On the complexity of scheduling checkpoints for computational workflows

Yves Robert[1,2], Frédéric Vivien[1] and Dounia Zaidouni[1]
1. Ecole Normale Supérieure de Lyon & INRIA, France
{Yves.Robert|Frederic.Vivien|Dounia.Zaidouni}@ens-lyon.fr
2. University of Tennessee Knoxville, USA

***Abstract*—This paper deals with the complexity of scheduling computational workflows in the presence of Exponentially distributed failures. When such a failure occurs, rollback and recovery is used so that the execution can resume from the last checkpointed state. The goal is to minimize the expected execution time, and we have to decide in which order to execute the tasks, and whether to checkpoint or not after the completion of each given task. We show that this scheduling problem is strongly NP-complete, and propose a (polynomial-time) dynamic programming algorithm for the case where the application graph is a linear chain. These results lay the theoretical foundations of the problem, and constitute a prerequisite before discussing scheduling strategies for arbitrary DAGS of moldable tasks subject to general failure distributions.**

I. Introduction

In this paper, we provide preliminary results on the complexity of scheduling workflows in the presence of Exponentially distributed failures. More precisely, there is an application graph to be executed, whose tasks are executed sequentially on some (parallel) platform. When a failure occurs, rollback and recovery is used so that the execution can resume from the last checkpointed state. In a nutshell, the goal is to decide (i) in which order to execute the tasks (always enforcing all dependences); and (ii) whether to checkpoint or not after the completion of each given task. The objective is to minimize the expectation of the total execution time. We show that this scheduling problem is quite difficult: we establish the strong NP-completeness of the instance with independent tasks (Section IV). We also propose a (polynomial-time) dynamic programming algorithm for the case where the application graph is a linear chain (Section V). Both results rely on an exact formula for the expected time needed to successfully execute a task and checkpoint right after it (Section III). To the best of our knowledge, this formula, together with its recursive proof, is original. Before detailing these results, we outline the framework in full details in Section II. In the last sections of the paper, we discuss some possible extensions of this work (Section VI), and we briefly survey related work (Section VII). Finally, we provide concluding remarks in Section VIII.

II. Framework

We are given an application task graph $G = (V, E)$, i.e., a Directed Acylic Graph (DAG) where nodes represent tasks and edges correspond to dependences between them. We let $V = \{T_1, T_2, \ldots, T_n\}$, and each task T_i is weighted by its computational weight w_i. The application DAG is executed on a platform of p identical processors. We use the term processor to indicate any individually scheduled compute resource (a core, a multi-core processor, a cluster node) so that our work is agnostic to the granularity of the platform. These processors are subject to failures, and we assume that a standard coordinated checkpointing and roll-back recovery is performed at the system level. At the end of the execution of each task T_i, we can decide either to perform a checkpoint, or to proceed with the computation of another task. The former option (checkpoint) induces overhead to the total execution time, but allows to recover from the current state if a failure happens further on, thereby reducing the time wasted due to that failure. The latter option (no checkpoint) saves time in a failure-free execution but is more risky, since a subsequent failure would cause a longer rollback from an older execution state. The objective is to minimize the expected total execution time (also called *makespan*).

The overhead due to checkpointing is modeled as follows: a checkpoint taken after executing task T_i requires an arbitrary time C_i. For recoveries, we assume that if the most recent available checkpoint was taken after task T_i, then the recovery takes an arbitrary time R_i. Finally, we add an additional overhead D for downtime. The downtime accounts for software rejuvenation (i.e., rebooting [1]) or for the replacement of the failed processor by a spare. Note that failures can take place during recovery, but not during downtime (otherwise simply combine downtime with recovery). For the sake of simplicity, we make two important restrictive assumptions:

Full parallelism: Each task is executed by all the p processors;

Poisson process: Processor failure inter-arrival times follow an Exponential distribution of parameter λ_{proc} (hence platform failure inter-arrival times follow an Exponential distribution of parameter $\lambda = p\lambda_{\text{proc}}$).

