Revisiting the double checkpointing algorithm

Jack Dongarra, Thomas Hérault, Yves Robert

▶ **To cite this version:**

Jack Dongarra, Thomas Hérault, Yves Robert. Revisiting the double checkpointing algorithm. APDCM 2013, IEEE, 2013, Boston, United States. hal-00925168

HAL Id: hal-00925168

https://hal.inria.fr/hal-00925168

Submitted on 7 Jan 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Revisiting the double checkpointing algorithm

Jack Dongarra[1], Thomas Herault[1] and Yves Robert[1,2]
1. University of Tennessee Knoxville, USA
2. Ecole Normale Supérieure de Lyon & INRIA, France
{dongarra|herault}@eecs.utk.edu, yves.robert@ens-lyon.fr

***Abstract*—Fast checkpointing algorithms require distributed access to stable storage. This paper revisits the approach base upon double checkpointing, and compares the blocking algorithm of Zheng, Shi and Kalé [1], with the non-blocking algorithm of Ni, Meneses and Kalé [2] in terms of both performance and risk. We also extend the model proposed in [1], [2] to assess the impact of the overhead associated to non-blocking communications. We then provide a new peer-to-peer checkpointing algorithm, called the triple checkpointing algorithm, that can work at constant memory, and achieves both higher efficiency and better risk handling than the double checkpointing algorithm. We provide performance and risk models for all the evaluated protocols, and compare them through comprehensive simulations.**

I. Introduction

Parallel computing environments follow an exponential trend in doubling their size on a regular basis. The Top 500 ranking[1] features a typical illustration of this trend in the High Performance Computing world: the measured performance doubled every 18 months for the last 15 years. Since the multicore revolution, motivated by the impediment of frequency increase, this growth is sustained by the multiplication of cores and sockets in parallel machines. The International Exascale Software Project (IESP) [3], [4] forecasts the Exaflop mark to be reached by high performance supercomputers by 2019-2022. In their study, which proposes an outline of the characteristics of an Exascale machine based on foreseeable limits of the hardware and maintenance costs, a machine in this performance range is expected to be built from GHz processing cores, with thousands of cores per computing node (up to 10^{12} flop/s/node), thus requiring millions of computing nodes to reach the goal.

A major concern in the IESP report is reliability. If we consider that failures of computing nodes are independent, the reliability probability of the whole system (i.e. the probability that all components will be up and running during the next time unit) is the product of the reliability probability of each of the components. A very conservative assumption of a fifty years MTBF (Mean Time Between Failures) translates into a probability of 0.999998 that a node will still be running in the next hour. However, if the system consists of a million of nodes, the probability that at least one unit will be subject to a failure during the next hour jumps to $1 - 0.999998^{10^6} > 0.86$. This probability is significantly high, especially since the machine was used for only one hour. One can conclude that many computing nodes will inevitably fail during the execution of a long-running Exascale application.

A traditional approach to tolerate failures in parallel computing relies on rollback/recovery: processes can take a checkpoint of their state, together (in coordinated checkpointing protocols), or independently (in uncoordinated checkpointing protocols with message logging). In case of failures, they are rolled back from these saved states, to allow further progress of the computation. A critical point of such an approach is to store the checkpoint images efficiently and reliably. One of the reasons why uncoordinated checkpointing can provide a better efficiency than coordinated checkpointing, despite the higher overheads it imposes on a failure free execution, is because it reduces the amount of data transferred at rollback [5].

Zheng, Shi and Kalé [1] consider the issue of where to store the checkpoint images in order to reduce the demand on I/O during checkpoint phases. They proposed a "buddy" algorithm, that we call the double checkpointing algorithm in the rest of this paper, because it creates a copy of the process checkpoint in a remote process. In this algorithm, processes are coupled: checkpoints are kept in the storage space (local drive or memory) of the buddy, and reciprocally. Each process also keeps a local copy (local drive or memory) of its last checkpoint image. In case of failure, all living processes rollback from the local image, while the processes replacing a process victim of a failure load the corresponding checkpoints from the designated buddies. This approach allows to use the high-speed network to transfer the checkpoint, and to distribute the load of checkpoint storage (which happens in a single wave in coordinated checkpointing protocols) between all the peers of the system. The double checkpointing algorithm comes in two versions: after the blocking version of [1], a non-blocking version has been introduced by Ni, Meneses and Kalé [2], which exhibits better performance, because checkpointing can then be (at least partially) overlapped with computations. However, the increased performance comes with a higher risk of fatal failure, which is not addressed in [2].

In this paper, we revisit both versions of the double checkpointing algorithm, and we introduce a unified and

[1] http://www.top500.org

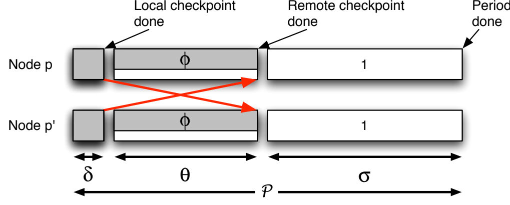


Figure 1: Non-blocking checkpoint algorithm (see [2]).

extended model to assess both the performance and risk of the different strategies. Our first major contribution is to provide a quantitative assessment of the impact of non-blocking checkpointing on both performance and risk. Our second major contribution is the design of a new peer-to-peer checkpointing algorithm, called the triple checkpointing algorithm, that can work at constant memory, and achieves both a higher efficiency and a better risk handling than the double checkpointing algorithm. We provide performance and risk models for all the evaluated protocols, and compare them through comprehensive simulations.

The rest of this paper is organized as follows: in Sec. II, we present the double checkpointing algorithm, together with our model extension to assess the impact of non-blocking protocols. We show how to compute the optimal checkpoint period in Sec. III. Next, we introduce the triple checkpointing algorithm in Sec. IV, and we conduct its analysis in Sec. V. Then, having a performance model for all the algorithms, we instantiate these models using a comprehensive set of parameters, and compare their efficiency and risk in Sec. VI. We discuss related work in Sec. VII. Finally, we provide concluding remarks.

II. The double checkpointing algorithm

In this section, we review and extend the double checkpointing algorithm that has been proposed in the literature, first with a blocking version by Zheng, Shi and Kalé [1], and then with a non-blocking version by Ni, Meneses and Kalé [2]. In both versions, the main idea is to avoid using a centralized stable storage by storing checkpoints in local memory. To avoid the possibility for a single failure to crash the whole application, local checkpoints must be replicated. Thus platform nodes are partitioned into pairs, and each node in a pair exchanges its checkpoint with its *buddy*. As a consequence, each node saves two checkpoints, one locally (storing its own data) and one remotely (receiving and storing its buddy's data), hence the name *double checkpointing*.

