Using replication for resilience on exascale systems

Marin Bougeret, Henri Casanova, Yves Robert, Frédéric Vivien, Dounia Zaidouni

Using replication for resilience on exascale systems

Marin Bougeret, Henri Casanova, Yves Robert, Frédéric Vivien, Dounia Zaidouni

RESEARCH REPORT

N° 7830

December 2011

Project-Team GRAAL

ISSN 0249-6399 ISRN INRIA/RR--7830--FR+ENG

Using replication for resilience on exascale systems

Marin Bougeret*, Henri Casanova†, Yves Robert‡§, Frédéric Vivien¶, Dounia Zaidouni§

Project-Team GRAAL

Research Report n° 7830 — December 2011 — 32 pages

Abstract: High performance computing applications must be tolerant to faults, which are common occurrences especially in post-petascale settings. The traditional fault-tolerance solution is checkpoint-rollback, by which the application saves its state to secondary storage throughout execution and recover from the latest saved state in case of a failure. An oft studied research question is that of the optimal checkpointing strategy: when should checkpoints be saved. Unfortunately, even using an optimal checkpointing strategy, the frequency of checkpointing must increase as platform scale increases, leading to higher checkpointing overhead. This overhead precludes high parallel efficiency for large-scale platforms, thus mandating other more scalable fault-tolerance mechanisms. One such mechanism is replication, which can be used in addition to checkpoint-rollback. Using replication, multiple processors perform the same computation so that a processor failure does not necessarily mean application failure. While at first glance replication may seem wasteful, it may be significantly more efficient than using solely checkpoint-rollback at large scale. In this work we investigate two approaches for replication. In the first approach, each process in a single instance of a parallel application is (transparently) replicated. In the second approach, entire application instances are replicated. We provide a theoretical study of these two approaches, comparing them to checkpoint-rollback only, in terms of expected application execution time.

Key-words: Fault-tolerance, scheduling, checkpoint, resilience, exascale, replication

* LIRMM Montpellier, France
† Univ. of Hawai'i at Manoa, Honolulu, USA
‡ University of Tennessee Knoxville, USA
§ Ecole Normale Supérieure de Lyon, France
¶ INRIA

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

La réplication pour la résilience des systèmes exascales

Résumé : Les applications de calcul haute-performance doivent être tolérantes aux pannes, et ce d'autant plus que les pannes seront fréquentes dans les plates-formes post-petascale. La solution traditionnelle de tolérance aux pannes est la sauvegarde de points de reprise (*checkpoint*) et le retour-arrière (*rollback*). Dans ce cadre, une application, au cours de son exécution, sauve son état dans un espace de stockage secondaire, état à partir duquel elle redémarrera en cas de panne. Une question souvent étudiée est celle de la politique de sauvegarde optimale: quand les points de reprises doivent-ils être pris ? Malheureusement, même avec une politique de sauvegarde optimale, la fréquence des sauvegardes doit augmenter avec la taille de la plate-forme, induisant une augmentation du surcoût dû au mécanisme de tolérance aux pannes. Ce surcoût interdit d'atteindre une bonne efficacité pour les applications parallèles sur plates-formes de très grande taille. D'autre mécanismes de tolérance aux pannes doivent donc être utilisés. Un de ces mécanismes est la réplication, qui peut être utilisée en association avec un mécanisme de sauvegarde de points de reprise. Avec la réplication, plusieurs processeurs exécutent le même calcul de telle sorte que la panne d'un processeur n'implique pas forcément une interruption de l'exécution de l'application. Bien que la réplication paraisse, à première vue, être un gaspillage de ressources, utiliser conjointement réplication et sauvegarde de points de reprise peut s'avérer significativement plus efficace que la seule utilisation des points de reprise, sur les plates-formes de très grande taille. Dans ce travail nous considérons deux mises en œuvre de la réplication. Dans la première approche, chaque processus d'une unique instance d'une application parallèle est répliqué (de manière transparente). Dans la seconde approche, des instances entières de l'application sont répliquées. Nous menons une étude théorique de ces deux approches, et nous les comparons à la seule sauvegarde des points de reprise, du point de vue de l'espérance du temps de complétion.

Mots-clés : Tolérance aux fautes, résilience, checkpoint, ordonnancement, réplication, exascale

1 Introduction

As plans are made for deploying post-petascale high performance computing (HPC) systems [8, 17], solutions need to be developed to ensure that applications on such systems are resilient to faults. Resilience is particularly critical for applications that enroll large numbers of processors, including those applications that are pushing the limit of current computational capabilities and that could benefit from enrolling all available processors. However, for such applications, processor failure is the common case rather than the exception. For instance, the 45,208-processor Jaguar platform is reported to experience on the order of 1 failure per day [16, 2], and its scale is modest compared to platforms in the plans for the next decade. Unfortunately, not all faults can be automatically detected and corrected in hardware, which leads to failures. For such faults, rollback recovery is used to resume job execution from a previously saved fault-free execution state, or *checkpoint*. Rollback recovery implies frequent (usually periodic) *checkpointing* events at which the job state is saved to resilient storage. More frequent checkpoints lead to higher overhead during fault-free execution, but less frequent checkpoints lead to a larger loss when a failure occurs. A *checkpointing strategy* then specifies when checkpoints should be taken.

To achieve high performance in a failure-prone environment, it is necessary to design efficient checkpointing strategies, i.e., ones that minimize expected job execution time. A large literature is devoted to identifying good strategies, including both theoretical and practical efforts. The former typically rely on assumptions regarding the probability distributions of times to failure of the processors (e.g., Exponential, Weibull), while the later rely on simulations driven by failure datasets obtained on real-world platforms. In a previous paper [4], we have made several contributions in this context, including optimal solutions for Exponential failures and dynamic programming solutions for Weibull failures.

A major problem with checkpoint-rollback is its lack of scalability as platforms become large: the necessary checkpoint frequency for tolerate faults in large-scale platforms is so large that processors spend more time saving state than computing state. It is thus expected that future platforms will lead to unacceptably low parallel efficiency if only checkpoint-rollback is used, no matter how good the checkpointing strategy. Consequently, additional mechanisms must be used. In this work we focus on *replication*: several processors perform the same computation synchronously, so that a fault on one of these processors does not lead to an application failure. Replication is an age-old fault-tolerant technique, but it has gained traction in the HPC context only relatively recently. While replication wastes compute resources in fault-free executions, it can alleviate the poor scalability of checkpoint-rollback.

We study two replication approaches. Consider a parallel application that is *moldable*, meaning that it can be executed on an arbitrary number of processors, which each processor running one application process. In the first approach, *process replication*, a single instance of the application is executed but each application process is (transparently) replicated. For instance, instead of executing the application with $2n$ distinct processes on a $2n$-processor platform, one executes the application with n processes so that there are two replicas of each process each running on a distinct physical processor. The advantage of this approach is that the mean time to failure of a group of two replicas is larger than that of a single processor, meaning that the checkpointing frequency can

be lowered in a view to improving parallel efficiency. In the second approach, *group replication*, multiple application instances are executed. For the same example, one could execute 2 distinct n-process application instances on the $2n$-processor platform. Each instance runs at a smaller scale, meaning that it has better parallel efficiency than a single $2n$-process instance due to a smaller checkpointing frequency. Furthermore, once an instance saves a checkpoint, it is possible for another instance to use this checkpoint immediately.

Given the above, our contributions in this work are:

- A theoretical analysis of the optimal number of processors to use for a checkpoint-rollback execution of a parallel application for various parallel workload models;
- A theoretical analysis of process replication, which leads to a dynamic programming solution for determining a good checkpointing policy.
- A simple yet effective algorithm for group replication and a theoretical analysis that yields a bound on the expected application execution time achieved by this algorithm.

This paper is organized as follows. Section 2 discusses related work. Section 3 defines our theoretical framework and states our key assumptions. Section 4 discusses the optimal number of processors for a checkpoint-rollback execution of a parallel application. Section 5 presents our results for process replication, and Section 6 presents our results for group replication. Finally, Section 7 provides some final remarks and perspectives.

2 Related work

Checkpointing policies have been widely studied in the literature. In [7], Daly studies periodic checkpointing policies for Exponentially distributed failures, generalizing the well-known bound obtained by Young [23]. Daly extended his work in [13] to study the impact of sub-optimal checkpointing periods. In [20], the authors develop an "optimal" checkpointing policy, based on the popular assumption that optimal checkpointing must be periodic. In [5], 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 [4], we have shown that this assumption is unreasonable for Weibull failure. We have developed optimal solutions for Exponential failures and dynamic programming solutions for Weibull failures, demonstrating performance improvements over checkpointing approaches proposed in the literature in the case of Weibull failures. Note that Weibull distribution is recognized as a reasonable approximation of failures in real-world systems [12, 19]. The work in this paper relates to checkpointing policies in the sense that we study a replication mechanism that is used as an addition to checkpointing. Part of our results build on the algorithms and results developed in [4].

In spite of all the above advances in the areas of checkpointing policies, several studies have questioned the feasibility of pure checkpoint-rollback for large-scale systems (see [10] for a discussion of this issue and for references to

such studies). In this work we study the used of replication as a mechanism complementary to checkpoint-rollback. Replication has long been used as a fault-tolerance mechanism in distributed systems [11] and more recently in the context of volunteer computing [15]. The idea to use replication together with checkpoint-rollback has been studied in the context of grid computing [22]. One concern about replication in HPC is the induced resource waste. However, given the scalability limitations of pure checkpoint-rollback, replication has recently received more attention in the HPC literature [18, 24, 9]. Most recently, the work by Ferreira et al. [10] has studied the use of process replication for MPI applications. They provide a theoretical analysis of parallel efficiency, an implementations of MPI that supports transparent process replication, and a set of convincing experimental and simulation results. The work in [10] only considers 2 replicas per application process. The theoretical analysis, admittedly not the primary objective of the authors, is not developed in details. Furthermore, it relies on an analogy with the birthday problem to compute the MTTF of the machine, which turns out to be incorrect. In Section 5 of this work we provide a full-fledge theoretical analysis of process replication.

3 Framework

We consider the execution of a tightly-coupled parallel application, or *job*, on a large-scale platform composed of p 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. We assume that a standard checkpointing and roll-back recovery is performed at the system level.