Section VI discusses several extensions to this model, and hints to alleviate these limitations. The first assumption (*full parallelism*) amounts to linearize the task graph and to execute all tasks sequentially. This is the only possibility when the original DAG is a linear chain, a situation very frequent in scientific applications [2], [3], [4], but it may prove a severe restriction when several tasks could be executed concurrently. However, *full parallelism* allows to avoid any resource allocation problem. Instead, we search for the best ordering of the tasks so as to minimize the expectation of the total execution time. As shown in Section IV, this simpler problem instance already is NP-hard in the strong sense, even with constant checkpoint costs. This negative result shows the intrinsic combinatorial difficulty to deciding which tasks to execute first, and which ones to checkpoint. On a more constructive basis, Section V presents a polynomial solution when the DAG is a linear chain, with arbitrary checkpoint costs. These important results provide theoretical foundations for the problem, and constitute a prerequisite before tackling even more challenging instances.

The second assumption (*Poisson process*) is standard in theoretical analyses [5], [6] because the memoryless property of Exponential distributions allows for deriving scheduling strategies that do not depend upon the history of previous failures. However, we do acknowledge that Weibull and/or log-normal distributions are considered more relevant in practice [7], [8], [9], [10].

III. Expectation of the time needed to execute a work and to checkpoint it

In this section, we provide an exact formula to compute the expected time $\mathbb{E}(T(W,C,D,R,\lambda))$ to execute a work of duration W followed by a checkpoint of duration C. If a failure interrupts a given attempt, there is a downtime of duration D followed by a recovery of length R. Recall that failures follow an Exponential distribution (of parameter λ) and can take place during recovery, but not during downtime. To the best of our knowledge, this important result is original. Daly only provides first and second order approximations [6], and the formula in Bouguerra et al. [11] is inaccurate (in their approach, a recovery always takes place before execution, which is false for the first attempt). In addition, the proof is original too: we use an elegant recursive approach, while in the literature, the standard approach is to consider consecutive execution attempts iteratively, until success.

Proposition 1.

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

Proof: Let T be the time needed for successfully executing a work of duration W. There are two cases: (i) if there is no failure during execution and checkpointing, then the time needed is exactly $W+C$; (ii) if there is one failure before successfully completing the work and its checkpoint, then some additional delays are incurred. These delays come from two sources: the time spent computing by the processors before the failure (accounted for by variable T_{lost}), and the time spent for downtime and recovery (accounted for by variable T_{rec}). Regardless, once a successful recovery has been completed, there still remain W units of work to execute. Thus we can write the following recursion:

$$T = \begin{cases} W+C & \text{if no failure happens} \\ T_{lost}+T_{rec}+T & \text{otherwise} \end{cases} \quad (1)$$

Here, T_{lost} denotes the amount of time spent by the processors before the first failure, knowing that this failure occurs within the next $W+C$ units of time. In other terms, it is the time that is wasted because computation and checkpoint were not both completed. The other variable T_{rec} represents the amount of time needed by the system to recover from the failure. Weighting the two cases in Equation (1) by the probability of their occurrence, and taking expectations, we obtain the following equation:

$$\begin{aligned}\mathbb{E}(T) &= e^{-\lambda(W+C)}(W+C) \\ &\quad + (1-e^{-\lambda(W+C)})\left[E(T_{lost})+E(T_{rec})+E(T)\right] \quad (2)\end{aligned}$$

This simplifies to:

$$\mathbb{E}(T) = W+C+(e^{\lambda(W+C)}-1)(E(T_{lost})+E(T_{rec})) \quad (3)$$

We have

$$\begin{aligned}\mathbb{E}(T_{lost}) &= \int_0^\infty x\mathbb{P}(X=x|X<W+C)dx \\ &= \frac{1}{\mathbb{P}(X<W+C)}\int_0^{W+C} e^{-\lambda x}dx,\end{aligned}$$

and $\mathbb{P}(X<W+C) = 1-e^{-\lambda(W+C)}$. Integrating by parts, we derive that

$$\mathbb{E}(T_{lost}) = \frac{1}{\lambda} - \frac{W+C}{e^{\lambda(W+C)}-1} \quad (4)$$