The double checkpointing algorithm is a coordinated protocol where nodes operate synchronously. In what follows, we reuse the notations of [2] whenever possible. Also, without loss of generality, it is assumed that the application progresses at unit speed when it is not slow-downed by checkpoint-related activities, so that time-units and work-units can be used indifferently. The non-blocking algorithm is illustrated in Figure 1 and is summarized below:

- Checkpoints are taken periodically, with a period $\mathcal{P} = \delta + \theta + \sigma$
- In the first part of the period, of length δ, each node checkpoints locally, in blocking mode. No application work is performed in parallel.
- In the second part of the period, of length θ, each node checkpoints remotely, i.e. it exchanges its checkpoint file with its buddy. Some application work is performed in parallel, but not at full speed, due to the overhead induced by the concurrent communications for exchanging files. This overhead is expressed as ϕ work units.
- In the third part of the period, of length σ, the application progresses at full speed.

Altogether, in the absence of failures, the work executed during each period of length $\mathcal{P}$ is

$$W = (\theta - \phi) + \sigma = \mathcal{P} - \delta - \phi$$

Note that in the original paper [2], the period is decomposed is only two parts, with $\tau = \theta + \sigma$ being the time after the local checkpoint, letting $\mathcal{P} = \delta + \tau$ and $W = \tau - \phi$. Decomposing the period into three parts is equivalent but makes thing clearer when failures strike (see Sec. III).

A key-feature of the non-blocking algorithm is to overlap computations and checkpoint file exchanges during the second part of the period of length θ, at the price of some overhead ϕ. Intuitively, the larger θ, the more flexibility to hide the cost of the file exchange, hence the smaller the overhead due to checkpointing in the absence of failures. However, in [2], the overhead ϕ is fixed independently of the value of θ. We propose to extend the model as follows:

- When $\theta = \theta_{\text{min}}$, the communication has the smallest possible duration. In this case it is fully blocking, and no computation can take place concurrently. The overhead is 100%, i.e., $\phi = \theta_{\text{min}}$.
- When $\theta = \theta_{\text{max}}$, the communication is made long enough so that it can fully be overlapped with computation. In that case, the overhead is $\phi = 0$.
- We use a linear interpolation between these extremes. The overhead is ϕ when the communication time is

$$\theta(\phi) = \theta_{\text{min}} + \alpha(\theta_{\text{min}} - \phi)$$

We derive that $\phi = 0$ for $\theta = \theta_{\text{max}} = (1 + \alpha)\theta_{\text{min}}$. This last equation gives an intuitive explanation for the parameter α, which measures the rate at which the overhead decreases when the communication length increases.

In a failure-free environment, both θ and σ should be made as large as possible, in order to minimize the overhead due to local and remote checkpointing. This is equivalent to letting the ratio $\frac{W}{\mathcal{P}}$ tend to 1. But the advent of failures calls for smaller period lengths. Indeed, let M denote the Mean Time Between Failures (MTBF) of the platform. When a

failure hits a node, which happens every M time-units in average, the work executed since the last checkpoint is lost, and must be re-executed, which induces an overhead proportional to the loss. The optimal period length $\mathcal{P}$ is the result of the trade-off between minimizing the waste due to checkpointing (large periods) and re-executing only a small amount of work when a failure strikes (small periods).

When a failure hits a node, there is a downtime period of length D for that node, that represent the overhead to detect the failure and allocate a new replacement node for the computation. Then we can start the recovery from the buddy node. There are two checkpoint files that have been lost due to the failure, and which the buddy node must re-send to the faulty processor: (i) the checkpoint file of the faulty node, which is needed for recovery; and (ii) the checkpoint file of the buddy node, which has been lost after the failure and which will be needed if the buddy node would fail later on.

Obviously, the first file (checkpoint of the faulty node) should be sent as fast as possible, i.e. in time θ_{min}, because all processors are stopped until the faulty one has recovered from the failure. Using the notations of [2], the recovery time R is thus equal to θ_{min}. As for the second file (checkpoint of the buddy), there are two possibilities:

- The file is sent at the same speed as in regular (failure-free) mode, in time $\theta(\phi)$. Some overlap is possible, and the overhead is ϕ. This scenario, which we call DOUBLENBL ((NBL for *Non-Blocking*), is the one chosen in [2].
- The file is sent as fast as possible, in time $\theta_{\text{min}} = R$. This scenario, which we call DOUBLEBOF (BOF for *Blocking on Failure*), does not allow for any overlap during the communication.

The application is at risk as long as the faulty processor has not stored a copy of its buddy's checkpoint file. In other words, until complete reception of both messages, it is impossible to recover from a second failure that would hit the buddy. One can say that the DOUBLEBOF favors risk reduction, at the price of a higher overhead, while DOUBLENBL favors performance, at the price of a higher risk. In Sec. III, we provide a detailed analysis of the performance and risk of both strategies.

III. ANALYSIS OF THE DOUBLE CHECKPOINTING ALGORITHM

In this section, we compute the overhead induced by the double checkpointing algorithm, and we analytically determine the optimal checkpointing period.

A. Computing the waste

Let T_{base} be the base time of the application without any overhead due to resilience techniques. First, assume a fault-free execution of the application: every period of length $\mathcal{P}$, only $W = \mathcal{P} - \delta - \phi$ units of work are executed, hence the time T_{ff} for a fault-free execution is

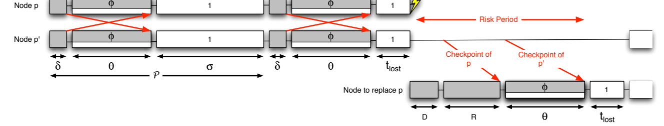


Figure 2: DOUBLENBL strategy, failure during third part of the period

$$T_{\text{ff}} = \frac{\mathcal{P}}{W} T_{\text{base}} \quad (1)$$

Now, let T denote the expectation of the execution time with the double checkpointing algorithm (any version). T can refer to a single application or to the platform life if many jobs are running concurrently. In average, failures occur every M time-units, and for each of them we lose $\mathcal{F}$ time-units, so there are $\frac{T}{M}$ failures during the execution. Hence we derive the equation:

$$T = T_{\text{ff}} + \frac{T}{M}\mathcal{F} \quad (2)$$

which we rewrite as

$$\left(1 - \frac{\mathcal{F}}{M}\right)\left(1 - \frac{\delta + \phi}{\mathcal{P}}\right) T = T_{\text{base}} \quad (3)$$

