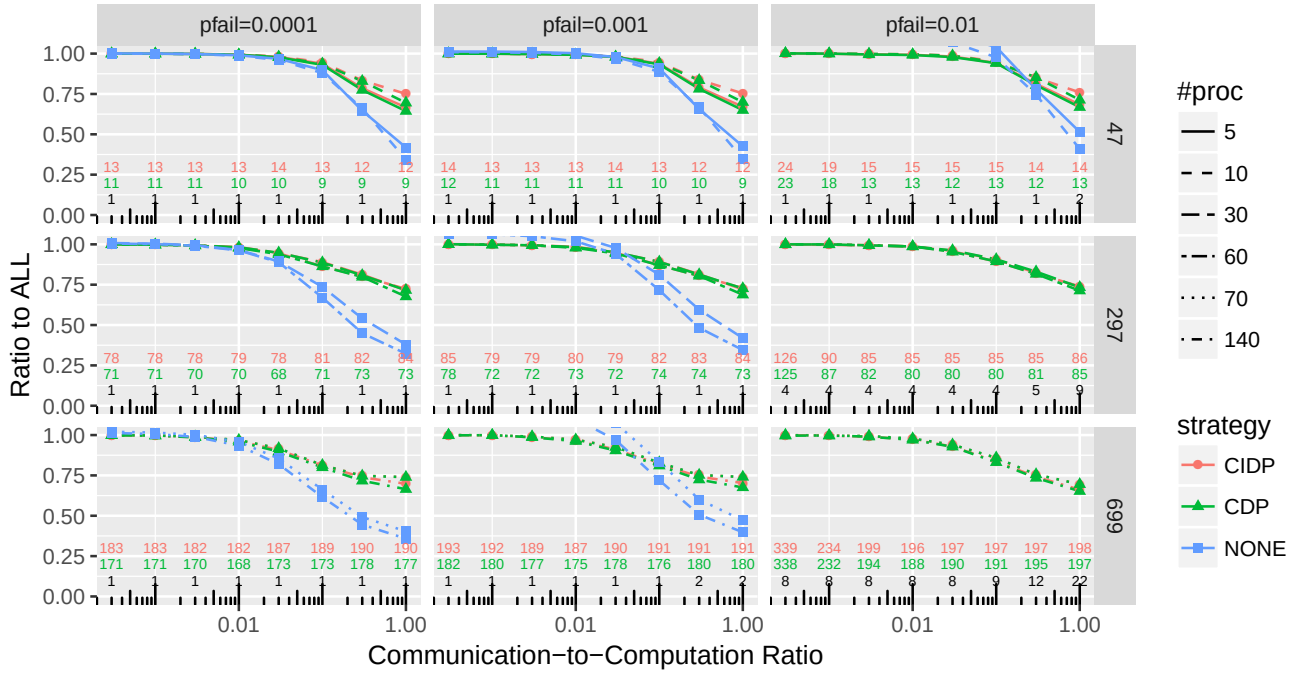

Figure 15. Performance of the different checkpointing strategies for Genome using HEFTC for task mapping and scheduling.