The job must complete $\mathcal{W}$ units of (divisible) work, which can be split arbitrarily into separate *chunks*. The job can execute on any number $q \leq p$ processors. Letting $\mathcal{W}(q)$ be the time required for a failure-free execution on q processor, we use three models:

- Embarrassingly parallel jobs: $\mathcal{W}(q) = \mathcal{W}/q$.
- Generic parallel jobs: $\mathcal{W}(q) = \mathcal{W}/q + \gamma\mathcal{W}$. As in Amdahl's law [1], $\gamma < 1$ is the fraction of the work that is inherently sequential.
- Numerical kernels: $\mathcal{W}(q) = \mathcal{W}/q + \gamma\mathcal{W}^{2/3}/\sqrt{q}$. This is representative of a matrix product or a LU/QR factorization of size N on a 2D-processor grid, where $\mathcal{W} = O(N^3)$. In the algorithm in [3], $q = r^2$ 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.

Each participating processor is subject to *failures*. A failure causes a *downtime* period of the failing processor, of duration D. When a processor fails, the whole execution is stopped, and all processors must recover from the previous checkpointed state. We let $C(q)$ denote the time needed to perform a checkpoint, and $R(q)$ the time to perform a recovery. The downtime accounts for software rejuvenation (i.e., rebooting [14, 6]) or for the replacement of the failed processor by a spare. Regardless, we assume that after a downtime the processor is fault-free and begins a new lifetime at the beginning of the recovery period. This recovery period corresponds to the time needed to restore the last

checkpoint. Assuming that the application's memory footprint is V bytes, with each processor holding V/q bytes, we consider two scenarios:

- Proportional overhead: $C(q) = R(q) = \alpha V/q = C/q$ for some constant α. This is representative of cases in which the bandwidth of the network card/link at each processor is the I/O bottleneck.
- Constant overhead: $C(q) = R(q) = \alpha V = C$, which is representative of cases in which the bandwidth to/from the resilient storage system is the I/O bottleneck.

We asssume coordinated checkpointing [21], meaning that no message logging/replay is needed when recovering from failures. Finally, we assume that failures can happen during recovery or checkpointing, but not during a downtime (otherwise, the downtime period could be considered part of the recovery period).

We assume that the parallel job is tightly coupled, meaning that all q processors operate synchronously throughout the job execution. These processors execute the same amount of work $\mathcal{W}(q)$ in parallel, chunk by chunk. The total time (on one processor) to execute a chunk of size ω, and then checkpointing it, is $\omega + C(q)$. Finally, we assume that failure arrivals at all processors are independent and identically distributed (*iid*).

4 Optimal number of processors for execution

In this section we consider a parallel job of size $\mathcal{W}$ executing on q processors, with values for $\mathcal{W}(q)$, $C(q)$ and $R(q)$ given by one of the previous scenarios.

We assume that failure laws follow an Exponential distribution law. Let $\mathbb{E}(q)$ be the expectation of the execution time when using q processors. Is it true than choosing $q = p$ minimizes this quantity? Otherwise, what can we say about the optimal number of processors q_{opt} which minimizes $\mathbb{E}(q)$? This question was partially and empirically addressed in [20], via experiments for 4 MPI applications for up to 35 processors. Our approach here is radically different since we target large-scale platforms and seek theoretical results in the form of optimal solutions. The main objective of this section is to show analytically that, for Exponential failures, $\mathbb{E}(q)$ decreases and then increases as q increases, and thus admits a unique minimum.

Assume that failure inter-arrival times follow an Exponential distribution with parameter λ. In our recent work [4], we have shown that the optimal strategy to minimize the expected makespan $\mathbb{E}(q)$ is to split $\mathcal{W}$ into $K^* = \max(1, \lfloor K_0(q) \rfloor)$ or $K^* = \lceil K_0(q) \rceil$ same-size chunks, whichever leads to the smaller value, where $K_0(q) = \frac{q\lambda\mathcal{W}(q)}{1+\mathbb{L}(-e^{-q\lambda C(q)-1})}$ is the optimal (non integer) number of chunks. $\mathbb{L}$ denotes the Lambert function, defined as $\mathbb{L}(z)e^{\mathbb{L}(z)} = z$. This result shows that the optimal strategy is periodic. The optimal expectation of the makespan is computed as:

$$\mathbb{E}^*(q) = K^*(q)\left(\frac{1}{q\lambda} + \mathbb{E}(T_{rec}(q))\right)\left(e^{\frac{q\lambda\mathcal{W}(q)}{K^*(q)}+q\lambda C(q)} - 1\right) \quad (1)$$

where $\mathbb{E}(T_{rec}(q))$ denotes the expectation of the recovery time, i.e., the time spent recovering from failure during the computation of a chunk. All chunks

have the same recovery time because they all have the same size and because of the memoryless property of the Exponential distribution. It turns out that although we can compute the optimal number of chunks (and thus the chunk size), we cannot compute $\mathbb{E}^*(q)$ analytically because $\mathbb{E}(T_{rec}(q))$ is difficult to compute. This is because failures can occur during recovery. Many previous works conveniently assume that no failure occurs during recovery. To circumvent this difficulty we write the following recursion:

$$T_{rec}(q) = \begin{cases} X_D(q) + R(q) & \text{if no processor fails during} \\ & R(q) \text{ units of time,} \\ X_D(q) + T_{lost}(R(q)) + T_{rec}(q) & \text{otherwise.} \end{cases} \quad (2)$$

$X_D(q)$ is the downtime of a group of q processors, that is the time between the first failure of one of the processors and the first time at which all of them are available (accounting for the fact a processor can fail while another one is down, thus prolonging the downtime). $T_{lost}(R(q))$ is the amount of time spent computing by these processors before a first failure, knowing that the next failure occurs within the next $R(q)$ units of time. In other terms, it is the compute time that is wasted because checkpoint recovery was not completed. The time until the next failure of a group of q processors is the minimum of q *iid* Exponentially distributed variables, and is thus Exponential with parameter $q\lambda$. We can thus compute $\mathbb{E}(T_{lost}(R(q))) = \frac{1}{q\lambda} - \frac{R(q)}{e^{q\lambda R(q)}-1}$ (see [4] for details). Plugging this value into Equation 2 leads to:

$$\begin{aligned} \mathbb{E}(T_{rec}(q)) = & \\ & e^{-q\lambda R(q)}(\mathbb{E}(X_D(q)) + R(q)) \\ & + (1 - e^{-q\lambda R(q)}) \left(\mathbb{E}(X_D(q)) + \frac{1}{q\lambda} - \frac{R(q)}{e^{q\lambda R(q)} - 1} + \mathbb{E}(T_{rec}(q)) \right) \end{aligned} \quad (3)$$

Equation 3 reads as follows: after the downtime $X_D(q)$, either the recovery succeeds for everybody, or there is a failure during the recovery and we have to make another attempt. Both events are weighted by their respective probabilities. Simplifying the above expression we get:

$$\mathbb{E}(T_{rec}(q)) = \mathbb{E}(X_D(q))e^{q\lambda R(q)} + \frac{1}{q\lambda}(e^{q\lambda R(q)} - 1) \quad (4)$$

The difficulty to compute $\mathbb{E}(T_{rec}(q))$ now comes from the $\mathbb{E}(X_D(q))$ term. With a single processor ($q = 1$), $X_D(q)$ has constant value D, but with several processors there could be cascading downtimes. At any rate, we can compute the following lower and upper bounds for $\mathbb{E}(X_D(q))$:

Proposition 1. *Let $X_D(q)$ denote the downtime of a group of q processors. Then*

$$D \leq \mathbb{E}(X_D(q)) \leq \frac{e^{(q-1)\lambda D} - 1}{(q-1)\lambda} \quad (5)$$

Proof. We always have $X_D(q) \geq X_D(1) \geq D$, hence the lower bound. For the upper bound, consider a date at which one of the q processors, say processor i_0, just had a failure and initiates its downtime period for D time units. Some other processors might be in the middle of their downtime period: for each

processor i, $1 \leq i \leq q$, let t_i denote the remaining duration of the downtime of processor i. We have $0 \leq t_i \leq D$ for $1 \leq i \leq q$, $t_{i_0} = D$, and $t_i = 0$ means that processor i is up and running. Let $X_D^{t_1,..,t_q}(q)$ be the *remaining* downtime of a group of q processors, knowing that processor i, $1 \leq i \leq q$, will still be down for a duration of t_i, and that a failure just happened (i.e., there exists i_0 such that $t_{i_0} = D$). Given the values of the t_i's, we have the following equation for the random variable $X_D^{t_1,..,t_q}(q)$:

$$X_D^{t_1,..,t_q}(q) = \begin{cases} D \\ \quad \text{if none of the processors of the group fails} \\ \quad \text{during the next } D \text{ units of time} \\ T_{lost}^{t_1,..,t_q}(D) + X_D^{t'_1,..,t'_q}(q) \\ \quad \text{otherwise.} \end{cases}$$

In the second case of the equation, consider the next D time-units. Processor i can only fail in the *last* $D - t_i$ of these time-units. Here the values of the t'_i's depend on the t_i's and on $T_{lost}^{t_1,..,t_q}(D)$. Indeed, except for the last processor to fail, say i_1, for which $t'_{i_1} = D$, we have $t'_i = \max\{t'_i - T_{lost}^{t_1,..,t_q}(D), 0\}$. More importantly we always have $T_{lost}^{t_1,..,t_q}(D) \leq T_{lost}^{D,0,..,0}(D)$ and $X_D^{t_1,..,t_q}(q) \leq X_D^{D,0,..,0}(q)$ because the probability for a processor to fail during D time units is always larger than that to fail during $D - t_i$ time-units. Thus, $\mathbb{E}(X_D^{t_1,..,t_q}(q)) \leq \mathbb{E}(X_D^{D,0,..,0}(q))$. Following the same line of reasoning, we derive an upper-bound for $X_D^{D,0,..,0}(q)$:

$$X_D^{D,0,..,0}(q) \leq \begin{cases} D \\ \quad \text{if none of the } q-1 \text{ running processors of the group fails} \\ \quad \text{during the downtime } D \\ T_{lost}^{D,0,..,0}(D) + X_D^{D,0,..,0}(q) \\ \quad \text{otherwise.} \end{cases}$$

Weighting both cases by their probability and taking expectations, we obtain

$$\mathbb{E}\left(X_D^{D,0,..,0}(q)\right) \leq e^{-(q-1)\lambda D} D + (1 - e^{-(q-1)\lambda D}) \left(E\left(T_{lost}^{D,0,..,0}(D)\right) + E\left(X_D^{D,0,..,0}(q)\right)\right)$$

hence $\mathbb{E}\left(X_D^{D,0,..,0}(q)\right) \leq D + (e^{(q-1)\lambda D} - 1) E\left(T_{lost}^{D,0,..,0}(D)\right)$, with $E\left(T_{lost}^{D,0,..,0}(D)\right) = \frac{1}{(q-1)\lambda} - \frac{D}{e^{(q-1)\lambda D} - 1}$. We derive

$$\mathbb{E}\left(X_D^{t_1,..,t_q}(q)\right) \leq \mathbb{E}\left(X_D^{0,..,0}(q)\right) \leq \frac{e^{(q-1)\lambda D} - 1}{(q-1)\lambda} .$$

which concludes the proof. As a sanity check, we observe that the upper bound is at least D, using the identity $e^x \geq 1 + x$ for $x \geq 0$. □