Defining the *waste* as

$$\text{WASTE} = 1 - \left(1 - \frac{\mathcal{F}}{M}\right)\left(1 - \frac{\delta + \phi}{\mathcal{P}}\right) \quad (4)$$

we can express Equation (3) as $\left(1 - \text{WASTE}\right)T = T_{\text{base}}$. The waste is the fraction of time where nodes do not perform useful computations. In Equation (4), we identify the two sources of overhead: (i) the term $\text{WASTE}_{\text{ff}} = \frac{\delta+\phi}{\mathcal{P}}$, which is the waste due to checkpointing in a fault-free execution, by construction of the algorithm; and (ii) the term $\text{WASTE}_{\text{fail}} = \frac{\mathcal{F}}{M}$, which is the waste due to failures striking during execution. With these notations, Equation (4) writes: $1 - \text{WASTE} = \left(1 - \text{WASTE}_{\text{fail}}\right)\left(1 - \text{WASTE}_{\text{ff}}\right)$, and we derive

$$\text{WASTE} = \text{WASTE}_{\text{fail}} + \text{WASTE}_{\text{ff}} - \text{WASTE}_{\text{fail}}\text{WASTE}_{\text{ff}} \quad (5)$$

There remains to determine the (expected) value of $\mathcal{F}$ in each strategy, denoted as $\mathcal{F}_{\text{nbl}}$ for DOUBLENBL and $\mathcal{F}_{\text{bof}}$ for DOUBLEBOF. Then we will be able to determine the value of $\mathcal{P}$ that minimizes T in Equation (3), or, equivalently, that minimizes WASTE in Equation (5).

DOUBLENBL *strategy:* Here we aim at determining the expected value of $\mathcal{F}_{\text{nbl}}$ for the DOUBLENBL strategy, where the fault node starts by receiving its own checkpoint file in time R before the buddy's checkpoint file in time $\theta(\phi)$. The faulty node undergoes a downtime and recovery, of length $D + R$. Then it starts re-executing the work that has been lost. The amount of work to re-execute depends upon the part of the period where the failure strikes, hence there are three cases. Now, failures strikes uniformly across the period, regardless of the distribution law of the failures:

this is because the instants at which periods begin and at which faults strike are independent. Hence we can compute

$$\mathcal{F}_{\text{nbl}} = D + R + \frac{\delta}{\mathcal{P}}\mathcal{RE}_1 + \frac{\theta}{\mathcal{P}}\mathcal{RE}_2 + \frac{\sigma}{\mathcal{P}}\mathcal{RE}_3 \tag{6}$$

where $\mathcal{RE}_i$ is the expected re-execution time when the failure strikes during the i-th part of the period.

We start with the case when the failure causes the least damage i.e. when it strikes during the third part of the period, after both checkpoints have been taken. See Figure 2 for an illustration. We compute $\mathcal{RE}_3$ as follows:

- The faulty processor cannot execute any work during $D + R$ time-units.
- Then it starts re-executing the work that has been lost, namely $W_{lost} = (\theta - \phi) + t_{lost}$. The first term corresponds to the work executed during the second part of the period, taking into account the overhead associated to this part. The second term comes from the third part of the period, where work executes at full speed.
- During the first θ time-units of re-execution, there is an overhead ϕ due to receiving the buddy's checkpoint. After that, the re-execution of the remaining work $W_{lost} - (\theta - \phi)$ progresses at full speed.

Altogether, the re-execution time is $\theta + W_{lost} - (\theta - \phi) = \theta + t_{lost}$. The expected value of t_{lost} is $\frac{\sigma}{2}$, because failures strike uniformly during the third part of the period. This leads to

$$\mathcal{RE}_3 = \theta + \frac{\sigma}{2}$$

When a failure hits the first part of the period, during local checkpointing, it causes more damage: the work W during the whole previous period has to be re-executed, and we get $W_{lost} = W + t_{lost}$. Just as before, the re-execution time is $\theta + W_{lost} - (\theta - \phi)$. Here the expected value of t_{lost} is $\frac{\delta}{2}$. We derive:

$$\mathcal{RE}_1 = \theta + \sigma + \frac{\delta}{2}$$

When a failure hits the second part of the period, during remote checkpointing, it causes the same damage as during the first part: the work W during the whole previous period has to be re-executed, and we get $W_{lost} = W + t_{lost}$. Again, the re-execution time is $\theta + W_{lost} - (\theta - \phi)$, but the expected value of t_{lost} is even higher than when the failure hits the first part of the period: $t_{lost} = \delta + \frac{\theta}{2}$. We derive:

$$\mathcal{RE}_2 = \theta + \sigma + \delta + \frac{\theta}{2}$$

We are ready to compute the value of $\mathcal{F}_{\text{nbl}}$ using Equation (6). After some simplifications, we obtain

$$\mathcal{F}_{\text{nbl}} = D + R + \theta + \frac{\mathcal{P}}{2} \tag{7}$$

This is almost in accordance with the value reported in [2]: they derive the value $\mathcal{F}_{\text{nbl}} - \phi$ instead of $\mathcal{F}_{\text{nbl}}$, because they forgot the overhead due to receiving the second checkpoint file while re-executing.

DOUBLEBOF *strategy:* Here we aim at determining the expected value of $\mathcal{F}_{\text{bof}}$ for the DOUBLEBOF strategy, where both checkpoint files are received in minimum time $\theta_{\min} = R$. As before, there are three cases, depending upon the part of the period where the failure strikes, and the computation goes in a similar way as for the DOUBLENBL strategy. Indeed, for each part of the period, the amount of work to re-execute W_{lost} is the same, but it is entirely executed at full speed instead of being slowed-down during the first θ time-units (since all communications are already completed). In other words, we add R to the recovery time to account for the second blocking message, and we suppress ϕ from the time needed to re-execute, which leads to:

$$\mathcal{F}_{\text{bof}} = \mathcal{F}_{\text{nbl}} + R - \phi = D + 2R + \theta - \phi + \frac{\mathcal{P}}{2} \tag{8}$$

B. Waste minimization

We use a computer algebra system (Maple[2]) to compute the optimal period that minimizes the total waste. We obtain the following formulas:

$$\mathcal{TO}_{\text{nbl}} = \sqrt{2(\delta + \phi)(M - R - D - \theta)} \tag{9}$$

$$\mathcal{TO}_{\text{bof}} = \sqrt{2(\delta + \phi)(M - 2R - D - \theta + \phi)} \tag{10}$$

There is a similarity with the formulas of Young [6], namely $\mathcal{T} = \sqrt{2M\delta} + \delta$, and of Daly [7], namely $\mathcal{T} = \sqrt{2(M + (D+R))\delta} + \delta$. However, in both these formulas, δ represents the time needed to checkpoint the whole application onto stable storage, while with the (distributed) double checkpointing algorithm, δ is the time needed to checkpoint a single node locally. The value of the optimal period is therefore much larger for the double checkpointing algorithm than for a centralized scheme based on global remote storage. The value of the optimal waste, whose dominant term is $\sqrt{\frac{2\delta}{M}}$ for all reasonably large values of M, is reduced accordingly.