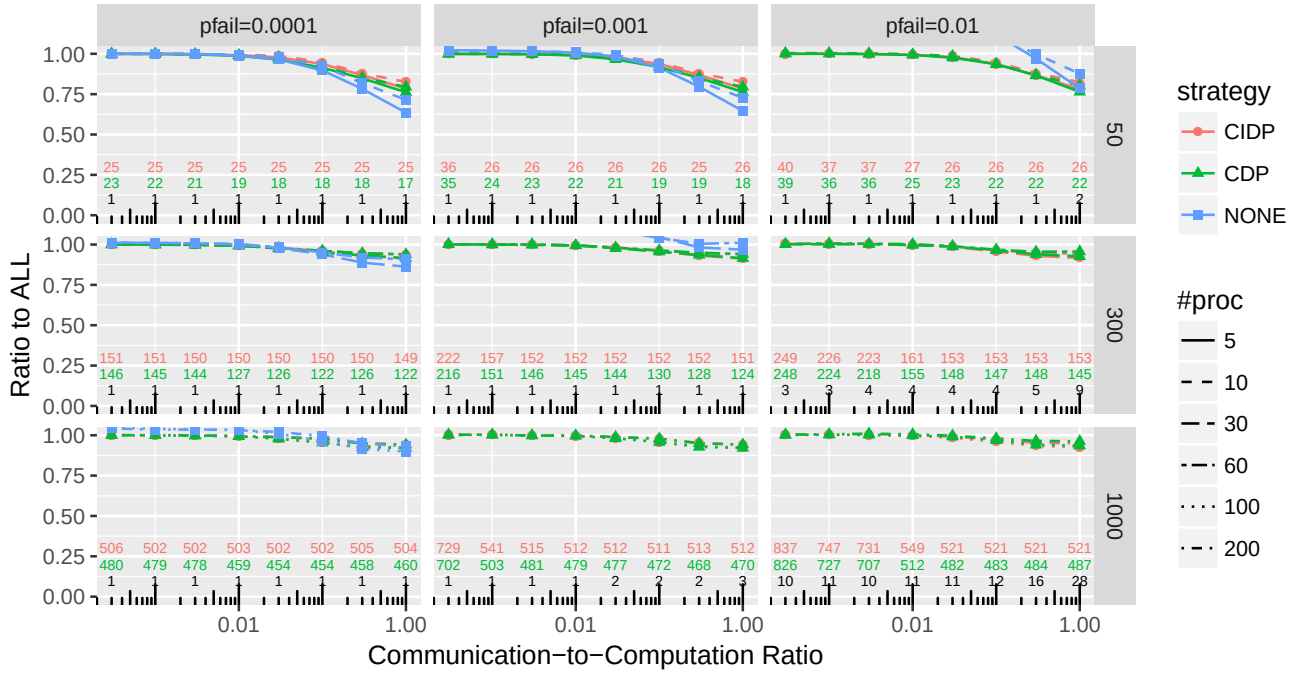

Figure 16. Performance of the different checkpointing strategies for Ligo using HEFTC for task mapping and scheduling.

workflow is not a chain but must be linearized). In the most general case, tasks of a workflow do not necessarily span the whole platform when executing. Existing work in this most general context diverges from ours as follows: either there is a limit to the number of failures that an execution can cope with (Wang et al. (2014)), or the optimization objective is reliability (Assayad et al. (2004)), meaning that the application execution can fail altogether. The only exception that we are aware of is our previous work (Han et al. (2018a)). The limitation of that work was different: the proposed solution could only deal with workflows whose structure was a Minimal Series-Parallel Graph (a generalization of Series-Parallel Graph).

To the best of our knowledge, this work is the first approach (beyond application-specific solutions) that (i) does not resort to linearizing the entire workflow as a chain of (parallel) tasks; (ii) can be applied to any workflow; (iii) can cope with an arbitrary number of failures; (iv) always guarantees a successful application execution; and (v) minimizes the (expectation of) the application execution time. As a result, we propose the first DAG scheduling/checkpointing algorithm that allows arbitrary