We use the lower bound on $\mathbb{E}(X_D(q))$ to prove the following result: for several relevant scenarios, the expected execution time $\mathbb{E}^*(q)$ is minimum when using a finite number of processors (while in a failure free environment, it would always decrease as q increases). We obtain the following theorem:

Theorem 1. *$\mathbb{E}^*(q)$ reaches its minimum for some finite value of q in the following scenarios: all job types (embarrassingly parallel, generic and numerical) with constant overhead, and generic or numerical jobs with proportional overhead.*

Proof. We show that $\lim_{q\to+\infty} \mathbb{E}^*(q) = +\infty$ for the relevant scenarios. We first plug the lower-bound of Equation 5 into Equation 4 and obtain:

$$\mathbb{E}(T_{rec}(q)) \geq De^{q\lambda R(q)} + \frac{1}{q\lambda}\left(e^{q\lambda R(q)} - 1\right).$$

From Equation 1 we then derive the lower-bound:

$$\mathbb{E}^*(q) \geq K_0(q)\left(\frac{1}{q\lambda} + D\right)e^{q\lambda R(q)}\left(e^{\frac{q\lambda\mathcal{W}(q)}{K_0(q)}+q\lambda C(q)} - 1\right)$$

using the fact that, by definition, the expression in the right hand-side of Equation 1 is minimized by K_0, where $K_0(q) = \frac{q\lambda\mathcal{W}(q)}{1+\mathbb{L}(-e^{-q\lambda C(q)-1})}$.

Embarrassingly parallel jobs with constant overhead. Here we assume that $\mathcal{W}(q) = \mathcal{W}/q$ and use constant overhead $C(q) = R(q) = C$. We get the lower bound:

$$\mathbb{E}^*(q) \geq K_0(q)\left(\frac{1}{q\lambda} + D\right)e^{q\lambda C}\left(e^{\frac{\lambda\mathcal{W}}{K_0(q)}+q\lambda C} - 1\right)$$

where $K_0(q) = \frac{\lambda\mathcal{W}}{1+\mathbb{L}(-e^{-q\lambda C-1})}$. We show that $\lim_{q\to+\infty} \mathbb{E}^*_{min}(q) = +\infty$. As a consequence, we will also have $\lim_{q\to+\infty} \mathbb{E}^*(q) = +\infty$, hence the desired result. When q tends to $+\infty$, $K_0(q)$ tends to $\lambda\mathcal{W}$, while $(\frac{1}{q\lambda}+D)e^{q\lambda C}\left(e^{\frac{\lambda\mathcal{W}}{K_0(q)}+q\lambda C} - 1\right)$ tends to $+\infty$. This concludes the proof. This result also implies that $\mathbb{E}^*(q)$ reaches a minimum for a finite q value for other job types (generic, numerical) with constant overhead, just because the execution time is larger in that case than with embarrassingly parallel jobs.

Generic parallel job with proportional overhead. Here we assume that $\mathcal{W}(q) = \mathcal{W}/q + \gamma\mathcal{W}$, and use proportional overhead: $C(q) = R(q) = \frac{C}{q}$. We get the lower bound:

$$\mathbb{E}^*(q) \geq K_0(q)\left(\frac{1}{q\lambda} + D\right)e^{\lambda C}\left(e^{\frac{\lambda\mathcal{W}+q\lambda\gamma\mathcal{W}}{K_0(q)}+\lambda C} - 1\right)$$

where $K_0(q) = \frac{\lambda\mathcal{W}+q\lambda\gamma\mathcal{W}}{1+\mathbb{L}(-e^{-\lambda C-1})}$. As before, we show that $\lim_{q\to+\infty} \mathbb{E}^*_{min}(q) = +\infty$ to get the result. When q tends to $+\infty$, $K_0(q)$ tends to $+\infty$, while $(\frac{1}{q\lambda} + D)e^{\lambda C}\left(e^{\frac{\lambda\mathcal{W}+q\lambda\gamma\mathcal{W}}{K_0(q)}+\lambda C} - 1\right)$ tends to some positive constant. This concludes the proof. Note that this proof also serves for generic parallel jobs with constant overhead, simply because the execution time is larger in that case than with proportional overhead.

Numerical kernels with proportional overhead. Here we assume that $\mathcal{W}(q) = \mathcal{W}/q + \gamma\mathcal{W}^{2/3}/\sqrt{q}$, and use proportional overhead: $C(q) = R(q) = \frac{C}{q}$. We get the lower bound:

$$\mathbb{E}^*(q) \geq K_0(q)\left(\frac{1}{q\lambda} + D\right)e^{\lambda C}\left(e^{\frac{\lambda\mathcal{W}+\lambda\gamma\mathcal{W}^{2/3}\sqrt{q}}{K_0(q)}+\lambda C} - 1\right)$$

where $K_0(q) = \frac{\lambda\mathcal{W}+\lambda\gamma\mathcal{W}^{2/3}\sqrt{q}}{1+\mathbb{L}(-e^{-\lambda C-1})}$. As before, we show that $\lim_{q\to+\infty}\mathbb{E}^*_{min}(q) = +\infty$ to get the result. When q tends to $+\infty$, $K_0(q)$ tends to $+\infty$, while $(\frac{1}{q\lambda} + D)e^{\lambda C}\left(e^{\frac{\lambda\mathcal{W}+\lambda\gamma\mathcal{W}^{2/3}\sqrt{q}}{K_0(q)}+\lambda C} - 1\right)$ tends to some positive constant. This concludes the proof.

□

Note that the only open scenario is with embarrassingly parallel jobs and proportional overhead: the lower bound for $\mathbb{E}^*(q)$ decreases to some constant while the upper bound tends to infinity as q tends to infinity.

5 Process Replication

Process replication was recently studied in [10], in which the authors propose to replicate each application process transparently on two processors. Only when both these processors fail must the job recover from the previous checkpoint. If there are p available processors, the job executes on $p/2$ pairs of processors. By definition, one replica performs redundant computations, which may be seen as a waste of resources. However, the probability that both replicas fail is much smaller than that of a single replica, thereby allowing to reduce the checkpoint frequency. The results in [10] show large performance improvements due to process replication. The authors also develop an MPI library that implements transparent process replication (failure detection, consistent message ordering among replicas, etc.).

The objective of this section is to provide a full theoretical analysis of Process replication, considering the general case in which each application process is replicated $g \geq 2$ times. In the following we call *replica-group* the set of all the replicas of a given process, and we denote by n_{rg} the number of replica-groups. Altogether, there are $replica - group \times n_{rg} \leq p$ processes running on the platform.

5.1 Mean number of failures needed to bring down an application

Following [10], we assume that when one of the g replicas of a replica-group fails, it is not restarted and the execution of the application proceeds as long as there is still at least one running process in each of the replica-groups [1].

[1] One can envision a scenario where the failed process is restarted based on the *current* state of the remaining replicas. This would increase application resiliency but would also be time-consuming. A certain amount of time would indeed be needed to copy the state of one of the remaining replicas. Because all replicas of a same process must have a coherent state, the execution of the still running replicas would have to be paused during this copying. In our tightly coupled application model, the copying-time would be a time during which the

Then, for the whole application to fail, one of the replica-group must be hit by g failures. Ferreira et al. [10] consider the case $g = 2$, and observe that the generalized birthday problem is related to the problem of determining the number of process failures needed to induce the failure of the whole application. The generalized birthday problem answers the following question: what is the lowest integer n such that, when randomly drawing n integers from a discrete uniform distribution with range $[1, m]$, there is a probability at least equal to 50% that two numbers are the same? In our scope, $m = n_{rg}$ is the number of replica-groups, and n denotes the number of failures. In [10] it is stated that the mean number of faults that can happen so that there is a 50% chance that both replicas have not failed is:

$$NF(n_{rg}) = 1 + \sum_{k=1}^{n_{rg}} \frac{n_{rg}!}{(n_{rg} - k)! \cdot n_{rg}^k} \approx \sqrt{\frac{\pi n_{rg}}{2}} + \frac{2}{3} \tag{6}$$

However, the target problem is not identical to the generalized birthday problem, and Equation 6 turns out incorrect. Consider the case $g = 2$ and the situation right after the first failure occurred. In the generalized birthday problem one assumes that all integers in the range are uniformly distributed. In our problem, the replica-group that suffered from the first failure only contains a single running replica after that failure, while all the other replica-groups still contain two running replicas. Therefore, if the probability of failures is uniformly distributed among still running processes (which is usually assumed), then the replica-group hit by the first failure has a probability to be stricken by the second failure twice smaller than the other replica-groups, because it has twice less running replicas! The following theorem gives the correct value of the mean number of failures needed for the whole application to fail:

Theorem 2. *If the failure inter-arrival times on the different processors are independent and identically distributed, then under the* Process replication *scheme, the expectation of the number of failures needed for the whole application to fail, that is the Mean Number of Failures To Interruption (MNFTI), is:*

Case $g = 2$***:*** $MNFTI = \mathbb{E}(NFTI|0)$ *where*

$$\mathbb{E}(NFTI|n_f) = \begin{cases} 1 & \text{if } n_f = n_{rg}, \\ 1 + \frac{2n_{rg} - 2n_f}{2n_{rg} - n_f} \mathbb{E}(NFTI|n_f + 1) & \text{otherwise.} \end{cases}$$

execution of the whole application must be paused. Consequently, restarting a failed replica would only be beneficial if the restarting cost were very small, when taking in consideration the frequency of failures, and the checkpoint and restart costs. The benefit of such an approach is doubtful, and we do not consider it (it was also ignored in [10]). In any case, if a scheme involving process restart were to be put in place, it is unlikely that there would beneficial cases with $g > 2$.