C. Risk

When a failure strikes a node, the application is at risk until the faulty node has recovered and received the copy of its buddy's checkpoint. We let Risk denote the length of the risk period, which is $\mathsf{Risk} = D + 2R$ for DOUBLEBOF and $\mathsf{Risk} = D + R + \theta$ for DOUBLENBL.

In this section, we compute the success probability of the application (no fatal failure throughout execution) for both strategies. Let n be the number of processors in the whole platform. Assume as in [1] that failures strike with uniform distribution over time, and let $\lambda = \frac{1}{nM}$ denote the instantaneous failure rate of each processor. The inverse of λ

[2]http://www.maplesoft.com/products/Maple/

is the individual processor MTBF and is estimated to range from a few years to one century in the literature.

Recall that T denotes the expectation of the execution time of the application with the double checkpointing algorithm (any version). Consider a pair made of one processor and its buddy. The probability of having the first processor fail during execution is λT, and the probability of having the pair failing during execution is $1 - (1 - \lambda T)^2 \approx 2\lambda T$. Now, given that one processor fails, the probability of having the second one fail right after, within the risk period, is $\lambda\mathsf{Risk}$. Hence the probability that the pair experiences a fatal failure during execution is $2\lambda T\lambda\mathsf{Risk}$. Since the application succeeds if and only if all pairs succeed, the probability that the application will fail is $1 - (1 - 2\lambda^2 T\mathsf{Risk})^{n/2}$, or equivalently, the success probability is

$$\mathbb{P}_{\text{double}} = (1 - 2\lambda^2 T\mathsf{Risk})^{n/2} \quad (11)$$

This equation was originally given in [1], except that they forgot the factor 2 to account for the failure probability of both processors in the pair. They also compare the value of $\mathbb{P}_{\text{double}}$ with the probability $\mathbb{P}_{\text{base}}$ that the application will succeed in the absence of checkpointing. In that case, the execution time is T_{base}, and we derive that

$$\mathbb{P}_{\text{base}} = (1 - \lambda T_{\text{base}})^n \quad (12)$$

Equation (11) assesses the impact of the risk period length Risk on the resilience of the application. We can now quantitatively compare the DOUBLENBL and DOUBLEBOF strategies in terms of both performance and reliability. We provide such an evaluation in Sec. VI.

IV. THE TRIPLE CHECKPOINTING ALGORITHM

In this section, we introduce a new algorithm based on processor triples rather than on processor pairs. We show that this new algorithm is both more efficient and reliable than the double checkpointing algorithm, while equally memory-demanding. In fact, the main motivation to design a novel in-memory checkpointing algorithm is to provide a better answer to the following question: given a fixed amount of memory available for checkpointing, what is the best strategy for performance and reliability?

The double checkpointing algorithm of [1], [2] requires that sets of two checkpoint files can be stored in the memory of each processor. As with all coordinated checkpointing protocols, the collection, for all processes in the system, of a set of checkpoints, represents the (global) snapshot of the parallel application. Such sets must be updated atomically. This is implemented by keeping two sets at all time: the last set of checkpoints that was successful (by definition the first set of checkpoints is represented by the starting configuration of the application and is always successful), and the current set of checkpoints, that might be unfinished at the time when a failure hits the system.

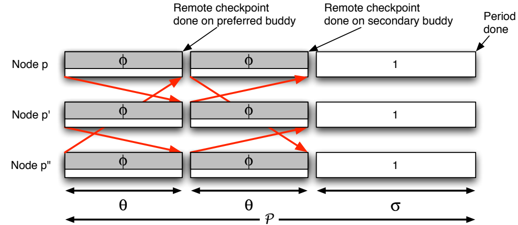


Figure 3: The triple checkpointing algorithm

So, in double checkpointing algorithms, a local set contains two images: the image of the current process and the image of the buddy process. Given this memory constraint, can we do better than pairing each processor with a buddy? In fact, when a failure strikes a processor, the local checkpoint is lost, and must be recovered from the buddy. This calls for replacing the local checkpoint by that of a secondary buddy. Let us consider how checkpoint images are created: a process can create its checkpoint image using the `fork` system call that creates a copy in memory of the current state of the process. Modern operating systems do not create an explicit copy of all the pages of the parent process at the time of the call, but instead mark all pages of the parent process as copy-on-write, allowing to share most, if not all, the process pages. The parent process can continue its work, while the child process uploads its checkpoint image to the buddy file system, releasing its private copy of the pages as soon as they are successfully uploaded. This incurs a minimum overhead to the checkpointing process, and allows a significant overlap of application process and checkpoint transfer. If the rate of transmission is high enough, only a small number of pages will need to be actually duplicated before the child process releases them.

Buddy processes can then store the checkpoint images in their memory or in local storage, as they were doing in the double checkpointing algorithm. The system must decide for a trade-off between taking more time to upload the checkpoint image to the buddy processes in order to reduce the pressure on the network, and taking less time to upload the checkpoint image to the buddy processes in order to reduce the amount of pages that must be created with the copy-on-write mechanism. This trade-off is simplified by ordering the data that is uploaded to the buddy processes from the *most likely to be modified* to the *least likely to be modified* by the ongoing computation.

This idea leads us to the triple checkpointing algorithm, which is illustrated in Figure 3. Processors are organized in triples. Within a triple, each processor p has a preferred buddy p' and a secondary buddy p''. We organize a rotation of buddies, so that p' has p'' for preferred buddy and p for secondary buddy, and p'' has p for preferred buddy and p' for secondary buddy.