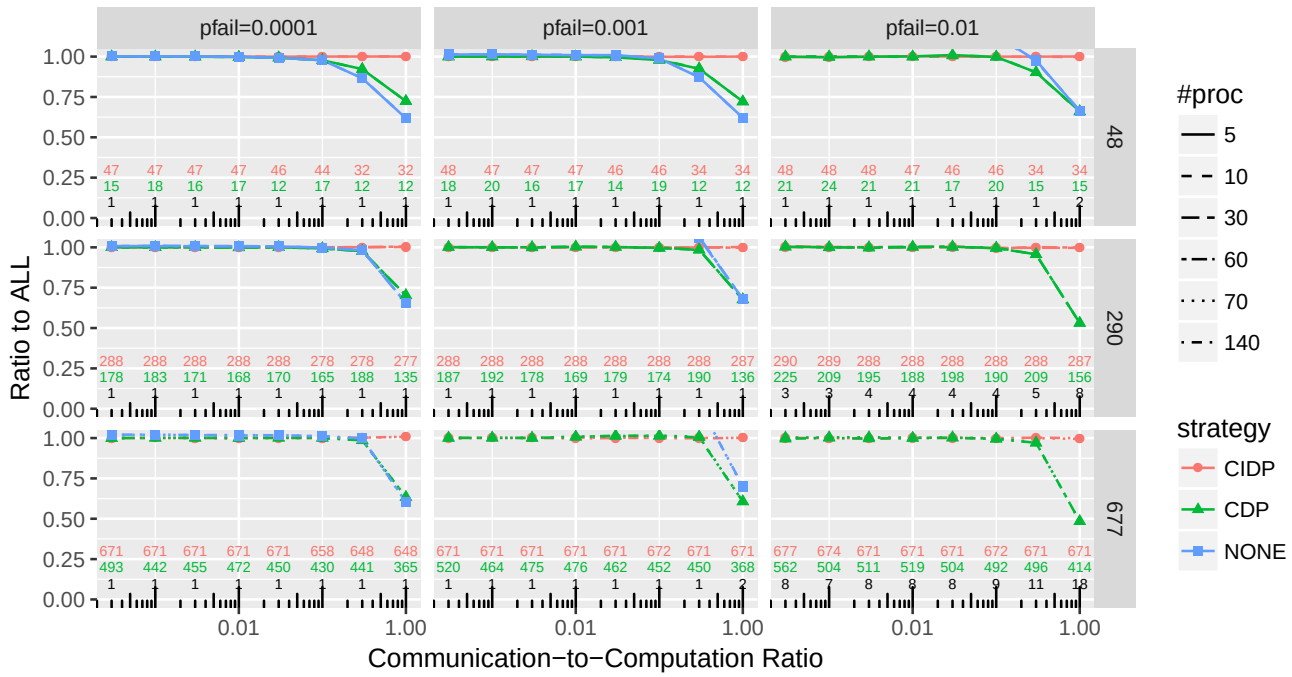

Figure 17. Performance of the different checkpointing strategies for Sipht using HEFTC for task mapping and scheduling.