General case: $MNFTI = \mathbb{E}\left(NFTI|\underbrace{0,...,0}_{g-1 \text{ zeros}}\right)$ *where:*

$$\mathbb{E}\left(NFTI|n_f^{(1)},...,n_f^{(g-1)}\right) =$$

$$1 + \frac{g \cdot \left(n_{rg} - \sum_{i=1}^{g-1} n_f^{(i)}\right)}{g \cdot n_{rg} - \sum_{i=1}^{g-1} i \cdot n_f^{(i)}} \cdot \mathbb{E}\left(NFTI|n_f^{(1)}, n_f^{(2)}, ..., n_f^{(g-1)}\right)$$

$$+ \sum_{i=1}^{g-2} \frac{(g-i) \cdot n_f^{(i)}}{g \cdot n_{rg} - \sum_{i=1}^{g-1} i \cdot n_f^{(i)}} \cdot \mathbb{E}\left(NFTI|n_f^{(1)},...,n_f^{(i-1)}, n_f^{(i)}-1, n_f^{(i+1)}+1, n_f^{(i+2)},...,n_f^{(g-1)}\right) \tag{7}$$

Proof. We begin by studying the case $g = 2$ before generalizing. Let $\mathbb{E}(NFTI|n_f)$ be the expectation of the number of failures needed for the whole application to fail knowing that the application is still running and that failures have already hit n_f different replica-groups. Because each process initially has 2 replicas, this means that n_f different processes are no longer replicated, and that $n_{rg} - n_f$ are still replicated. Overall, there are $n_f + 2(n_{rg} - n_f) = 2n_{rg} - n_f$ still running processors.

The case $n_f = n_{rg}$ is the simplest: a new failure will hit an already hit replica-group and hence lead to an application failure, hence

$$\mathbb{E}\left(NFTI\,|n_{rg}\right) = 1.$$

For the general case $0 \leq n_f \leq n_{rg} - 1$, either the next failure hits a new replica-group with 2 still running replicas, or it hits a replica-group that had already been hit. The latter case leads to an application failure; in that case, after n_f failures, the expected number of failures needed for the whole application to fail is exactly one. The failure probability is uniformly distributed among the $2n_{rg} - n_f$ running processors, hence the probability that the next failure hits a new replica-group is $\frac{2n_{rg}-2n_f}{2n_{rg}-n_f}$. In this case, the expected number of failures needed for the whole application to fail is one (the considered failure) plus $\mathbb{E}\left(NFTI|n_f + 1\right)$. Altogether we have derived that:

$$\mathbb{E}\left(NFTI|n_f\right) = \frac{2n_{rg} - 2n_f}{2n_{rg} - n_f} \times \left(1 + \mathbb{E}\left(NFTI|n_f + 1\right)\right) + \frac{n_f}{2n_{rg} - n_f} \times 1.$$

Therefore,

$$\mathbb{E}\left(NFTI|n_f\right) = 1 + \frac{2n_{rg} - 2n_f}{2n_{rg} - n_f}\mathbb{E}\left(NFTI|n_f + 1\right).$$

We now consider the general case $g \geq 2$. Let $\mathbb{E}\left(NFTI|n_f^{(1)},...,n_f^{(g-1)}\right)$ be the expectation of the number of failures needed for the whole application to fail, knowing that the application is still running and that, for $i \in [1..g-1]$, there are $n_f^{(i)}$ replica-groups that have already been hit by exactly i failures. Note that a replica-group hit by i failures still contains exactly $g - i$ running

replicas. Therefore, in a system where $n_f^{(i)}$ replica-groups have been hit by exactly i failures, there are still overall exactly $g \cdot n_{rg} - \sum_{i=1}^{g-1} i \cdot n_f^{(i)}$ running replicas, $g \cdot \left(n_{rg} - \sum_{i=1}^{g-1} n_f^{(i)}\right)$ of which are in replica-groups that have not yet been hit by any failure. Now, consider the next failure to hit the system. There are three cases to consider.

1. The failure hits a replica-group that has not been hit by any failure so far. This happens with probability:

$$\frac{g \cdot \left(n_{rg} - \sum_{i=1}^{g-1} n_f^{(i)}\right)}{g \cdot n_{rg} - \sum_{i=1}^{g-1} i \cdot n_f^{(i)}}$$

   and, in that case, the expected number of failures needed for the whole application to fail is one (the studied failure) plus $\mathbb{E}\left(NFTI|1 + n_f^{(1)}, n_f^{(2)}, ..., n_f^{(g-1)}\right)$. Remark that we should have conditioned the above expectation with the statement "if $n_{rg} > \sum_{i=1}^{g-1} n_f^{(i)}$". In order to keep Equation 7 as simple as possible we rather do not explicitly state the condition and use the following abusive notation:

$$\frac{g \cdot \left(n_{rg} - \sum_{i=1}^{g-1} n_f^{(i)}\right)}{g \cdot n_{rg} - \sum_{i=1}^{g-1} i \cdot n_f^{(i)}} \cdot \left(1 + \mathbb{E}\left(NFTI|1 + n_f^{(1)}, n_f^{(2)}, ..., n_f^{(g-1)}\right)\right),$$

   considering than when $n_{rg} = \sum_{i=1}^{g-1} n_f^{(i)}$ the first term is null and thus that it does not matter that the second term is not defined.

2. The failure hits a replica-group that has already been hit by $g-1$ failures. Such a failure leads to a failure of the whole application. As there are $n_f^{(g-1)}$ such group, each containing exactly one running replica, this event happens with probability:

$$\frac{n_f^{(g-1)}}{g \cdot n_{rg} - \sum_{i=1}^{g-1} i \cdot n_f^{(i)}}.$$

   In this case, the expected number of failures needed for the whole application to fail is exactly equal to one (the considered failure).

3. The failure hits a replica-group that had already been hit by at least one failure, and by at most $g-2$ failures. Let i be any value in $[1..g-2]$. The probability that the failure hits a group that had previously been the victim of exactly i failures is equal to:

$$\frac{(g-i) \cdot n_f^{(i)}}{g \cdot n_{rg} - \sum_{i=1}^{g-1} i \cdot n_f^{(i)}}$$

   as there are $n_f^{(i)}$ such replica-groups and that each contains exactly $g - i$ still running replicas. In this case, the expected number of failures needed for the whole application to fail is one (the studied failure) plus

$\mathbb{E}\left(NFTI|n_f^{(1)},...,n_f^{(i-1)},n_f^{(i)}-1,n_f^{(i+1)}+1,n_f^{(i+2)},...,n_f^{(g-1)}\right)$ as there is one less replica-group hit by exactly i failures and one more hit by exactly $i+1$ failures.

We aggregate all the cases to obtain:

$$\begin{aligned}
&\mathbb{E}\left(NFTI|n_f^{(1)},...,n_f^{(g-1)}\right) = \\
&\frac{g\cdot\left(n_{rg}-\sum_{i=1}^{g-1}n_f^{(i)}\right)}{g\cdot n_{rg}-\sum_{i=1}^{g-1}i\cdot n_f^{(i)}}\cdot\left(1+\mathbb{E}\left(NFTI|1+n_f^{(1)},n_f^{(2)},...,n_f^{(g-1)}\right)\right)\\
&+\sum_{i=1}^{g-2}\frac{(g-i)\cdot n_f^{(i)}}{g\cdot n_{rg}-\sum_{i=1}^{g-1}i\cdot n_f^{(i)}}\\
&\qquad\cdot\left(1+\mathbb{E}\left(NFTI|n_f^{(1)},...,n_f^{(i-1)},n_f^{(i)}-1,n_f^{(i+1)}+1,n_f^{(i+2)},...,n_f^{(g-1)}\right)\right)\\
&+\frac{n_f^{(g-1)}}{g\cdot n_{rg}-\sum_{i=1}^{g-1}i\cdot n_f^{(i)}}\cdot 1
\end{aligned}$$

which can be rewritten as

$$\begin{aligned}
&\mathbb{E}\left(NFTI|n_f^{(1)},...,n_f^{(g-1)}\right) = \\
&\qquad 1+\frac{g\cdot\left(n_{rg}-\sum_{i=1}^{g-1}n_f^{(i)}\right)}{g\cdot n_{rg}-\sum_{i=1}^{g-1}i\cdot n_f^{(i)}}\cdot\mathbb{E}\left(NFTI|1+n_f^{(1)},n_f^{(2)},...,n_f^{(g-1)}\right)\\
&\qquad+\sum_{i=1}^{g-2}\frac{(g-i)\cdot n_f^{(i)}}{g\cdot n_{rg}-\sum_{i=1}^{g-1}i\cdot n_f^{(i)}}\\
&\qquad\qquad\cdot\mathbb{E}\left(NFTI|n_f^{(1)},...,n_f^{(i-1)},n_f^{(i)}-1,n_f^{(i+1)}+1,n_f^{(i+2)},...,n_f^{(g-1)}\right)
\end{aligned}$$

□

We point out that Theorem 2 does not make any assumption on the failure distribution. It only assumes that all processors are subject to independent and identically distributed failures.

5.2 Application failure distribution and mean time to interruption

In [10], for the case $g = 2$ the mean time to application interruption is computed using the formula:

$$MTTI = systemMTBF(2n_{rg}) \times NF(n_{rg})$$

where the value of $FN(n_{rg})$ is given by Equation 6. Here *systemMTBF* denotes the mean time between failures of a platform made up with $2n_{rg}$ processors. This expression assumes that the failures follow an Exponential distribution. This expression is inaccurate, even when one substitutes to *NF* the expression of *MNFTI* given by Theorem 2. The reason is the following: while $systemMTBF(2n_{rg})$ is the expectation of the date at which the first failure will

happen, this is not the expectation of the inter-arrival time of the first and second failures. Indeed, after the first failure there only remains, overall, $2n_{rg}-1$ running replicas. Therefore, the inter-arrival time of the first and second failure has an expectation of $systemMTBF(2n_{rg}-1)$.

One can in fact compute an exact expression for the application $MTTI$ when failures follow an Exponential distribution. The reasoning is similar to the proof of Theorem 2:

Theorem 3. *If the failure inter-arrival times on the different processors follow an Exponential law of parameter λ then, under the* PROCESS REPLICATION *scheme with $g=2$, the expectation of the time an application runs before failing, that is the Mean Time To application Interruption (MTTI), is: $MTTI = \mathbb{E}(TTI|0)$ where*

$$\mathbb{E}(TTI|n_f) = \begin{cases} \frac{1}{n_{rg}}\frac{1}{\lambda} & \text{if } n_f = n_{rg}, \\ \frac{1}{(2n_{rg}-n_f)}\frac{1}{\lambda} + \frac{2n_{rg}-2n_f}{2n_{rg}-n_f}\mathbb{E}(TTI|n_f+1) & \text{otherwise.} \end{cases}$$

Proof. We denote by $\mathbb{E}(TTI|n_f)$ the expectation of the time an application will run before failing, knowing that the application is still running and that failures have already hit n_f different replica-groups. Since each process initially has 2 replicas, this means that n_f different processes are no longer replicated and that $n_{rg}-n_f$ are still replicated. Overall, there are thus still $n_f + 2(n_{rg}-n_f) = 2n_{rg}-n_f$ running processors.