From Figure 3, we see that the algorithm operates in a similar way as the double checkpointing algorithm. The

period is still divided into three parts, but the first part is different: the local checkpoint is replaced by sending the checkpoint image to the preferred buddy (and receiving the checkpoint of the secondary buddy). The duration of the first part becomes $\theta(\phi)$ instead of δ. The second part of the period is for an exchange of checkpoint files (sending to the secondary buddy and receiving from the preferred one), and its duration is $\theta(\phi)$, just as before. During the third part of the period, computations proceed at full speed during σ seconds, just as before again.

The main advantage of the new scheme is to dramatically reduce the overhead induced by checkpointing (WASTE_{ff}, the waste due to checkpointing in a fault-free execution, tends to zero) while maintaining a smaller risk, even for large values of θ that might be needed to achieve a fully non-blocking, hence overhead-free, checkpoint. Indeed, there must be three successive failures within a processor triple for the application to experience a fatal failure, instead of two failures striking a processor and its buddy in the double checkpointing algorithm.

One can envision two versions for the triple checkpointing algorithm. After a failure, the preferred buddy always sends the checkpoint file of the faulty node as fast as possible, in blocking mode, and in time $\theta_{\text{min}} = R$. But there is a choice for the next two communications, that correspond to the checkpoint images of the two buddies. Either these communications are executed in blocking mode and in minimum time, or they are executed in overlapped mode, in time $\theta(\phi)$. The first version further reduces the risk, while the second version minimizes the overhead. Because the risk is already very low in both versions, we only deal with the second, non-blocking version, which we denote as TRIPLE.

V. ANALYSIS OF THE TRIPLE CHECKPOINTING ALGORITHM

In this section we compute the waste and the risk of the triple checkpointing algorithm TRIPLE. We use the same notations as in Sec. III. Since the derivation is similar, we omit details and only provide final formulas.

A. *Computing the waste*

As before (see Equation (5)), there are two sources of overhead: WASTE_{ff}, the waste due to checkpointing in a fault-free execution, and $\text{WASTE}_{\text{fail}} = \frac{\mathcal{F}_{\text{tri}}}{M}$, the waste due to failures striking during execution. It is easy to derive that $\text{WASTE}_{\text{ff}} = \frac{2\phi}{\mathcal{P}}$. As for the value of $\mathcal{F}_{\text{tri}}$, we use a modified version of Equation (6) to account for the different lengths of the three parts of the period:

$$\mathcal{F}_{\text{tri}} = D + R + \frac{\theta}{\mathcal{P}}\mathcal{RE}_1 + \frac{\theta}{\mathcal{P}}\mathcal{RE}_2 + \frac{\sigma}{\mathcal{P}}\mathcal{RE}_3 \tag{13}$$

We proceed to determine the (expected) value of $\mathcal{F}$:

- $\mathcal{RE}_1 = 2\theta + \sigma + \frac{\theta}{2}$
- $\mathcal{RE}_2 = \frac{3\theta}{2}$
- $\mathcal{RE}_3 = 2\theta + \frac{\sigma}{2}$

which leads to

$$\mathcal{F}_{\text{tri}} = D + R + \theta + \frac{\mathcal{P}}{2} \tag{14}$$

We observe that the value of $\mathcal{F}$ is the same for DOUBLENBL and TRIPLE ($\mathcal{F}_{\text{nbl}} = \mathcal{F}_{\text{tri}}$). Hence the value of $\text{WASTE}_{\text{fail}}$ is the same too. However, the final waste is different, because we now have $\text{WASTE}_{\text{ff}} = \frac{2\phi}{\mathcal{P}}$ instead of $\text{WASTE}_{\text{ff}} = \frac{\delta+\phi}{\mathcal{P}}$.

B. *Waste minimization*

As before, we use the computer algebra system to compute the optimal period that minimizes the total waste. We obtain the following formula, which is similar to the value obtained for DOUBLENBL (Equation (9)):

$$\mathcal{TO}_{\text{tri}} = 2\sqrt{\phi(M - D - R - \theta)} \tag{15}$$

C. *Risk*

When a failure strikes a node, the application is at risk until the faulty node has recovered and received the copy of its two buddy checkpoints. We let Risk denote the length of the risk period, which is $\mathsf{Risk} = D + R + 2\theta$ (note that it would be reduced to $\mathsf{Risk} = D + 3R$ if we used a blocking-on-failure version of the algorithm).

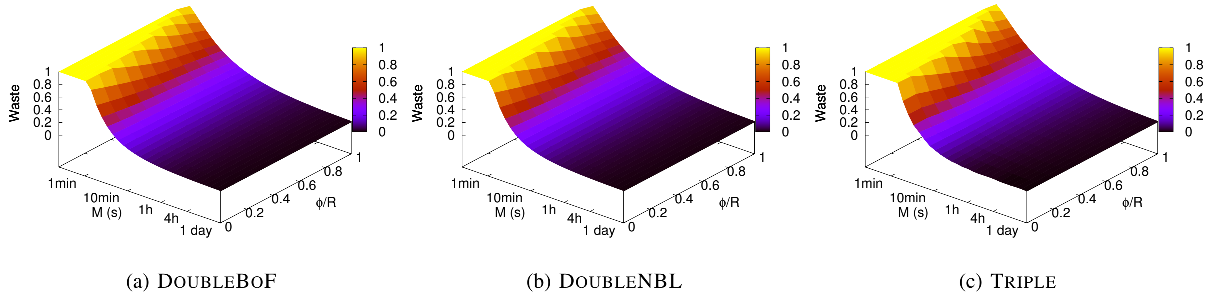
We compute the probability of a fatal failure using the same line of reasoning as before. Let T denote the expectation of the execution time of the application with the triple checkpointing algorithm (any version). Consider a triple made of one processor and its two buddies. The probability of having one of the three processors fail during execution is $3\lambda T$, up to second order terms. Given that one processor fails, the probability of having the pair of remaining processors fail right after, within the risk period, is $2\lambda\mathsf{Risk}$, up to second order terms. And given that situation, the probability that the last processor also fails during the risk period is $\lambda\mathsf{Risk}$. Finally, the probability that the triple experiences a fatal failure during execution is $6\lambda^3 T\mathsf{Risk}^2$. Since the application succeeds if and only if all triples succeed, the probability that the application will succeed is

$$\mathbb{P}_{\text{triple}} = (1 - 6\lambda^3 T\mathsf{Risk}^2)^{n/3} \tag{16}$$

Sec. VI provides a quantitative comparison of the success probability of the double and triple checkpointing algorithms.

VI. EXPERIMENTS

In this section, we consider the efficiency and risk of DOUBLENBL, DOUBLEBOF and TRIPLE, under different realistic conditions. Table I summarizes the two different scenarios that we consider, namely $Base$ taken from [2] and Exa which models future exscale platforms.