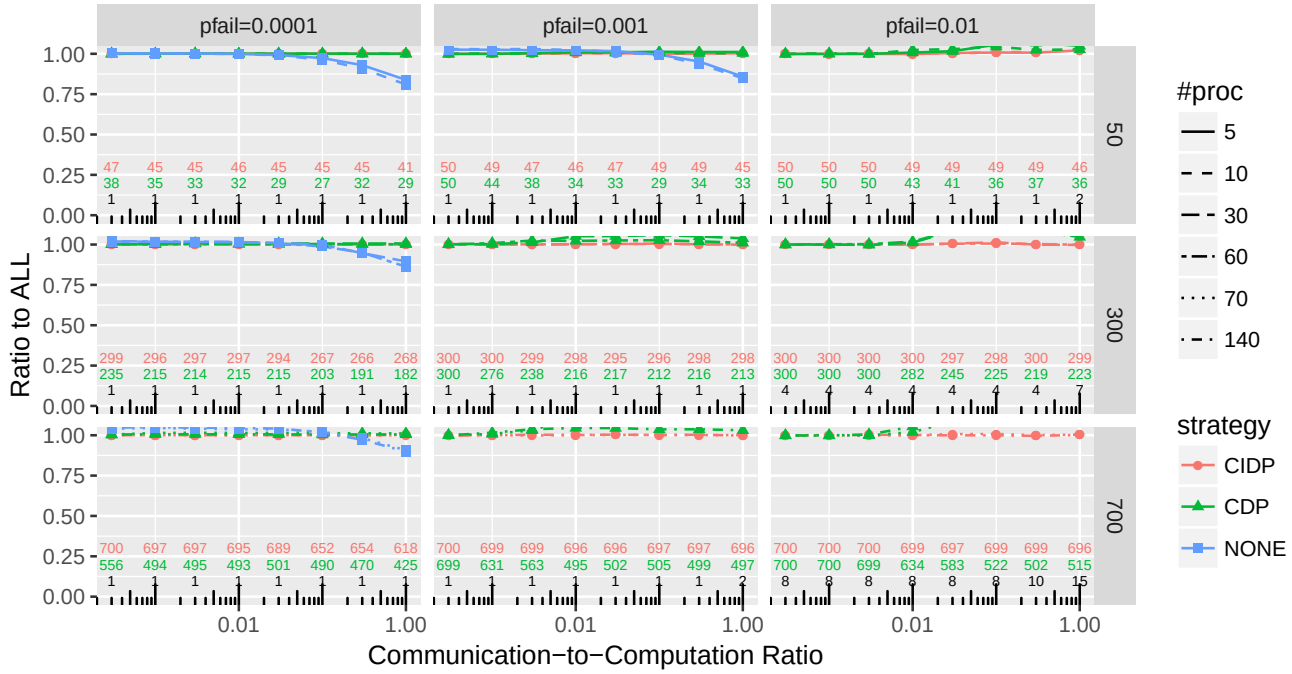

Figure 18. Performance of the different checkpointing strategies for CyberShake using HEFTC for task mapping and scheduling.

workflows to execute concurrently on multiple failure-prone processors in standard task-parallel fashion.

7 Conclusion

This work tackles the challenging problem of executing arbitrary workflows on homogeneous processors, with reasonable performance in presence of failures but without incurring a prohibitive cost when no failure strikes. While CkptAll meets the first objective by expensively checkpointing every task and CkptNone meets the second one by avoiding any checkpoint at all, we propose new strategies that provide different trade-offs between these two extremes. First, all crossover dependences, corresponding to file transfers between processors, are checkpointed, which prevents re-execution propagation between processors in case of failure. Then, a DP (Dynamic Programming) solution is used to insert additional checkpoints to minimize the expected completion time. Additional (induced) checkpoints may be added prior to the DP execution to provide it with more accurate information. Moreover, different mapping strategies that extend classical ones to reduce the number of checkpoints were also proposed. To the best of our

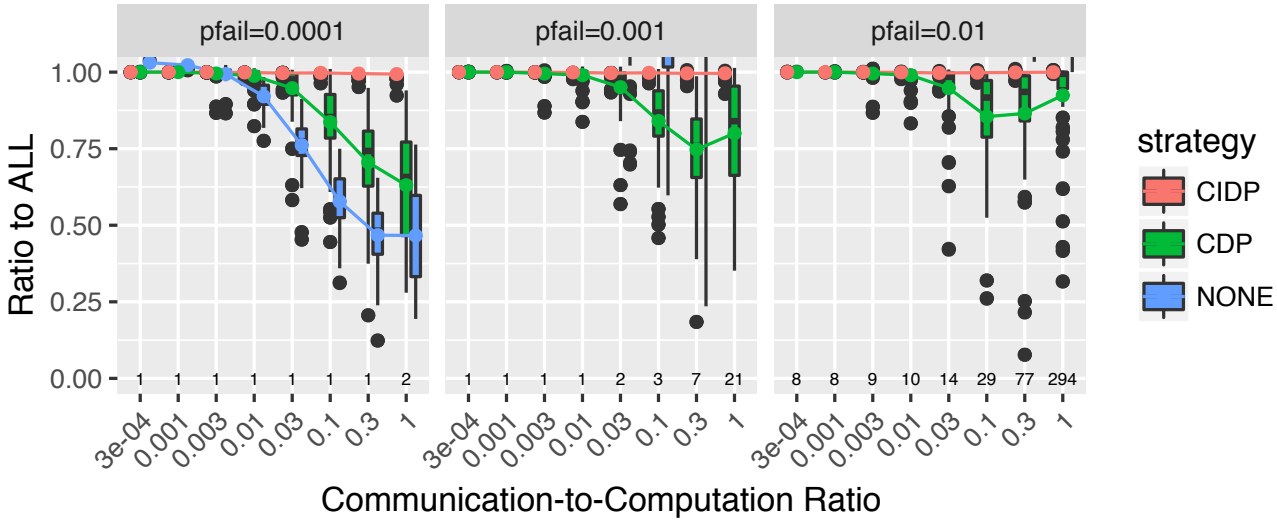

Figure 19. Average performance of the different checkpointing strategies for the STG task graphs using HEFTC for task mapping and scheduling.