The case $n_f = n_{rg}$ is the simplest: a new failure will hit an already stricken replica-group and hence leads to an application failure. As there are exactly n_{rg} remaining running processors, the inter-arrival times of the n_{rg}-th and $(n_{rg}+1)$-th failures is equal to $\frac{1}{\lambda n_{rg}}$ (minimum of n_{rg} Exponential laws). Hence:

$$\mathbb{E}\left(TTI\,|n_{rg}\right) = \frac{1}{\lambda n_{rg}}.$$

For the general case, $0 \leq n_f \leq n_{rg}-1$, either the next failure hits a replica-group with still 2 running processors, or it strikes a replica-group that had already been victim of a failure. The latter case leads to an application failure; then, after n_f failures, the expected application running time before failure is equal to the inter-arrival times of the n_f-th and (n_f+1)-th failures, which is equal to $\frac{1}{(2n_{rg}-n_f)\lambda}$. The failure probability is uniformly distributed among the $2n_{rg}-n_f$ running processors, hence the probability that the next failure strikes a new replica-group is $\frac{2n_{rg}-2n_f}{2n_{rg}-n_f}$. In this case, the expected application running time before failure is equal to the inter-arrival times of the n_f-th and (n_f+1)-th failures plus $\mathbb{E}\left(TTI|n_f+1\right)$. We derive that:

$$\mathbb{E}\left(TTI|n_f\right) = \frac{2n_{rg}-2n_f}{2n_{rg}-n_f} \times \left(\frac{1}{(2n_{rg}-n_f)\lambda} + \mathbb{E}\left(TTI|n_f+1\right)\right) + \frac{n_f}{2n_{rg}-n_f} \times \frac{1}{(2n_{rg}-n_f)\lambda}.$$

Therefore,

$$\mathbb{E}\left(TTI|n_f\right) = \frac{1}{(2n_{rg}-n_f)\lambda} + \frac{2n_{rg}-2n_f}{2n_{rg}-n_f}\mathbb{E}\left(TTI|n_f+1\right).$$

□

One can generalize Theorems 2 and 3 to deal with any value of g. But this technique, based on recurrence equations, is limited to failures following an Exponential distribution. To extend the computation of the MTTI to arbitrary distributions, we use another approach, based on the failure distribution law at the platform level. We explicit in Theorem 4 the probability of successfully completing a work of size $\mathcal{W}$ under the PROCESS REPLICATION scheme, and this for any failure distribution. This theoretical result (Theorem 4) enables us to compute the MTTI for arbitrary failure distributions, using numerical integration schemes. We refine the result and provide closed-form expressions for the MTTI when failures follow an Exponential distribution (Theorem 5) or a Weibull distribution with fresh processors (Theorem 6). These closed-form expressions are directly amenable to numerical evaluation (see Section 5.4)

Theorem 4. *Consider an application with n_{rg} processes, each replicated g times under the* PROCESS REPLICATION *scheme, such that processor P_i, $1 \leq i \leq g \cdot n_{rg}$, executes a replica of process $\left\lceil \frac{i}{g} \right\rceil$. Assume that the failure inter-arrival times on the different processors are independent and identically distributed,vand let τ_i denote the time elapsed since the last failure of processor P_i. Let F denote the cumulative distribution function of the failure probability, and $F(t|\tau)$ be the probability that a processor fails in the next t units of time, knowing that its last failure happened τ units of time ago. The probability that the application will still be running after t units of time is equal to:*

$$R(t) = \prod_{j=1}^{n_{rg}} \left(1 - \prod_{i=1}^{g} F\left(t|\tau_{i+g(j-1)}\right) \right). \tag{8}$$

And the Mean Time To application Interruption is equal to:

$$MTTI = \int_0^{+\infty} \prod_{j=1}^{n_{rg}} \left(1 - \prod_{i=1}^{g} F\left(t|\tau_{i+g(j-1)}\right) \right) \mathrm{d}t. \tag{9}$$

While failure independence is necessary to prove Theorem 4, the hypothesis that the failures are identically distributed can be removed. We have added this hypothesis assumption to simplify the writing of Equations 8 and9.

Proof. The probability that processor P_i suffers from a failure during the next t units of time, knowing that the time elapsed since its last failure is τ_i, is equal by definition to $F_i(t) = F(t|\tau_i)$. Then the probability that the g processors running the replicas of process j, $1 \leq j \leq n_{rg}$, all suffer from a failure during the next t units of time is then equal to:

$$F_j^{(g)}(t) = \prod_{i=1}^{g} F_{i+g(j-1)}(t) = \prod_{i=1}^{g} F\left(t|\tau_{i+g(j-1)}\right).$$

Therefore, the probability that at least one the g duplicates of process j is still running after t units of time is equal to:

$$R_j^{(g)}(t) = 1 - F_j^{(g)}(t) = 1 - \prod_{i=1}^{g} F\left(t|\tau_{i+g(j-1)}\right).$$

For the whole application to still be running after t units of time, each of the n_{rg} application processes must still be running (i.e., each must have at least one of its g initial replicas still running). So, the probability that the application is still running after t units of time is:

$$R(t) = \prod_{j=1}^{n_{rg}} R_j^{(g)}(t) = \prod_{j=1}^{n_{rg}} \left(1 - \prod_{i=1}^{g} F\left(t|\tau_{i+g(j-1)}\right)\right).$$

We can then compute the Mean Time To Interruption of the whole application:

$$MTTI = \int_0^{+\infty} R(t)\mathrm{d}t = \int_0^{+\infty} \prod_{j=1}^{n_{rg}} \left(1 - \prod_{i=1}^{g} F\left(t|\tau_{i+g(j-1)}\right)\right) \mathrm{d}t.$$

□

We now consider the case of the Exponential law.

Theorem 5. *Consider an application made of n_{rg} processes, each replicated g times under the* Process replication *scheme. If the probability distribution of the time to failure of each processor follows an Exponential law of parameter λ, then the Mean Time To application Interruption is equal to:*

$$MTTI = \frac{1}{\lambda} \sum_{i=1}^{n_{rg}} \sum_{j=1}^{i \cdot g} \left(\frac{\binom{n_{rg}}{i}\binom{i \cdot g}{j}(-1)^{i+j}}{j} \right)$$

Proof. According to Theorem 4, the probability that the application is still running after t units of time is:

$$R(t) = \left(1 - \left(1 - e^{-\lambda t}\right)^g\right)^{n_{rg}}$$

and the Mean Time To Interruption of the whole application is:

$$\begin{aligned}
MTTI &= \int_0^{+\infty} R(t)\mathrm{d}t \\
&= \int_0^{+\infty} \left(1-\left(1-e^{-\lambda t}\right)^g\right)^{n_{rg}} \mathrm{d}t \\
&= \int_0^{+\infty} \sum_{i=0}^{n_{rg}} \binom{n_{rg}}{i} (-1)^i \left(1-e^{-\lambda t}\right)^{i\cdot g} \mathrm{d}t \\
&= \int_0^{+\infty} \sum_{i=0}^{n_{rg}} \binom{n_{rg}}{i} (-1)^i \left(\sum_{j=0}^{i\cdot g} \binom{i\cdot g}{j} (-1)^j e^{-\lambda j t}\right) \mathrm{d}t \\
&= \int_0^{+\infty} \sum_{i=0}^{n_{rg}} \binom{n_{rg}}{i} (-1)^i \left(1+\sum_{j=1}^{i\cdot g} \binom{i\cdot g}{j} (-1)^j e^{-\lambda j t}\right) \mathrm{d}t \\
&= \int_0^{+\infty} \left[\sum_{i=0}^{n_{rg}} \binom{n_{rg}}{i} (-1)^i + \sum_{i=0}^{n_{rg}} \left(\binom{n_{rg}}{i} (-1)^i \sum_{j=1}^{i\cdot g} \binom{i\cdot g}{j} (-1)^j e^{-\lambda j t}\right)\right] \mathrm{d}t \\
&= \int_0^{+\infty} \sum_{i=0}^{n_{rg}} \left[\binom{n_{rg}}{i} (-1)^i \sum_{j=1}^{i\cdot g} \binom{i\cdot g}{j} (-1)^j e^{-\lambda j t}\right] \mathrm{d}t \\
&= \sum_{i=0}^{n_{rg}} \left[\binom{n_{rg}}{i} (-1)^i \sum_{j=1}^{i\cdot g} \binom{i\cdot g}{j} (-1)^j \int_0^{+\infty} e^{-\lambda j t}\mathrm{d}t\right] \\
&= \sum_{i=0}^{n_{rg}} \left[\binom{n_{rg}}{i} (-1)^i \sum_{j=1}^{i\cdot g} \frac{\binom{i\cdot g}{j}(-1)^j}{\lambda j}\right] \\
&= \sum_{i=1}^{n_{rg}} \left[\binom{n_{rg}}{i} (-1)^i \sum_{j=1}^{i\cdot g} \frac{\binom{i\cdot g}{j}(-1)^j}{\lambda j}\right]
\end{aligned}$$

Thus,

$$MTTI = \sum_{i=1}^{n_{rg}} \sum_{j=1}^{i\cdot g} \frac{\binom{n_{rg}}{i}\binom{i\cdot g}{j}(-1)^{i+j}}{j\lambda}.$$

□

Corollary 1. *Consider an application made of n_{rg} processes, each replicated 2 times under the* Process replication *scheme. If the probability distribution of the time to failure of each processor follows an Exponential law of parameter λ, then the Mean Time To application Interruption is equal to:*

$$MTTI = \frac{1}{\lambda} \sum_{i=1}^{n_{rg}} \sum_{j=1}^{i\cdot 2} \left(\frac{\binom{n_{rg}}{i}\binom{i\cdot 2}{j}(-1)^{i+j}}{j}\right) = \frac{2^{n_{rg}}}{\lambda} \sum_{i=0}^{n_{rg}} \left(\frac{-1}{2}\right)^i \frac{\binom{n_{rg}}{i}}{(n_{rg}+i)}.$$