(a) DOUBLEBOF (b) DOUBLENBL (c) TRIPLE

Figure 4: Waste for Scenario $Base$, function of ϕ/R and M

Scenario	D	δ	ϕ	R	α	n
$Base$	0	2	$0 \leq \phi \leq 4$	4	10	324×32
Exa	60	30	$0 \leq \phi \leq 60$	60	10	10^6

Table I: Parameters for the different scenarios: D is the down time; δ the time to take the local checkpoint; ϕ, the amount of overhead; R the base time to load a remote checkpoint in blocking mode; α the overlap speedup factor, which defines θ the time to upload a remote checkpoint; n is the number of platform nodes (used for the risk assessment).

A. $Base$ scenario

The $Base$ scenario takes the same values as [2]: for checkpointing a memory of 512MB, the time to produce a local checkpoint at the speed of SSDs is about 2s; for uploading the same amount of data to a remote neighbor, at the considered network speed, the time to upload (without any work in parallel) will be about 4s. Since [2] does not consider the time to allocate a new node on the machine, we let $D = 0$. They consider only two cases for ϕ and α: when the checkpoint operation cannot overlap with any application progress ($\phi = R$), or when checkpointing does not imply any overhead on the progress ($\phi = 0$, and $\alpha > 0$).

Figure 4 presents the waste, with the model-computed optimal checkpoint time, of each algorithm, as a function of ϕ (between 0 and R; the ratio ϕ/R is presented in the figure for normalization) and of M (from $15s$, where no progress happens for any protocol, up to 1 day, where the waste is almost 0 for all), the latter shown on a logarithmic scale. By varying ϕ, we consider the waste when the amount of work that can be done during the checkpoint phase varies from 0 to no overhead at all. Moreover, since $\alpha = 10$, checkpoint communication is completely hidden between application communications if the optimal checkpointing period allows a duration of at least $(\alpha + 1)R = 11R$. We point out that this is a conservative assumption on the communication-to-computation ratio.

Comparing the three subfigures together, one can see that TRIPLE behaves slightly differently than DOUBLENBL and DOUBLEBOF: indeed, TRIPLE takes a higher benefit of a low value of ϕ, because it does not suffer from the period δ during which no progress is done in the double checkpointing protocols. In a realistic setup, ϕ will not go down completely to 0 in the triple checkpointing protocol, because during the checkpoint transfer, some pages may need to be copied by the copy on write mechanism of `fork`; still, a very small ratio ϕ/R can be achieved for large enough values of θ, the file exchange phase. Similarly, using the same approach, the value of δ could be reduced significantly in the double checkpointing protocols, allowing for a better benefit of the ϕ parameter. All three kinds of protocols, however, clearly reduce their waste in a similar fashion when the MTBF increases.

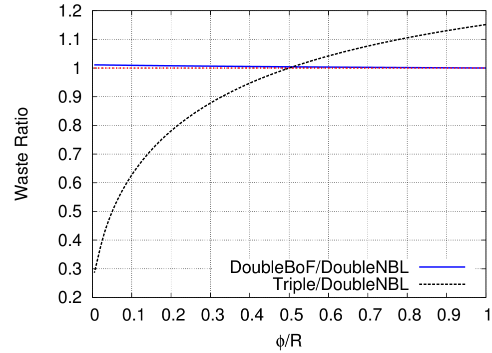
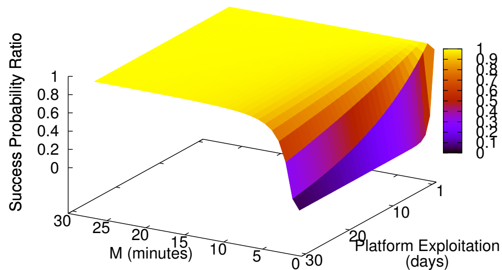


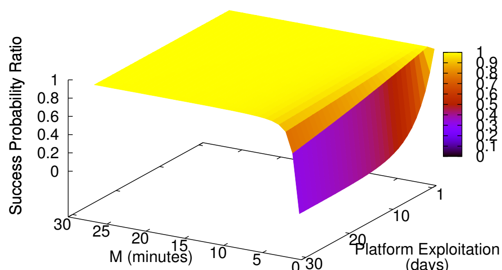
Figure 5: Waste for Scenario $Base$, ($M = 7h$).

The differences between the three protocols is better illustrated with Figure 5 which compares the waste of the three algorithms, with a fixed value for $M = 7h$, relatively to the efficiency of DOUBLENBL. The benefit of a non-blocking approach is small, but noticeable: DOUBLEBOF has always a higher waste than DOUBLENBL, until the ratio of work that can be done during the checkpoint makes waiting for the checkpoint transfer transparent.

Up to $\phi/R \leq 0.5$, TRIPLE has a much smaller waste than any of the double checkpointing protocols. Because the number of faults is low, the dominating part of the waste comes from the failure-free case. TRIPLE does not suffer from a blocking checkpoint time, as DOUBLENBL and DOUBLEBOF do, and thus proves more efficient whenever a large amount of work can be done in parallel with



(a) Ratio DOUBLENBL/ DOUBLEBOF succes probability



(b) Ratio DOUBLEBOF/ TRIPLE succes probability

Figure 6: Relative success probability for Scenario $Base$, function of M and platform life T. $\theta = (\alpha + 1)R$.

the checkpoint. Once more time is spent communicating checkpoint data than computing, however, TRIPLE suffers from its double amount of data to communicate (compared to the double checkpointing approaches, TRIPLE needs to exchange twice the data on the network). The overhead, however, is limited to 15% more waste in the worst case.

Furthermore, DOUBLEBOF and TRIPLE are expected to provide a better risk preservation than DOUBLENBL. This is the property that we quantify next. Figure 6a represents the relative risk between DOUBLEBOF and DOUBLENBL. A lower value means that DOUBLEBOF provides a better risk tolerance. As illustrated by the figure, this is measurable for long period of times (above 10 days), and for very low MTBF ($M \leq 60s$); otherwise all protocols have a success probability almost equal to 1. On this setup, the benefit of blocking during the checkpoint is not significant, even if it induces a waste lower than 2%. Figure 6b presents the same comparison, but with TRIPLE compared to DOUBLENBL, the most secure version of double checkpointing. Again, a lower value means that TRIPLE provides a better risk tolerance than DOUBLENBL. This time, in the same range as where DOUBLEBOF was providing a small risk improvement, the gain is quite significant, providing risk mitigation by orders of magnitude. More importantly, even when the MTBF increases, and the application duration decreases,

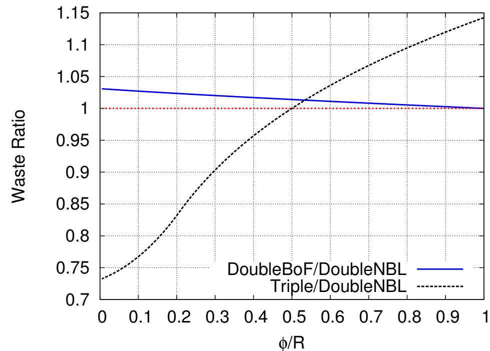


Figure 8: Waste for Scenario Exa, ($M = 7h$).