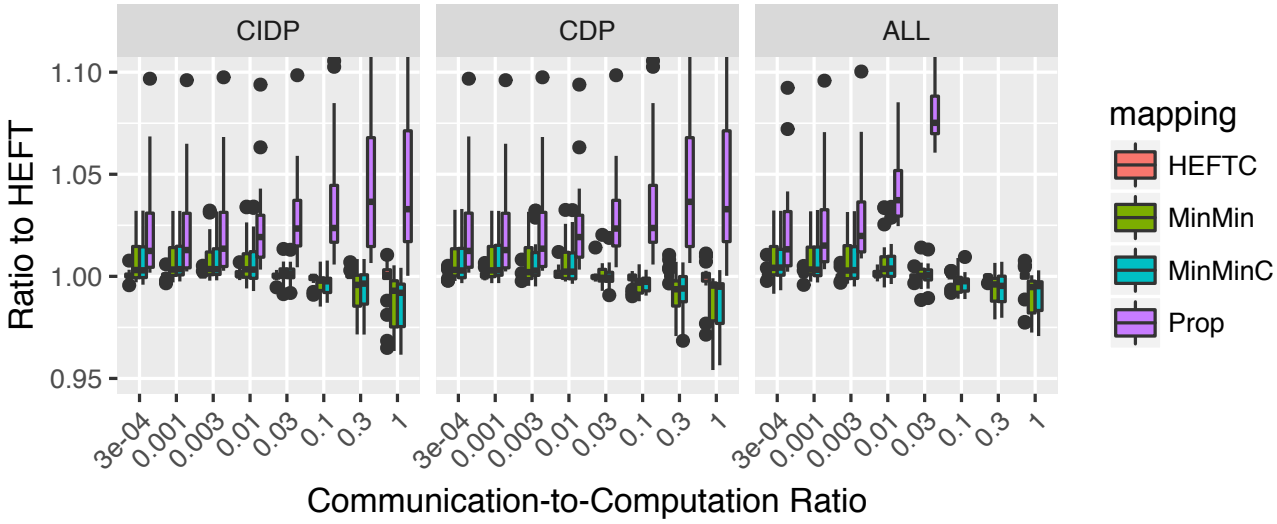

Figure 20. Relative performance of the four task mapping and scheduling strategies and of PROPCKPT for Montage.