Next, to compute $\mathbb{E}(T_{rec})$, we have a recursive equation quite similar to Equation (2):

$$\mathbb{E}(T_{rec}) = e^{-\lambda R}(D+R)+(1-e^{-\lambda R})(D+R_{lost}+\mathbb{E}(T_{rec}))$$

Here, R_{lost} is the expected amount of time lost to executing the recovery before a failure happens, knowing that this failure occurs within the next R units of time. Replacing $W+C$ by R in Equation (4), we obtain

$$R_{lost} = \frac{1}{\lambda} - \frac{R}{e^{\lambda R}-1}.$$

The expression for $\mathbb{E}(T_{rec})$ simplifies to

$$\mathbb{E}(T_{rec}) = De^{\lambda R}+\frac{1}{\lambda}(e^{\lambda R}-1) \quad (5)$$

Plugging the values of $\mathbb{E}(T_{lost})$ and $\mathbb{E}(T_{rec})$ into Equation (3) leads to the desired value:

$$\mathbb{E}(T(W,C,D,R,\lambda)) = e^{\lambda R}\left(\frac{1}{\lambda} + D\right)(e^{\lambda(W+C)} - 1) \quad (6)$$

■

Equation (6) is fully general, in the sense that the values of W, C, D, and R may arbitrarily depend upon the number p of processors in the platform (recall that $\lambda = p\lambda_{\text{proc}}$ linearly depends on p for Exponential laws). Let us write $W(p)$, $C(p)$, $R(p)$ and $D(p)$ to make the dependence on p explicit. In [12] we have identified several relevant scenarios:

Workload model. For a total sequential load W_{total}, we can have:
(i) $W(p) = W_{\text{total}}/p$: perfectly parallel jobs;
(ii) $W(p) = (1-\gamma)W_{\text{total}}/p + \gamma W_{\text{total}}$: generic parallel jobs where, as in Amdahl's law [13], $\gamma < 1$ is the fraction of the work that is inherently sequential;
(iii) $W(p) = W_{\text{total}}/p + \gamma W_{\text{total}}^{2/3}/\sqrt{p}$: numerical kernels, such as a matrix product or a LU/QR factorization of size N on a 2D-processor grid, where $W_{\text{total}} = O(N^3)$. In the algorithm in [14], $q = r^2$ and each processor receives $2r$ blocks of size N^2/r^2 during the execution. Here γ is the communication-to-computation ratio of the platform.

Checkpoint overhead. Assuming that the application's memory footprint is V bytes, with each processor holding V/p bytes, we can have:
(i) $C(p) = R(p) = \alpha V/p = C/p$ with α some constant: proportional overhead, for cases where the bandwidth of the network card/link at each processor is the I/O bottleneck;
(ii) $C(p) = R(p) = \alpha V = C$ with α some constant: constant overhead, for cases where the bandwidth to/from the resilient storage system is the I/O bottleneck.

Finally, there is a technical difficulty hidden in Equation (6): with a single processor ($p = 1$), the downtime has constant value D, but with several processors, the duration of the downtime is difficult to compute: a processor can fail while another one is down, thereby leading to cascading downtimes. Hence in Equation (6), D should be taken as the expectation of the variable $D(p)$, whose exact value is unknown, but for which an upper bound can be provided (see [15] for details). In most practical cases, the lower bound $D(p) = D(1) = D$ is expected to be very accurate.