TRIPLE is able to tolerate twice more runs without incurring a fatal failure than DOUBLENBL. It is striking to point out that these numbers are achieved with $\theta = (\alpha + 1)R$, which corresponds to the largest possible risk duration for TRIPLE.

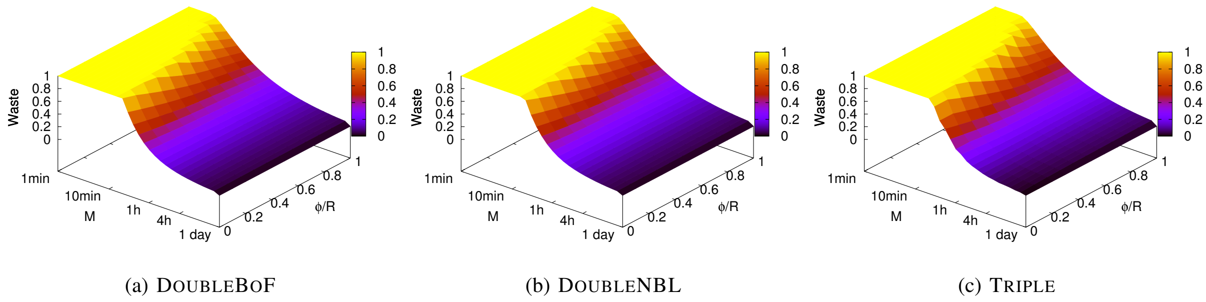
B. *Exa* scenario

We now consider a future exascale machine, as can be envisioned by the IESP work force in [3], [4]. Such a machine is summarized in Table I under the name Exa: based on the assumption of a 1-GHz limit for each core, it will hold 10^9 1-GHz cores. Taking the "slim" exascale assumption, these cores would be distributed among 10^6 nodes, with $1,000$ cores per nodes. Memory previsions plan around 64GB of memory per core, and we took the assumption of a 1TB/s/node network capacity, and 500Gb/s/node of bus limitation for the local storage capacity.

We then conducted the same set of evaluations as with the $Base$ scenario. Results are presented in Figures 7 to 9b. First, we observe the same general behavior. TRIPLE remains more robust than double checkpointing protocols for very high failure frequency and long applications. DOUBLENBL and DOUBLEBOF have a similar waste, also, as expressed by Figure 8, but the gain of TRIPLE increases up to 25% of that of DOUBLENBL when $\phi/R = 1/10$ while being more reliable (see Figure 9b). The model also forecasts that on such machines, the waste will be important when failures hit the system more than once a day. Last, for such an environment, Figure 9a shows that DOUBLEBOF can provide a higher reliability than DOUBLENBL, to a higher extent than on the $Base$ scenario, for long-running applications. As expected, TRIPLE provides an even higher robustness with this respect (see Figure 9b), even with the largest possible risk period ($\theta = (\alpha + 1)R$).

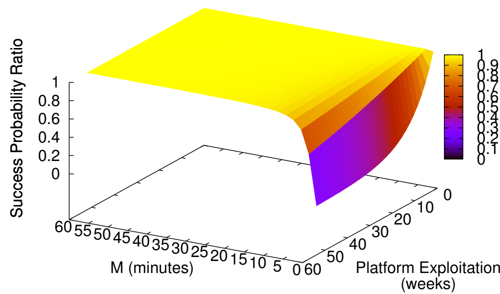
VII. RELATED WORK

Coordinated checkpointing has been studied since many years. The major appeal of the coordinated approach is its simplicity, because a parallel job using n processors of individual MTBF M_{ind} can be viewed as a single processor job with MTBF $M = \frac{M_{ind}}{n}$. Given the value of M, an approximation of the optimal checkpointing period can be

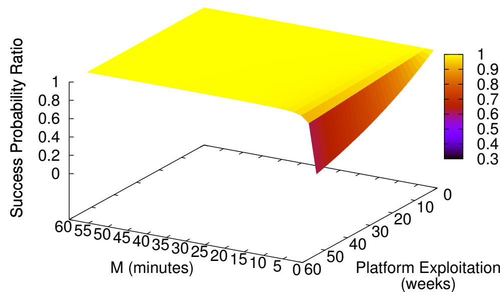


(a) DOUBLEBOF (b) DOUBLENBL (c) TRIPLE

Figure 7: Waste for Scenario Exa, function of ϕ/R and M



(a) Ratio DOUBLENBL/ DOUBLEBOF succes probability



(b) Ratio DOUBLEBOF/ TRIPLE succes probability

Figure 9: Relative success probability for Scenario Exa, function of M and platform life T. $\theta = (\alpha + 1)R$.

computed as a function of the key parameters (downtime D, checkpoint time C and recovery time R). The first estimate had been given by Young [6] and later refined by Daly [7]. Both use a first-order approximation for Exponential failure distributions; their derivation is similar to the approach in Equations (1) to (5). More accurate formulas for Weibull failure distributions are provided in [8], [9], [10]. The optimal checkpointing period is known only for Exponential failure distributions [11]. Dynamic programming heuristics for arbitrary distributions are proposed in [12], [13], [11].

The literature proposes different works [14], [15], [16], [17], [18] on the modeling of coordinated checkpointing protocols. In particular, [15] and [14] focus on the usage of available resources: some may be kept as backup in order to replace the down ones, and others may be even shutdown in order to decrease the failure risk or to prevent storage consumption by saving less checkpoint snapshots.

The major drawback of coordinated checkpointing protocols is their lack of scalability at extreme-scale. These protocols will lead to I/O congestion when too many processes are checkpointing at the same time. Even worse, transferring the whole memory footprint of an HPC application onto stable storage may well take so much time that a failure is likely to take place during the transfer! A few papers [18], [19] propose a scalability study to assess the impact of a small MTBF (i.e., of a large number of processors). The mere conclusion is that checkpoint time should be dramatically reduced for platform waste to become acceptable.