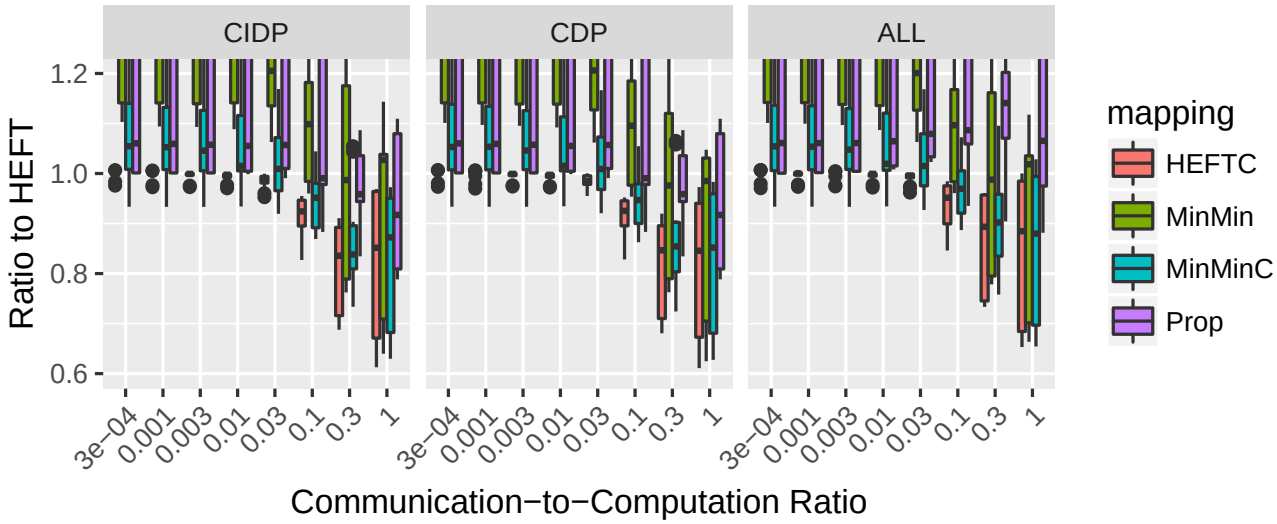

Figure 21. Relative performance of the four task mapping and scheduling strategies and of PROPCKPT for Ligo.

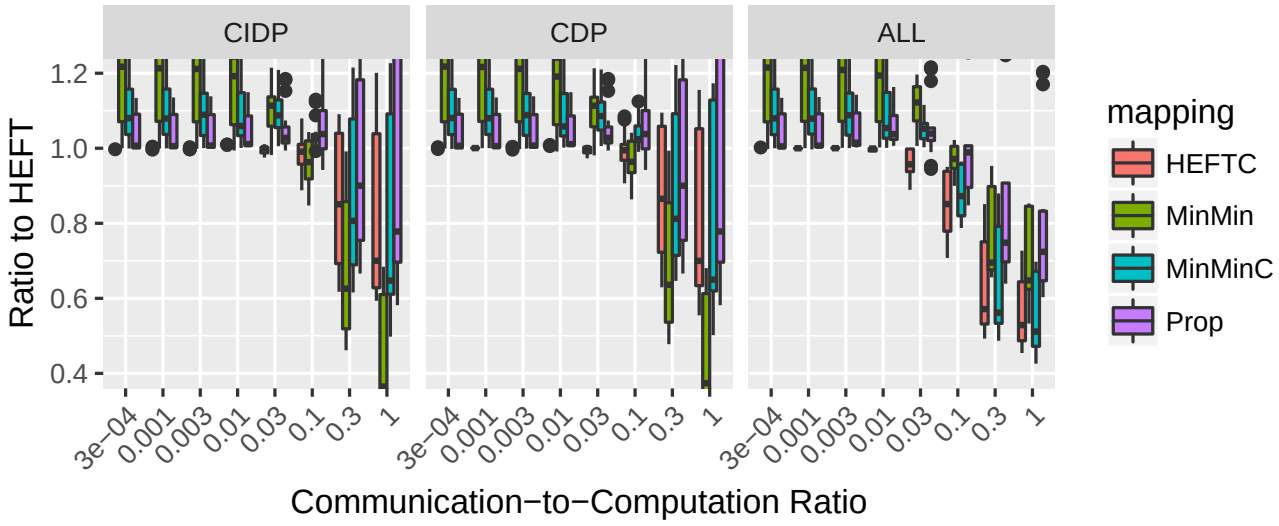

Figure 22. Relative performance of the four task mapping and scheduling strategies and of PROPCKPT for Genome.

knowledge, these new strategies are the first to be tuned to minimize the need for checkpointing while mapping tasks. Extensive experiments with a discrete event simulator, conducted for both synthetic and realistic instances, show that our approaches significantly outperform CKPTALL and CKPTNONE in most scenarios.

Future work will aim at extending our approach to workflows with parallel moldable tasks (Drozdowski (2009)). Such an extension raises yet another significant challenge: now the number of processors assigned to each task becomes a parameter to the proposed solutions, with a dramatic impact on both performance and resilience.

Acknowledgements

Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments and suggestions, which helped improve the quality of this paper. A shorter version of this work has been published in the proceedings of ICPP'18 (Han et al. (2018b)). This work is partially supported by a JORISS grant from ENS Lyon and ECNU.

Notes

∗. A preliminary version of this work appeared in the proceedings of ACM ICPP 2018.

†. In fact, because we have homogeneous processors, we use MCP (Modified Critical Path) (Wu and Gajski (1990)) with backfilling, which is exactly HEFT in this context.