IV. Complexity

In this section we prove that the general scheduling problem is NP-complete in the strong sense. This result holds true for independent tasks and the simplest checkpoint cost model where all costs are equal. Intuitively, this shows that deciding for an ordering to execute several independent tasks, and after which task completions to checkpoint, is a difficult combinatorial problem. Note that this holds true independently of the value of the number of processors p. In particular, the result holds when using a single-processor platform.

Proposition 2. *Consider n independent tasks, T_1, ..., T_n, with task T_i of duration w_i for $1 \leq i \leq n$. All checkpoint and recovery times are equal to C, and there is no downtime ($D = 0$). The problem to schedule these tasks, and to decide after which tasks to checkpoint, so as to minimize the expected execution time, is NP-complete in the strong sense.*

Proof: The decision problem is the following: given a time bound K, can we find an ordering for the execution, and decide after which tasks to checkpoint, so that the expected execution time $\mathbb{E}$ does not exceed K? The problem clearly belongs to NP, with the ordered list of tasks and checkpoints being a linear-size certificate. To establish the completeness, we use a reduction from 3-PARTITION, which is NP-complete in the strong sense [16]. Consider the following general instance $\mathcal{I}_1$ of 3-PARTITION: given $3n$ integers $a_1, \ldots, a_{3n}$ and a number T such that $\sum_{1 \leq j \leq 3n} a_j = nT$, and $\frac{T}{4} < a_i < \frac{T}{2}$ for $1 \leq i \leq 3n$, does there exist a partition in n subsets $B_1, \ldots, B_n$ of $\{a_1, \ldots, a_{3n}\}$ such that for all $1 \leq i \leq n$, $\sum_{a_j \in B_i} a_j = T$? Note that necessarily in any solution, each B_i has cardinal 3.

We build the following instance $\mathcal{I}_2$ of our problem with $3n$ independent tasks, T_1, ..., T_{3n}, task T_i being of size $w_i = a_i$. We let

$$\lambda = \frac{1}{2T}, \quad C = R = \frac{1}{\lambda}(\ln(2) - \frac{1}{2}),$$

$$D = 0, \quad K = n\frac{e^{\lambda C}}{\lambda}(e^{\lambda(T+C)} - 1).$$

The size of $\mathcal{I}_2$ is polynomial (even linear) in the size of $\mathcal{I}_1$. We show that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ has a solution.

Suppose first that $\mathcal{I}_1$ has a solution $B_1, \ldots, B_n$. We propose the following solution for $\mathcal{I}_2$: we execute the subsets in any order; for each subset B_i, we schedule its three tasks in any order, and we checkpoint after the third one. From Proposition 1, the expected execution time for each subset is $\frac{e^{\lambda C}}{\lambda}(e^{\lambda(T+C)} - 1)$. The expected total execution time is $\mathbb{E} = n\frac{e^{\lambda C}}{\lambda}(e^{\lambda(T+C)} - 1) = K$, hence a solution to $\mathcal{I}_2$.

Suppose now that $\mathcal{I}_2$ has a solution, which includes m checkpoints. Let B_1 be the set of tasks executed before the first checkpoint, and for $2 \leq i \leq m$, let B_i be the set of tasks executed between checkpoints $i-1$ and i. For $1 \leq i \leq m$, let T_i be the total duration of the tasks in B_i. We have $\sum_{i=1}^{m} T_i = nT$ (total work to be executed). From Proposition 1, the expected total execution time is $\mathbb{E} = \sum_{i=1}^{m} \frac{e^{\lambda C}}{\lambda}(e^{\lambda(T_i+C)} - 1)$, and by hypothesis we have

$\mathbb{E} \leq K$. We write $\mathbb{E}$ as

$$\mathbb{E} = \left(\frac{e^{2\lambda C}}{\lambda} \sum_{i=1}^{m} e^{\lambda T_i} \right) - m\frac{e^{\lambda C}}{\lambda}.$$