This very conclusion is the major motivation for the development of distributed checkpoint mechanisms. A first idea is to use a multi-level approach, with local disks for a high-rate checkpointing period and global stable storage for a smaller-rate checkpointing period. Another possibility is the in-memory blocking approach (with a buddy) suggested by Zheng, Shi and Kalé [1]. This in-memory checkpointing technique was later extended to a non-blocking version by Ni, Meneses and Kalé [2]. As already mentioned, these two papers constitute the main motivation for this work. While [2] discusses the advantages of the non-blocking version over the blocking version in terms of performance, it fails to mention the augmented risk. This is why we have presented a two-criteria assessment of both versions. In addition, this paper is the first attempt at providing a unified model for quantifying the impact and overhead of checkpointing in parallel with application progress.

VIII. CONCLUSION

Checkpoint transfer and storage are the most critical issues of rollback/recovery protocols for the next few years. The overhead of transferring the checkpoint images to a stable storage dominates the cost related to this approach, and

algorithms that allow to distribute this load among the whole system provide a much better scalability in the number of processors. However, since checkpoint storage is not reliable anymore, these algorithms introduce a risk of non-recoverable failures.

In this work, we have reconsidered the double checkpointing algorithms proposed by Zheng, Shi and Kalé [1] and by Ni, Meneses and Kalé in [2], and we have introduced a new version, the DOUBLEBOF algorithm, that takes the same approach as [2], but tries to reduce the duration of the risk period by focusing all resources to restore a safe state, at the cost of increasing the overhead of each failure. More importantly, we have provided a unified and extended model that allows a performance/risk bi-criteria assessment of existing and future double-checkpointing algorithms. The model incorporates a new parameter α that dictates how fast a checkpoint can be transferred to overlap entirely the transfer cost with application computation.

We have also designed a new triple checkpointing algorithm that builds on modern operating system features to save the checkpoint on two remote processes instead of one, without much more memory or storage requirements. The new algorithm has excellent success probability and almost no failure-free overhead when full overlap of checkpoint transfers can be enabled. We have derived the performance and risk factors of the new algorithm using our unified model, and we have compared these factors to those of both double checkpointing versions. We have instantiated our model with realistic scenarios, which all conclude to the superiority of the triple-checkpointing algorithm.

Future work will proceed along two main directions. First, we took conservatively high values for the new model parameter α in this study, thereby reducing the potential benefit of the triple checkpointing algorithm. We plan to extend this work by studying real-life applications and propose refined values for α for a set of widely-used benchmarks. Second, the perspective of very small MTBF values on future exascale platforms calls for combining distributed in-memory strategies such as those discussed in this paper, with uncoordinated or hierarchical checkpointing protocols with message logging, in order to further reduce the waste due to failure recovery.

Acknowledgments. Y. Robert is with the Institut Universitaire de France. This work was supported in part by the ANR RESCUE project.

REFERENCES

[1] G. Zheng, L. Shi, and L. V. Kale, "FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI," in *Proc. 2004 IEEE Int. Conf. Cluster Computing*. IEEE Computer Society, 2004.

[2] X. Ni, E. Meneses, and L. V. Kalé, "Hiding checkpoint overhead in HPC applications with a semi-blocking algorithm," in *Proc. 2012 IEEE Int. Conf. Cluster Computing*. IEEE Computer Society, 2012.

[3] J. Dongarra, P. Beckman, P. Aerts, F. Cappello, T. Lippert, S. Matsuoka, P. Messina, T. Moore, R. Stevens, A. Trefethen, and M. Valero, "The international exascale software project: a call to cooperative action by the global high-performance community," *Int. J. High Perform. Comput. Appl.*, vol. 23, no. 4, pp. 309–322, 2009.

[4] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, "Toward Exascale Resilience," *Int. J. High Perform. Comput. Appl.*, vol. 23, no. 4, pp. 374–388, 2009.

[5] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello, "MPICH-V: a multiprotocol fault tolerant MPI," *IJHPCA*, vol. 20, no. 3, pp. 319–333, 2006.

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

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

[8] Y. Ling, J. Mi, and X. Lin, "A variational calculus approach to optimal checkpoint placement," *IEEE Trans. Computers*, pp. 699–708, 2001.

[9] T. Ozaki, T. Dohi, H. Okamura, and N. Kaio, "Distribution-free checkpoint placement algorithms based on min-max principle," *IEEE TDSC*, pp. 130–140, 2006.

[10] 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.

[11] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien, "Checkpointing strategies for parallel jobs," in *SC'2011 Supercomputing Conference*. ACM Press, 2011.

[12] S. Toueg and O. Babaoglu, "On the optimum checkpoint selection problem," *SIAM J. Computing*, vol. 13, no. 3, pp. 630–649, 1984.

[13] M.-S. Bouguerra, D. Trystram, and F. Wagner, "Complexity analysis of checkpoint scheduling with variable costs," *IEEE Transactions on Computers*, vol. 99, no. PrePrints, 2012.

[14] J. S. Plank and M. G. Thomason, "Processor allocation and checkpoint interval selection in cluster computing systems," *J. Parallel Dist. Computing*, vol. 61, p. 1590, 2001.

[15] H. Jin, Y. Chen, H. Zhu, and X.-H. Sun, "Optimizing HPC fault-folerant environment: an analytical approach," in *ICPP' 2010*. IEEE Computer Society, 2010.

[16] L. Wang, P. Karthik, Z. Kalbarczyk, R. Iyer, L. Votta, C. Vick, and A. Wood, "Modeling Coordinated Checkpointing for Large-Scale Supercomputers," in *Proceedings of ICDSN'05*, 2005, pp. 812–821.

[17] R. Oldfield, S. Arunagiri, P. Teller, S. Seelam, M. Varela, R. Riesen, and P. Roth, "Modeling the impact of checkpoints on next-generation systems," in *Proceedings of IEEE MSST'07*, 2007, pp. 30 –46.

[18] Z. Zheng and Z. Lan, "Reliability-aware scalability models for high performance computing," in *Proc. of IEEE Cluster'09*, 2009, pp. 1 –9.

[19] F. Cappello, H. Casanova, and Y. Robert, "Preventive migration vs. preventive checkpointing for extreme scale supercomputers," *Parallel Processing Letters*, vol. 21, no. 2, pp. 111–132, 2011.