‡. Each boxplot consists of a bold line for the median, a box for the quartiles, whiskers that extend at most to 1.5 times the interquartile range from the box and additional points for outliers.

References

Albrecht M, Donnelly P, Bui P and Thain D (2012) Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids. In: *1st ACM SWEET SIGMOD*. ACM.

Altintas I, Berkley C, Jaeger E, Jones M, Ludascher B and Mock S (2004) Kepler: an extensible system for design and execution of scientific workflows. In: *Proc. of 16th SSDBM*. IEEE, pp. 423–424.

Assayad I, Girault A and Kalla H (2004) A bi-criteria scheduling heuristic for distributed embedded systems under reliability and real-time constraints. In: *Dependable Systems Networks (DSN)*. IEEE.

Augonnet C, Thibault S, Namyst R and Wacrenier PA (2011) Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concur. and Comp.: Pract. and Exp.* 23(2): 187–198.

Aupy G, Benoit A, Casanova H and Robert Y (2016) Scheduling computational workflows on failure-prone platforms. *Int. J. of Networking and Computing* 6(1): 2–26.

Baldoni R, Helary J, Mostefaoui A and Raynal M (1997) A communication-induced checkpointing protocol that ensures rollback-dependency trackability. In: *Proc. IEEE 27th International Symposium on Fault Tolerant Computing*. IEEE, pp. 68–77.

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

Bautista Gomez L and Cappello F (2015) Detecting and correcting data corruption in stencil applications through multivariate interpolation. In: *FTS*. IEEE.

Benoit A, Cavelan A, Robert Y and Sun H (2016) Assessing general-purpose algorithms to cope with fail-stop and silent errors. *ACM Trans. Parallel Computing* 3(2).

Berrocal E, Bautista-Gomez L, Di S, Lan Z and Cappello F (2015) Lightweight silent data corruption detection based on runtime data analysis for HPC applications. In: *HPDC*. ACM.

Bharathi S, Chervenak A, Deelman E, Mehta G, Su MH and Vahi K (2008) Characterization of scientific workflows. In: *Workflows in Support of Large-Scale Science (WORKS)*. IEEE, pp. 1–10.

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

Braun TD, Siegel HJ, Beck N, Bölöni LL, Maheswaran M, Reuther AI, Robertson JP, Theys MD, Yao B, Hensgen D et al. (2001) A comparison of eleven static heuristics for mapping a class

of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed computing* 61(6): 810–837.

Cao C, Herault T, Bosilca G and Dongarra J (2015) Design for a soft error resilient dynamic task-based runtime. In: *IPDPS*. IEEE, pp. 765–774.

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

Choi J, Dongarra JJ, Ostrouchov LS, Petitet AP, Walker DW and Whaley RC (1996) Design and implementation of the scalapack lu, qr, and cholesky factorization routines. *Scientific Programming* 5(3): 173–184.

da Silva RF, Chen W, Juve G, Vahi K and Deelman E (2014) Community resources for enabling research in distributed scientific workflows. In: *e-Science (e-Science), 2014 IEEE 10th International Conference on*, volume 1. IEEE, pp. 177–184.

Darte A, Robert Y and Vivien F (2000) *Scheduling and automatic parallelization*. Birkhäuser. ISBN 978-3-7643-4149-7.

Deelman E, Singh G, Su MH, Blythe J, Gil Y, Kesselman C, Mehta G, Vahi K, Berriman GB, Good J et al. (2005) Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming* 13(3): 219–237.

Deelman E, Vahi K, Juve G, Rynge M, Callaghan S, Maechling PJ, Mayani R, Chen W, Ferreira da Silva R, Livny M and Wenger K (2015) Pegasus, a workflow management system for science automation. *Future Generation Computer Systems* 46: 17–35.

Downey AB (2001) The structural cause of file size distributions. In: *MASCOTS 2001*. IEEE, pp. 361–370.

Drozdowski M (2009) *Scheduling for Parallel Processing*. Computer Communications and Networks. Springer.