Consider $\sum_{i=1}^{m} e^{\lambda T_i}$: by convexity, since $\sum_{i=1}^{m} T_i = nT$ is constant, this sum is minimal when all terms T_i are equal, and their common value must be $T_i = \frac{nT}{m}$. Hence

$$\mathbb{E} \geq \mathbb{E}_0 = m\frac{e^{\lambda C}}{\lambda}(e^{\lambda(\frac{nT}{m}+C)} - 1),$$

with equality if and only if all the T_i's are equal. We write $\mathbb{E}_0 = \frac{e^{\lambda C}}{\lambda} g(m)$, where

$$g(m) = m(e^{\lambda(\frac{nT}{m}+C)} - 1),$$

and we differentiate g:

$$g'(m) = \left(1 - \frac{\lambda n T}{m}\right) e^{\lambda(\frac{nT}{m}+C)} - 1$$

$$g''(m) = \frac{\lambda^2 n^2 T^2}{m^3} e^{\lambda(\frac{nT}{m}+C)} > 0$$

Hence g is convex and its first derivative its a strictly increasing function. Then g has a unique minimum which is achieved for $m = n$, since $g'(n) = 0$. Indeed, we have $g'(n) = (1 - \lambda T)e^{\lambda(T+C)} - 1$. By hypothesis, we have $\lambda = \frac{1}{2T}$ and $C = \frac{1}{\lambda}(\ln(2) - \frac{1}{2})$. Therefore,

$$e^{\lambda(T+C)} = e^{\frac{1}{2}+(\ln(2)-\frac{1}{2})} = e^{\ln(2)} = 2,$$

and finally $g'(n) = 0$.

Altogether, the minimum value of $\mathbb{E}$ is uniquely reached for $m = n$ and $T_i = T$ for all i, in which case $\mathbb{E} = K$. This shows that $B_1, \ldots, B_n$ is a solution for $\mathcal{I}_1$, which concludes the proof. ■

V. Linear chains

In this section, we present a polynomial-time dynamic programming algorithm to compute the optimal execution time for applications whose DAG is a linear chain $T_1 \rightarrow T_2 \cdots \rightarrow T_n$. Recall that the execution time of task T_i is w_i for $1 \leq i \leq n$. We allow for different checkpoint times and let C_i denote the duration of a checkpoint after task T_i for $1 \leq i \leq n$. Similarly, we let R_i be the recovery time if the most recent available checkpoint was taken after task T_i, for $1 \leq i \leq n-1$ (no need to recover from after T_n). As before, we have a downtime of duration D after each failure, and failures follow an Exponential law of parameter λ.

Proposition 3. *Algorithm 1 provides the optimal solution for a linear chain of n tasks. Its complexity is $O(n^2)$.*

Proof: In Algorithm 1, DPMAKESPAN(x,n) computes the optimal expectation of the time needed for successfully executing the last (n-x+1) tasks, with $1 \leq x \leq n$. Our goal is to compute DPMAKESPAN(1,n). Note that DPMAKESPAN(x,n) returns a couple formed by the optimal expectation of the execution time, and the index of the task that precedes the checkpoint in the outermost recursion level (needed to reconstruct the solution).

Algorithm 1 contains a loop of size (n-x) and inside the loop, we perform a recursive call of the function DPMAKESPAN. This recursive call has linear complexity because we only compute each instance DPMAKESPAN(x,n,) once, using memoization [17] (i.e., storing the result of the recursive calls that are have already been computed). Thus, the complexity of Algorithm 1 is $O(n^2)$. ■

VI. Extensions

In this section we discuss three extensions of this work. In particular, we address the two limitations outlined in Section II, namely *full parallelism* and *Poisson process*.