Proof. The first expression is a simple corollary of Theorem 5 for the case $g = 2$. The second expression is obtained through direct computation. Let $f(t)$ be the probability density function associated to the cumulative distribution function $F(t)$. Then, we have:

$$\begin{aligned}
MTTI &= \int_0^{+\infty} t \cdot f(t)\mathrm{d}t \\
&= \int_0^{+\infty} t\ 2^k k\lambda \left(1 - e^{-\lambda t}\right) e^{-\lambda k t} \left(1 - \frac{e^{-\lambda t}}{2}\right)^{k-1} \mathrm{d}t \\
&= 2^k k\lambda \int_0^{+\infty} t \left(1 - e^{-\lambda t}\right) e^{-\lambda k t} \sum_{i=0}^{k-1} \binom{k-1}{i} \left(\frac{-1}{2}\right)^i e^{-\lambda i t} \mathrm{d}t \\
&= 2^k k\lambda \sum_{i=0}^{k-1} \binom{k-1}{i} \left(\frac{-1}{2}\right)^i \int_0^{+\infty} t \left(1 - e^{-\lambda t}\right) e^{-\lambda (k+i) t} \mathrm{d}t \\
&= 2^k k\lambda \sum_{i=0}^{k-1} \binom{k-1}{i} \left(\frac{-1}{2}\right)^i \int_0^{+\infty} \left(te^{-\lambda (k+i) t} - te^{-\lambda (k+i+1) t}\right) \mathrm{d}t.
\end{aligned}$$

As $\int_0^{+\infty} te^{-\lambda t} = \frac{1}{\lambda^2}$, the expression of $MTTI$ can be further refined as follows:

$$\begin{aligned}
MTTI &= 2^k k\lambda \sum_{i=0}^{k-1} \binom{k-1}{i} \left(\frac{-1}{2}\right)^i \left(\frac{1}{(k+i)^2 \lambda^2} - \frac{1}{(k+i+1)^2 \lambda^2}\right) \\
&= \frac{2^k k}{\lambda} \sum_{i=0}^{k-1} \binom{k-1}{i} \left(\frac{-1}{2}\right)^i \left(\frac{1}{(k+i)^2} - \frac{1}{(k+i+1)^2}\right) \\
&= \frac{2^k k}{\lambda} \sum_{i=0}^{k-1} \left[\binom{k-1}{i} \left(\frac{-1}{2}\right)^i \frac{1}{(k+i)^2}\right] - \frac{2^k k}{\lambda} \sum_{i=0}^{k-1} \left[\binom{k-1}{i} \left(\frac{-1}{2}\right)^i \frac{1}{(k+i+1)^2}\right] \\
&= \frac{2^k k}{\lambda} \left(\sum_{i=0}^{k-1} \left[\binom{k-1}{i} \left(\frac{-1}{2}\right)^i \frac{1}{(k+i)^2}\right] - \sum_{I=1}^{k} \left[\binom{k-1}{I-1} \left(\frac{-1}{2}\right)^{I-1} \frac{1}{(k+I)^2}\right]\right) \\
&= \frac{2^k k}{\lambda} \left(\sum_{i=1}^{k-1} \left[\binom{k-1}{i} \left(\frac{-1}{2}\right)^i \frac{1}{(k+i)^2}\right] + \binom{k-1}{0} \left(\frac{-1}{2}\right)^0 \frac{1}{(k)^2}\right. \\
&\qquad \left. + 2\sum_{I=1}^{k-1} \left[\binom{k-1}{I-1} \left(\frac{-1}{2}\right)^I \frac{1}{(k+I)^2}\right] + 2\binom{k-1}{k-1} \left(\frac{-1}{2}\right)^k \frac{1}{(2k)^2}\right) \\
&= \frac{2^k k}{\lambda} \left(\frac{1}{k^2} + \left(\frac{-1}{2}\right)^k \frac{1}{2k^2} + \sum_{i=1}^{k-1} \left[\left(\frac{-1}{2}\right)^i \frac{1}{(k+i)^2} \left(\binom{k-1}{i} + 2\binom{k-1}{i-1}\right)\right]\right).
\end{aligned}$$

Using the equation $\binom{k-1}{i} + 2\binom{k-1}{i-1} = \binom{k}{i}\frac{(k+i)}{k}$, we derive the desired expression

for $MTTI$:

$$\begin{aligned} MTTI &= \frac{2^k k}{\lambda}\left(\frac{1}{k^2} - \left(\frac{-1}{2}\right)^{k+1}\frac{1}{k^2} + \sum_{i=1}^{k-1}\left(\frac{-1}{2}\right)^i \frac{\binom{k}{i}}{(k+i)^2}\frac{(k+i)}{k}\right) \\ &= \frac{2^k k}{\lambda}\left(\frac{1}{k^2}\left(1 - \left(\frac{-1}{2}\right)^{k+1}\right) + \sum_{i=1}^{k-1}\left(\frac{-1}{2}\right)^i \frac{\binom{k}{i}}{(k+i)\,k}\right) \\ &= \frac{2^k}{\lambda}\left(\frac{1}{k}\left(1 + \frac{1}{2}\left(\frac{-1}{2}\right)^k\right) + \sum_{i=1}^{k-1}\left(\frac{-1}{2}\right)^i \frac{\binom{k}{i}}{(k+i)}\right) \\ &= \frac{2^k}{\lambda}\sum_{i=0}^{k}\left(\frac{-1}{2}\right)^i \frac{\binom{k}{i}}{(k+i)} \end{aligned}$$

□

Theorem 6. *Consider an application made of n_{rg} processes, each replicated g times under the* Process replication *scheme. If the probability distribution of the time to failure of each processor follows a Weibull law of scale parameter λ and shape parameter k, then the Mean Time To application Interruption is equal to:*

$$\frac{\lambda}{k}\,\Gamma\left(\frac{1}{k}\right)\,\sum_{i=1}^{n_{rg}}\sum_{j=1}^{i\cdot g}\frac{\binom{n_{rg}}{i}\binom{i\cdot g}{j}(-1)^{i+j}}{j^{\frac{1}{k}}}.$$

Proof. According to Theorem 4, the probability that the application is still running after t units of time is:

$$R(t) = \left(1 - \left(1 - e^{-\left(\frac{t}{\lambda}\right)^k}\right)^g\right)^{n_{rg}}.$$

and the Mean Time To Interruption of the whole application is:

$$\begin{aligned}
MTTI &= \int_0^{+\infty} R(t)\mathrm{d}t \\
&= \int_0^{+\infty} \sum_{i=0}^{n_{rg}} \binom{n_{rg}}{i} (-1)^i \left(1 - e^{-\left(\frac{t}{\lambda}\right)^k}\right)^{i\cdot g} \mathrm{d}t \\
&= \int_0^{+\infty} \sum_{i=0}^{n_{rg}} \binom{n_{rg}}{i} (-1)^i \left(\sum_{j=0}^{i\cdot g} \binom{i\cdot g}{j} (-1)^j e^{-j\left(\frac{t}{\lambda}\right)^k}\right) \mathrm{d}t \\
&= \int_0^{+\infty} \sum_{i=0}^{n_{rg}} \binom{n_{rg}}{i} (-1)^i \left(1 + \sum_{j=1}^{i\cdot g} \binom{i\cdot g}{j} (-1)^j e^{-j\left(\frac{t}{\lambda}\right)^k}\right) \mathrm{d}t \\
&= \int_0^{+\infty} \left[\sum_{i=0}^{n_{rg}} \binom{n_{rg}}{i} (-1)^i + \sum_{i=0}^{n_{rg}} \binom{n_{rg}}{i} (-1)^i \left(\sum_{j=1}^{i\cdot g} \binom{i\cdot g}{j} (-1)^j e^{-j\left(\frac{t}{\lambda}\right)^k}\right)\right] \mathrm{d}t \\
&= \int_0^{+\infty} \sum_{i=0}^{n_{rg}} \binom{n_{rg}}{i} (-1)^i \left[\sum_{j=1}^{i\cdot g} \binom{i\cdot g}{j} (-1)^j e^{-j\left(\frac{t}{\lambda}\right)^k}\right] \mathrm{d}t \\
&= \sum_{i=1}^{n_{rg}} \binom{n_{rg}}{i} (-1)^i \left[\sum_{j=1}^{i\cdot g} \binom{i\cdot g}{j} (-1)^j \int_0^{+\infty} e^{-j\left(\frac{t}{\lambda}\right)^k} \mathrm{d}t\right]
\end{aligned}$$

We consider any value $j \in [0..n_{rg} \cdot g]$ and we make the following change of variable: $u = \frac{j}{\lambda^k} t^k$. This is equivalent to $t = \lambda \left(\frac{u}{j}\right)^{\frac{1}{k}}$ and thus $\mathrm{d}t = \frac{\lambda}{k} \left(\frac{1}{j}\right)^{\frac{1}{k}} u^{\left(\frac{1}{k}-1\right)} \mathrm{d}u$. With this notation,

$$\int_0^{+\infty} e^{-j\left(\frac{t}{\lambda}\right)^k} \mathrm{d}t = \frac{\lambda}{k j^{\frac{1}{k}}} \Gamma\left(\frac{1}{k}\right).$$

Therefore,

$$MTTI = \sum_{i=1}^{n_{rg}} \binom{n_{rg}}{i} (-1)^i \left(\sum_{j=1}^{i\cdot g} \binom{i\cdot g}{j} (-1)^j \frac{\lambda}{k j^{\frac{1}{k}}} \Gamma\left(\frac{1}{k}\right)\right).$$

Thus,

$$MTTI = \frac{\lambda}{k} \Gamma\left(\frac{1}{k}\right) \sum_{i=1}^{n_{rg}} \sum_{j=1}^{i\cdot g} \frac{\binom{n_{rg}}{i}\binom{i\cdot g}{j} (-1)^{i+j}}{j^{\frac{1}{k}}}.$$

□

5.3 Comparison of MNFTI values

Table 1 shows the MNFTI values as computed by the formula in [10] and by our recursive formula in Theorem 2, for various values of n_{rg} and for $g = 2$. The percentage relative difference is indicated as well. We see that the two values diverge significantly, with relative differences over 25% for large values of n_{rg}. We conclude that the formula in [10] significantly under-estimates the MNFTI.

Table 1: Comparison of the MNFTI (Mean Number of Failures To Interruption) as computed by the formula in [10] and by the recursive formula developed in this work, for $n_{rg} = 2^0, \ldots, 2^{20}$, with $g = 2$.