Duan R, Prodan R and Fahringer T (2005) Dee: A distributed fault tolerant workflow enactment engine for grid computing. In: *International Conference on High Performance Computing and Communications*. Springer, pp. 704–716.

Fahringer T, Prodan R, Duan R, Hofer J, Nadeem F, Nerieri F, Podlipnig S, Qin J, Siddiqui M, Truong HL et al. (2007) Askalon: A development and grid computing environment for scientific workflows. In: *Workflows for e-Science*. Springer, pp. 450–471.

Han L, Canon LC, Casanova H, Robert Y and Vivien F (2018a) Checkpointing workflows for fail-stop errors. *IEEE Transactions on Computers* .

Han L, Fèvre VL, Canon LC, Robert Y and Vivien F (2018b) A generic approach to scheduling and checkpointing workflows. In: *ICPP'2018, the 47th Int. Conf. on Parallel Processing*. IEEE Computer Society Press.

Huang KH and Abraham JA (1984) Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.* 33(6): 518–528.

Hwang S and Kesselman C (2003) Grid workflow: a flexible failure handling framework for the grid. In: *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on*. IEEE, pp. 126–137.

Hérault T and Robert Y (eds.) (2015) *Fault-Tolerance Techniques for High-Performance Computing*, Computer Communications and Networks. Springer Verlag.

Jin H, Sun XH, Zheng Z, Lan Z and Xie B (2009) Performance Under Failures of DAG-based Parallel Computing. In: *CCGRID '09*. IEEE Computer Society.

Juve G, Chervenak A, Deelman E, Bharathi S, Mehta G and Vahi K (2013) Characterizing and profiling scientific workflows. *Future Generation Computer Systems* 29(3): 682–692.

Kail E, fchtpen P and Kozlovszky M (2016) A novel adaptive checkpointing method based on information obtained from workflow structure. *Computer Science* 17(3).

Lin CY, Chen TH and Cheng YN (2013) On improving fault tolerance for heterogeneous hadoop mapreduce clusters. In: *2013 International Conference on Cloud Computing and Big Data*. IEEE, pp. 38–43.

Pegasus (2014) Pegasus workflow generator. `https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator`.

Pothen A and Sun C (1993) A mapping algorithm for parallel sparse cholesky factorization. *SIAM J. on Scientific Computing* 14(5): 1253–1257.

Shantharam M, Srinivasmurthy S and Raghavan P (2012) Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In: *ICS*. ACM.

Tobita T and Kasahara H (2002) A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. *Journal of Scheduling* 5(5): 379–394.

Tolosana-Calasanz R, Bañares JÁ, Álvarez P, Ezpeleta J and Rana O (2010) An uncoordinated asynchronous checkpointing model for hierarchical scientific workflows. *Journal of Computer and System Sciences* 76(6): 403–415.

Topcuoglu H, Hariri S and Wu My (2002) Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems* 13(3): 260–274.

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

Valdes J, Tarjan RE and Lawler EL (1979) The recognition of series parallel digraphs. In: *Proc. of STOC'79*. ACM, pp. 1–12.

Wang P, Zhang K, Chen R, Chen H and Guan H (2014) Replication-based fault-tolerance for large-scale graph processing. In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. pp. 562–573.

Wilde M, Hategan M, Wozniak JM, Clifford B, Katz DS and Foster I (2011) Swift: A language for distributed parallel scripting. *Parallel Computing* 37(9): 633–652.

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

Wu MY and Gajski DD (1990) Hypertool: a programming aid for message-passing systems. *IEEE Trans. Parallel Distributed Systems* 1(3): 330–343.

Zhang F, Docan C, Parashar M, Klasky S, Podhorszki N and Abbasi H (2012) Enabling In-situ Execution of Coupled Scientific Workflow on Multi-core Platform. In: *Proc. 26th IEEE IPDPS*. pp. 1352–1363.

Zhu W, Chen H and Hu F (2016) Asc: Improving spark driver performance with automatic spark checkpoint. In: *2016 18th International Conference on Advanced Communication Technology (ICACT)*. IEEE, pp. 607–611.