The first extension relates to checkpointing costs. In Section II, we have considered that the time required to checkpoint only depends on the last task executed prior to that checkpoint. In a more general model, the time needed for a checkpoint after task T_i may depend on T_i itself, but also on some other tasks that have been executed since the last checkpoint. For instance, assume that two independent tasks T_i and then T_j are executed just after a checkpoint, and that there is a checkpoint after these tasks. The model from Section II states that the cost of this checkpoint is C_j, regardless whether there has been a checkpoint between T_i and T_j or not. In a more general and realistic model, the checkpoint cost would be $f(C_i, C_j)$, where f is an application-dependent function. In the general case, the cost of a checkpoint should account for all the tasks that have been executed since the last checkpoint and which have at least a successor task which has not been executed yet. Note that in the case of linear chains (Section V), there is always a single task that needs to be saved, so that the cost model that we used is fully general.

The second extension aims at alleviating the *full parallelism* assumption. In a model with variable parallelism, we have *moldable* tasks [18], that can be execute with an arbitrary number of processors. To compute the expected time of a task using any given number of processors, we can use the different workload models described at the end of Section III and then instantiate Equation (6). But we now face a challenging resource allocation problem: deciding how many resources to assign to each task. This problem is known to be difficult, but approximation algorithms are available for failure-free environments [18]. It would be very interesting to extend these algorithms to failure-prone platforms.

The third extension would allow for more general failure laws than Exponential distributions. Assuming that processor failures would follow Weibull and/or log-normal distributions [7], [8], [9], [10]. there are two main difficulties. The first difficulty is to compute, or better approximate, the failure distribution of a platform with p processors, which is the *superposition* of p independent and identically

Algorithm 1: DPMAKESPAN(x, n)

```
if $x = n$ then
    return $\left(e^{\lambda R_{n-1}}(\frac{1}{\lambda} + D)(e^{\lambda(w_n + C_n)} - 1), n\right)$
$best \leftarrow e^{\lambda R_{x-1}} \left(\frac{1}{\lambda} + D\right) \left(e^{\lambda\left(\left(\sum_{i=x}^{n} w_i\right) + C_n\right)} - 1\right)$;
$numTask \leftarrow n$;
for $j = x$ to $n - 1$ do
    $(exp\_succ, num\_Task) \leftarrow DPMAKESPAN(j + 1, n;)$
    $Cur \leftarrow exp\_succ$;
            $+ e^{\lambda R_{x-1}} \left(\frac{1}{\lambda} + D\right) \left(e^{\lambda\left(\left(\sum_{i=x}^{j} w_i\right) + C_j\right)} - 1\right)$;
    if $Cur < best$ then
        $best \leftarrow Cur$;
        $numTask \leftarrow j$
return $(best, numTask)$
```

distributed distributions (with a single processor). The second difficulty is to estimate the expected execution time of a work of duration W followed by a checkpoint of duration C. No closed-form formula is known. This is because we have to account for time elapsed since the last failure on each processor, since the failure distribution no longer is memoryless. We would have to use heuristic approaches even for linear chains. For instance, instead of aiming at the minimization of the expected execution time, it is possible to (greedily) aim at the maximization of the expected amount of work achieved before the next failure [12], [19]. Dynamic programming heuristics and simulation results are provided in [12] for single-task applications, using either synthetic traces or failure logs of production clusters [20].

VII. Related work

There is a large body of literature on checkpointing strategies for divisible jobs. The corresponding scheduling problem is to partition the job into several chunks and to checkpoint after each of them. In [6], Daly studies periodic checkpointing policies (same-size chunks) for Exponentially distributed failures, generalizing the well-known bound obtained by Young [21]. Daly extended his work in [22] to study the impact of sub-optimal checkpointing periods. In [23], the authors develop an "optimal" checkpointing policy, based on the popular assumption that optimal checkpointing must be periodic. In [11], Bouguerra et al. *prove* that the optimal checkpointing policy is periodic when checkpointing and recovery overheads are constant, for either Exponential or Weibull failures. But their results rely on the unstated assumption that all processors are rejuvenated after each failure and after each checkpoint. In our recent work [12], we have shown that this assumption is unreasonable for Weibull failures, and we have developed optimal solutions for Exponential failures and dynamic programming solutions for any failure distribution. As already mentioned, solving the problem for arbitrary distributions is difficult because, unlike in the memoryless case, there is no reason for the optimal solution to use a single chunk size [24]. In fact, the optimal solution is very likely to use chunk sizes that depend on additional information that becomes available during the execution (i.e., failure occurrences to date).