Number of processors	2^0	2^1	2^2	2^3	2^4	2^5	2^6
Formula in [10]	2	2.5	3.22	4.25	5.7	7.77	10.7
Recursive Formula (Theorem 2)	2	2.67	3.66	5.09	7.15	10.1	14.2
% Relative Diff	0	-6.2	-12	-17	-20	-23	-25
Number of processors	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}
Formula in [10]	14.9	20.7	29	40.8	57.4	80.9	114
Recursive Formula (Theorem 2)	20.1	28.4	40.1	56.7	80.2	113	160
% Relative Diff	-26	-27	-28	-28	-28	-29	-29
Number of processors	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
Formula in [10]	161	228	322	454	642	908	1284
Recursive Formula (Theorem 2)	227	321	454	642	907	1283	1815
% Relative Diff	-29	-29	-29	-29	-29	-29	-29

Table 2: Comparison of the MTTI as computed by the formula in [10] and by the recursive formula of Section 5.2, for $n_{rg} = 2^0, \ldots, 2^{20}$, with $g = 2$.

Number of processors	2^0	2^1	2^2	2^3	2^4	2^5	2^6
Formula in [10]	1	0.625	0.402	0.265	0.178	0.121	0.0836
Recursive Formula (Theorem 3)	1.5	0.917	0.582	0.381	0.255	0.173	0.119
% Relative Diff	-33.33	-31.82	-30.89	-30.32	-29.97	-29.75	-29.6
Simulated MTTI	1.498	0.9184	0.5831	0.3808	0.2542	0.1725	0.1188
Number of processors	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}
Formula in [10]	0.058	0.0405	0.0284	0.0199	0.014	0.00987	0.00696
Recursive Formula (Theorem 3)	0.0823	0.0574	0.0402	0.0282	0.0198	0.014	0.00985
% Relative Diff	-29.5	-29.44	-29.39	-29.36	-29.34	-29.33	-29.31
Simulated MTTI	0.08226	0.05738	0.0401	0.02825	0.01982	0.01399	0.009853
Number of processors	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
Formula in [10]	0.00492	0.00347	0.00245	0.00173	0.00123	0.00086	0.000612
Recursive Formula (Theorem 3)	0.00695	0.00491	0.00347	0.00245	0.00173	0.00122	0.000866
% Relative Diff	-29.31	-29.3	-29.3	-29.3	-29.29	-29.29	-29.29
Simulated MTTI	0.006929	0.004913	0.00347	0.002448	0.001732	0.001225	0.0008677

5.4 Comparison of MTTI values

Table 2 shows the MTTI values as computed by the formula in [10] and by our recursive formula, for various values of n_{rg} and for $g = 2$. The percentage relative difference of the formula of [10] from our recursive formula is indicated as well. We see that the two values diverge significantly, with relative differences between 29% and 33%. We conclude that the formula in [10] significantly underestimates the MTTI.

Of course, our recursive formula and the two formulas of Corollary 1 have the same numerical values. Furthermore, to assess the validity of these three equations, we computed the MTTI through simulations. For each studied value of n_{rg}, we randomly generated 200,000 instances of failure dates, computed the Time To application Interruption for each instance and then the mean. This *simulated MTTI*, also reported in Table 2, is in full agreement with our formulas.

Algorithm 1: DPNextFailure ($\mathcal{W}$, C, n_{rg}, g, τ_1, ..., $\tau_{g \cdot n_{rg}}$, *quantum*)

```
Function RecDPNextFailure (x, n)
begin
    if x = 0 then
        return 0
    if solution[x][n] = unknown then
        best ← 0
        δ ← (W − x · quantum) + n · C/* Time elapsed since
            beginning of execution                          */
        for i = 1 to x do
            work = first(RecDPNextFailure(x − i, n + 1))
            cur ← ∏_{j=1}^{n_rg} (1 − ∏_{i=1}^{g} F(i · quantum + C|τ_{i+g(j−1)} + δ)) ×
            (i · quantum + work)
            if cur > best then
                best ← cur;      chunksize ← i
        solution[x][n] ← (best, chunksize)
    return solution[x][n]
return RecDPNextFailure (0, 0)
```

5.5 Checkpointing policy

Theorem 4 gives the probability that the application will still be running after t units of time, knowing the history of the failures of the different processors. It is then straightforward to adapt the DPNextFailure algorithm proposed in [4] to be used in the context of the Process replication scheme. Algorithm 1 presents the resulting algorithm.

6 Group replication

In Group replication, different application instances execute on different groups of processors. But instead of having completely independent concurrent executions, groups can help each other. All groups always compute the same chunk simultaneously, and do so until one of the groups succeeds, potentially after several failed trials. Then all other groups stop executing the current chunk and recover from the checkpoint stored by the successful group. All groups then attempt to compute the next chunk. Like for Process replication, Group replication wastes resources. However, the groups co-operate to face failures.

A key difference between Process replication and Group replication is that Process replication requires a sophisticated replication-aware implementation of the MPI library so as to make Process replication transparent. Instead, Group replication can work with any MPI implementation. As far as the application is concerned, checkpointing is only slightly more complex (for implementing the aggressive use of a saved checkpoint from another group as soon as it is produced).

In addition, Process replication induces a larger increase in the number and volume of communications. Let V_{tot} be the total volume of inter-processor communications for a traditional execution. With Process replication using g replicas per replica-groups, each original communication now involves g

sources and g destinations, hence the total communication volume becomes $V_{tot} \times g^2$. Now with GROUP REPLICATION using g groups, each original communication takes place g times, hence the total communication volume increases only to $V_{tot} \times g$.

In this section we describe an execution protocol called ASAP (As Soon As Possible) for implementing GROUP REPLICATION. We then analyze its performance for Exponential failures.

6.1 The ASAP execution protocol

We start with some notations. We consider g groups, where each group has q processors, with $g \times q \leq p$. Recall that a group is available for execution if and only if all its q processors are available. As before, let $R(q)$ and $C(q)$ denote the recovery and checkpointing time for one group. Moreover, recall that in case of a failure, the downtime of a group is a random variable $X_D(q) \geq D$, whose expectation is bounded in Proposition 1. If a group encounters a first failure at time t, the group is *down* between t and $t + X_D(q)$. Finally, the total size of the work is $\mathcal{W}$, and thus the total amount of work that must be executed by each processor of each group is $\mathcal{W}(q)$, as defined in Section 3.

An execution of the ASAP algorithm can be described as k macro-steps, where macro-step j, $1 \leq j \leq k$, corresponds to all groups executing the j-th chunk of size ω_j. Note that the value of k, the total number of chunks, as well as the values of the ω_j's, the chunk sizes, are inputs to the algorithm (we always have $\sum_{j=1}^{k} \omega_j = \mathcal{W}(q)$). We discuss how to optimally choose these values for Exponential distributions in Section 6.2.

During macro-step j, each group independently attempts to execute the j-th chunk of size ω_j and then to checkpoint, restarting as soon as possible in case of failure. As soon as one of the groups succeeds, say at time t_j^{end}, all the other groups are immediately stopped, macro-step j is over, and macro-step $(j+1)$ starts (if $j < k$). Let $\Delta_j = t_j^{end} - t_{j-1}^{end}$ be the length of macro-step j, where t_0^{end} is the starting time of the algorithm. The total execution time of the ASAP algorithm is $\sum_{j=1}^{k} \Delta_j$.

The previous description hides two important things. First, before being able to start macro-step $(j+1)$, a group that has been stopped must execute a recovery, in order to restart from the checkpoint of the successful group. Second, this recovery may well start later than at time t_j^{end}, in the case where the group is down at time t_j^{end} (see group 1 in Figure 1). The only group that does not need to recover at the beginning of the next step is the group that was successful for the previous step, except during the first step where all groups can start computing right away.

In the next section, we provide an analytical evaluation of ASAP for Exponential failure laws, and show how to compute the optimal set of inputs, namely the number of macro-steps k and the values of the chunk sizes ω_j.

6.2 Exponential law

In this section, we consider the case where the failure rate of each processor obeys an Exponential law of parameter λ. For the sake of the theoretical analysis, we introduce a slightly modified version of the ASAP protocol, where each group,

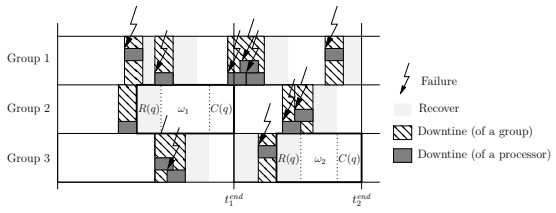

Figure 1: Execution of chunks ω_1 and ω_2 (macro-steps 1 and 2) using the ASAP protocol. At time t_1^{end}, group 1 is not ready, and group 2 is the only one that does not need to recover.

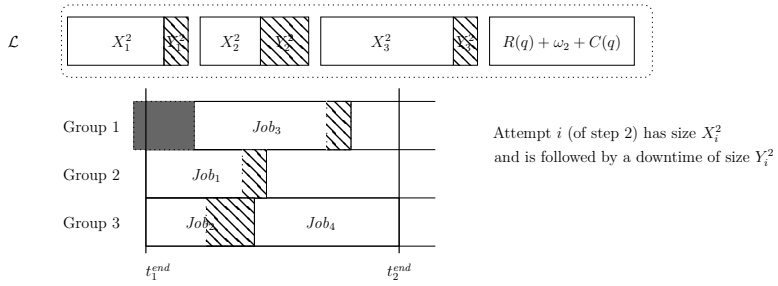

Figure 2: Zoom on macro-step 2 of the execution depicted in Figure 1, using the (X, Y) notation of Algorithm 3. Recall that Job_i has size $X_i^2 + Y_i^2$ for $1 \leq i \leq 3$, and Job_4 has size $R(q) + \omega_2 + C(q)$.

including the successful one, executes a recovery at the beginning of each macro-step. This strategy includes the first macro-step. This new version of ASAP is described in Algorithm 2. It is completely symmetric, which renders its analysis easier: now the amount of work to be executed at macro-step j is $R(q)+\omega_j+C(q)$ for all groups.

Let us now turn to the analysis of Algorithm 2. Consider the j-th macro step, number the attempts of all groups by their start time, and let N_j be the index of the earliest started attempt that succeeds to process ω_j. For example in Figure 2, the successful chunk of size $R + \omega_j + C$ is the fourth attempt, so $N_2 = 4$. Now, to represent each attempt, we sample random variables X_i^j and Y_i^j, where $1 \leq i \leq N_j$, that correspond respectively to the i^{th} tentative execution of the chunk and to the i^{th} downtime that follows it (if $i \neq N_j$). Note that $X_i^j < R + \omega_j + C$ for $i < N_j$, and $X_{N_j}^j \geq R + \omega_j + C$. All the X_i^j follow the same distribution D_X, namely an Exponential law of parameter $q\lambda$. And all the Y_i^j follow the same distribution $D_{X_D}(q)$, that of the the random variable $X_D(q)$ corresponding to the downtime of a group of q processors.