The problem studied in this paper is related to the previous line of work, but strongly differs in that checkpoints are allowed only after a task has been completed. In other words, instead of studying a divisible job that can be arbitrarily partitioned into chunks, we study a DAG of non-divisible computational tasks. To the best of our knowledge, there are few papers studying checkpointing strategies for computational workflows. Our work is motivated by the results of Bouguerra, Trystram, and Wagner [19], who study the problem instance with linear chains (as in Section V), with a single processor. Since they deal with arbitrary distributions, they cannot aim at minimizing the expected execution time. Instead, they aim at maximizing the amount of work done before the first failure, which is a natural greedy heuristic to minimize the total execution time. They show that their problem is NP-complete in the weak sense for uniform distributions, and they propose a pseudo-polynomial dynamic programming algorithm. Our results nicely complement those of [19], since we solve the original problem for Exponential distributions, while they provide uni-processor heuristics for general distributions.

Other scheduling approaches for the reliability of workflow applications can be classified along two main threads: (i) resource allocation strategies that trade-off between makespan and reliability (bi-criteria optimization problem): and (ii) replication-oriented heuristics. A small list of representative papers is [25], [26], [27]. Very recently, there have been studies [28], [29] on the use of replication as a mechanism complementary to checkpoint-recovery.

VIII. Conclusion

In this work, we have presented preliminary results on the complexity of checkpointing computational workflows. We have used a simplified framework with two main limitations, full parallelism (rigid tasks) and Exponential failure distributions. We have derived several new and important results: (i) a closed-form formula for the expected execution time of a computational workload followed by its checkpoint; (ii) the strong NP-hardness of the problem for independent tasks and constant checkpoint costs; and (iii) a dynamic programming algorithm for linear chains of tasks with arbitrary checkpoint costs. We believe that these results lay the theoretical foundations of the problem and constitute a prerequisite before discussing scheduling strategies for arbitrary DAGS of moldable tasks subject to general failure distributions. We have discussed several directions to tackle such a challenging problem.

Acknowledgments. The authors are with Université de Lyon, France. Y. Robert is with the Institut Universitaire de France. This work was supported in part by the ANR *RESCUE* project.

References

[1] N. Kolettis and N. D. Fulton, "Software rejuvenation: Analysis, module and applications," in *FTCS '95*. Washington, DC, USA: IEEE CS, 1995, p. 381.

[2] "DataCutter Project: Middleware for Filtering Large Archival Scientific Datasets in a Grid Environment," http://www.cs.umd.edu/projects/hpsl/ResearchAreas/DataCutter.htm.

[3] K. Taura and A. A. Chien, "A heuristic algorithm for mapping communicating tasks on heterogeneous resources," in *Heterogeneous Computing Workshop*. IEEE Computer Society Press, 2000, pp. 102–115.

[4] Q. Wu and Y. Gu, "Supporting distributed application workflows in heterogeneous computing environments," in *14th Int. Conf. on Parallel and Distributed Systems (ICPADS)*. IEEE Computer Society Press, 2008.

[5] A. Duda, "The effects of checkpointing on program execution time," *Inf. Processing Letters*, vol. 16, no. 5, pp. 221–229, 1983.

[6] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 303–312, 2004.

[7] T. Heath, R. P. Martin, and T. D. Nguyen, "Improving cluster availability using workstation validation," *SIGMETRICS Perf. Eval. Rev.*, vol. 30, no. 1, pp. 217–227, 2002.

[8] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," in *Proc. of DSN*, 2006, pp. 249–258.

[9] Y. Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Paun, and S. Scott, "An optimal checkpoint/restart model for a large scale high performance computing system," in *IPDPS 2008*. IEEE, 2008, pp. 1–9.

[10] E. Heien, D. Kondo, A. Gainaru, D. LaPine, B. Kramer, and F. Cappello, "Modeling and tolerating heterogeneous failures in large parallel systems," in *Proc. SC'2011 Int. Conf. for High Performance Computing, Networking, Storage and Analysis*. ACM Press, 2011.

[11] M.-S. Bouguerra, T. Gautier, D. Trystram, and J.-M. Vincent, "A flexible checkpoint/restart model in distributed systems," in *PPAM*, ser. LNCS, vol. 6067, 2010, pp. 206–215. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14390-8_22

[12] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien, "Checkpointing strategies for parallel jobs," in *Proc. SC'2011 Int. Conf. for High Performance Computing, Networking, Storage and Analysis*. ACM Press, 2011.

[13] G. Amdahl, "The validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS Conference Proceedings*, vol. 30. AFIPS Press, 1967, pp. 483–485.

[14] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*. SIAM, 1997.

[15] M. Bougeret, H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni, "Using group replication for resilience on exascale systems," INRIA, Research report RR-7876, February 2012. [Online]. Available: http://hal.inria.fr/hal-00668016

[16] M. R. Garey and D. S. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 2001.

[18] P. françois Dutot, G. Mounié, and D. Trystram, "Scheduling Parallel Tasks Approximation Algorithms," in *Handbook of Scheduling*. CRC Press, 2004, p. chapter 26.

[19] M.-S. Bouguerra, D. Trystram, and F. Wagner, "Complexity analysis of checkpoint scheduling with variable costs," *Computers, IEEE Transactions on*, 2012.

[20] D. Kondo, B. Javadi, A. Iosup, and D. Epema, "The failure trace archive: Enabling comparative analysis of failures in diverse distributed systems," *Cluster Computing and the Grid, IEEE International Symposium on*, vol. 0, pp. 398–407, 2010.

[21] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Communications of the ACM*, vol. 17, no. 9, pp. 530–531, 1974.

[22] W. Jones, J. Daly, and N. DeBardeleben, "Impact of sub-optimal checkpoint intervals on application efficiency in computational clusters," in *HPDC'10*. ACM, 2010, pp. 276–279.

[23] K. Venkatesh, "Analysis of Dependencies of Checkpoint Cost and Checkpoint Interval of Fault Tolerant MPI Applications," *Analysis*, vol. 2, no. 08, pp. 2690–2697, 2010.

[24] A. Tantawi and M. Ruschitzka, "Performance analysis of checkpointing strategies," *ACM TOCS*, vol. 2, no. 2, pp. 123–144, 1984.

[25] J. Dongarra, E. Jeannot, E. Saule, and Z. Shi, "Bi-objective scheduling algorithms for optimizing makespan and reliability on heterogeneous systems," in *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. ACM Press, 2007, pp. 280–288.

[26] A. Dogan and F. Özgüner, "Matching and scheduling algorithms for minimizing execution time and failure probability of applications in heterogeneous computing," *IEEE Trans. Parallel Distributed Systems*, vol. 13, no. 3, pp. 308–323, 2002.

[27] A. Girault, E. Saule, and D. Trystram, "Reliability versus performance for critical applications," *J. Parallel Distributed Computing*, vol. 69, no. 3, pp. 326–336, 2009.

[28] S. Yi, D. Kondo, B. Kim, G. Park, and Y. Cho, "Using Replication and Checkpointing for Reliable Task Management in Computational Grids," in *Proc. of the International Conference on High Performance Computing & Simulation*, 2010.

[29] K. Ferreira, J. Stearley, J. H. I. Laros, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, "Evaluating the Viability of Process Replication Reliability for Exascale Systems," in *Proceedings of the 2011 ACM/IEEE Conference on Supercomputing*, 2011.