The main idea here is to view the N_j execution attempts as jobs, where the size of job i is $X_i^j + Y_i^j$, and to distribute them across the g groups in a greedy manner (see Proposition 2). The key point is that this formulation allows us to provide an upper bound for the starting time of job N_j, and hence for the length of macro-step j, using a well-known scheduling argument (see Proposition 3).

Proposition 2. *The j-th macro-step of the ASAP protocol can be simulated using Algorithm 3: the last job scheduled by Algorithm 3 ends exactly at time*

Algorithm 2: ASAP $(\omega_1, \ldots, \omega_k)$

for $j = 1$ *to* k **do**
  $todo \leftarrow R(q) + \omega_j + C(q)$
  for *each group* **do in parallel**
    repeat
      finish current downtime (if any)
      try to execute chunk of size *todo*
      if *execution successful* **then**
        signal other groups to immediately stop their attempt
        $t_j^{end} \leftarrow$ time of success
      else
        restart immediately
    until *one of the groups has a successful attempt*
$makespan \leftarrow t_k^{end}$

Algorithm 3: Step j of ASAP $(\omega_1, \ldots, \omega_k)$

$i \leftarrow 1$
`/* ` i `represents the number of attempts/jobs */`
$\mathcal{L} \leftarrow \emptyset$
`/* ` $\mathcal{L}$ `represents the list of attempts/jobs */`
sample X_i^j and Y_i^j using D_X and $D_{X_{D(q)}}$ respectively
while $X_i^j < R(q) + \omega_j + C(q)$ **do**
  add Job_i, with processing time $X_i^j + Y_i^j$, to $\mathcal{L}$
  $i \leftarrow i + 1$
  sample X_i^j and Y_i^j using D_X and $D_{X_{D(q)}}$ respectively
$N_j \leftarrow i$
add Job_{N_j}, with processing time $R(q) + \omega_j + C(q)$, to $\mathcal{L}$
`/* the first successful job has size` $R(q) + \omega_j + C(q)$`, not` $X_{N_j}^j + Y_{N_j}^j$ `*/`
from time t_{j-1}^{end} on, execute a Greedy Scheduling algorithm to distribute jobs of $\mathcal{L}$ to the different groups (recall that some groups may be not be ready at time t_{j-1}^{end})

t_j^{end}.

Proof. The Greedy Scheduling algorithm distributes the next job to the first available group. Because of the memoryless property of Exponential laws, it is equivalent (i) to generate the attempts *a priori* and greedily schedule them, or (ii) to generate them independently within each group. □

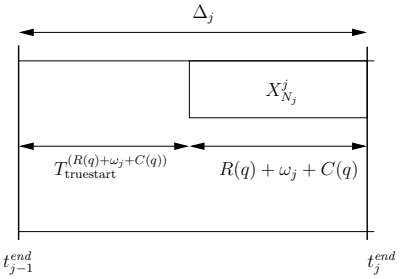

Figure 3: Notations used in Proposition 3.

Proposition 3. *Let $T^{(R(q)+\omega_j+C(q))}_{truestart}$ be the time elapsed between t^{end}_{j-1} and the beginning of Job_{N_j} (see Figure 3). We have*

$$\mathbb{E}\left(T^{(R(q)+\omega_j+C(q))}_{truestart}\right) \leq \mathbb{E}(Y) + \frac{\mathbb{E}(N_j)\mathbb{E}(X) - \mathbb{E}(X_j^{N_j}) + (\mathbb{E}(N_j)-1)\mathbb{E}(Y)}{g}$$

where X and Y are random variables corresponding to an attempt (sampled using D_X and $D_{X_D(q)}$ respectively). Moreover, we have

$$\mathbb{E}(N_j) = e^{\lambda q(R(q)+\omega_j+C(q))} \qquad \text{and} \qquad \mathbb{E}(X_j^{N_j}) = \frac{1}{q\lambda} + R(q) + \omega_j + C(q).$$

Proof. For group x, $1 \leq x \leq g$, let $\tilde{Y_x}$ denote the time elapsed before it is ready for macro-step j. For example in Figure 2, we have $\tilde{Y_1} > 0$ (group 1 is down at time t^{end}_{j-1}), while $\tilde{Y_2} = \tilde{Y_3} = 0$ (groups 2 and 3 are ready to compute at time t^{end}_{j-1}). Proposition 2 has shown that executing macro-step j can be simulated by executing a Greedy Schedule on the job list $\mathcal{L}' = \mathcal{L} \cup \bigcup_{x=1}^{g} \tilde{Y_x}$. Note that the job list $\mathcal{L}'$ may contain fewer jobs than macro-step j: the jobs that start after the successful job Job_{N_j} are discarded from the list $\mathcal{L}'$. However, both schedules have the same makespan, and jobs common to both systems have the same running dates. Thus, we have $T^{(R(q)+\omega_j+C(q))}_{\text{truestart}} \leq \frac{\sum_{x=1}^{g}(\tilde{Y_x}) + \sum_{i=1}^{N_j-1}(X_i^j + Y_i^j)}{g}$: this key inequality is due to the property of greedy scheduling: the group which is assigned the last job is the least loaded when this assignment is decided, hence its load does not exceed the average load (which is the total load divided by the number of groups). Given that $\mathbb{E}(\tilde{Y_x}) \leq \mathbb{E}(Y)$, we derive

$$\mathbb{E}\left(T^{(R(q)+\omega_j+C(q))}_{\text{truestart}}\right) \leq \mathbb{E}(Y) + \frac{\mathbb{E}\left(\sum_{i=1}^{N_j-1} X_i^j\right) + \mathbb{E}\left(\sum_{i=1}^{N_j-1}(Y_i^j)\right)}{g}$$

But N_j is the stopping criterion of the (X_i^j) sequence, hence using Wald's theorem we have $\mathbb{E}(\sum_{i=1}^{N_j} X_i^j) = \mathbb{E}(N_j)\mathbb{E}(X)$ which leads to $\mathbb{E}(\sum_{i=1}^{N_j-1} X_i^j) =$

$\mathbb{E}(N_j)\mathbb{E}(X) - \mathbb{E}(X_j^{N_j})$. Moreover, as N_j and Y_i^j are independent variables, we have $\mathbb{E}(\sum_{i=1}^{N_j-1} Y_i^j) = (\mathbb{E}(N_j)-1)\mathbb{E}(Y)$, and we get the desired bound for $\mathbb{E}(T_{\text{truestart}}^{(R(q)+\omega_j+C(q))})$.

Finally, as the expected number of attempts when repeating independently until success an event of probability α is $\frac{1}{\alpha}$ (geometric law), we get $\mathbb{E}(N_j) = e^{\lambda q(R(q)+\omega_j+C(q))}$. The value of $\mathbb{E}(X_j^{N_j})$ can be directly computed from the definition, recalling that $X_j^{N_j} \geq R(q)+\omega_j+C(q)$ and each X_j^i follows an Exponential distribution of parameter $q\lambda$. □

Building on Proposition 3, we derive the following upper bound on the execution time of ASAP:

Theorem 7. *The expected execution time of ASAP has the following upper bound:*

$$\frac{g-1}{g}\mathcal{W}(q) + \frac{1}{g}\left(\frac{1}{q\lambda}+\mathbb{E}(Y)\right)e^{\lambda q(R(q)+C(q))}k^* e^{\lambda q\frac{\mathcal{W}(q)}{k^*}} + k^*\left(\frac{g-1}{g}\left(\mathbb{E}(Y)+R(q)+C(q)\right) - \frac{1}{g}\frac{1}{q\lambda}\right)$$

which is obtained when using $k^* = \max(1, \lfloor k_0 \rfloor)$ or $k^* = \lceil k_0 \rceil$ same-size chunks, whichever leads to the smaller value, where

$$k_0 = \frac{\lambda q \mathcal{W}(q)}{1+\mathbb{L}\left(\left(g-1+\frac{(g-1)q\lambda(R(q)+C(q))-g}{1+q\lambda\mathbb{E}(Y)}\right)e^{-(1+\lambda q(R(q)+C(q)))}\right)}.$$

Here $\mathbb{L}$, the Lambert function, is defined as $\mathbb{L}(z)e^{\mathbb{L}(z)} = z$.

Proof. From Proposition 3, the expected execution time of ASAP has upper bound $T_{ASAP} = \sum_{j=1}^{k}\alpha_j$, where

$$\alpha_j = \mathbb{E}(Y) + \frac{\mathbb{E}(N_j)\mathbb{E}(X) - \mathbb{E}(X_j^{N_j}) + (\mathbb{E}(N_j)-1)\mathbb{E}(Y)}{g} + (R(q)+\omega_j+C(q)).$$

Our objective now is to find the inputs to the ASAP algorithm, namely the number k of macro-steps together with the chunk sizes $(\omega_1,\ldots,\omega_k)$, that minimize this T_{ASAP} bound.

We first have to prove that any optimal (in expectation) policy uses only a finite number of chunks. Let α be the expectation of the ASAP makespan using a unique chunk of size $\mathcal{W}(q)$. According to Proposition 3, $\alpha = \mathbb{E}(T_{\text{truestart}}^{(R(q)+\mathcal{W}(q)+C(q))}) + C(q)+\mathcal{W}(q)+R(q)$, and is finite. Thus, if an optimal policy uses k^* chunks, we must have $k^*C(q) \leq \alpha$, and thus k^* is bounded.

In the proof of Theorem 1 in [4], we have shown that any deterministic strategy uses the same sequence of chunk sizes, whatever the failure scenario, thanks to the memoryless property of the exponential distribution. We cannot prove such a result in the current context. For instance, the number of groups performing a downtime at time t_1^{end} depends on the scenario. There is thus no reason a priori for the size of the second chunk to be independent of the scenario. To overcome this difficulty, we restrict our analysis to strategies that

use the same sequence of chunk sizes whatever the failure scenario. We optimize T_{ASAP} in that context, at the possible cost of finding a larger upper bound.

We thus suppose that we have a fixed number of chunks, k, and a sequence of chunk sizes $(\omega_1, \ldots, \omega_k)$, and we look for the values of $(\omega_1, \ldots, \omega_k)$ that minimize $T_{ASAP} = \sum_{j=1}^{k} \alpha_j$. Let us first compute one of the α_j term. Replacing $\mathbb{E}(N_j)$ and $\mathbb{E}(X_j^{N_j})$ by the values given in Proposition 3, and $\mathbb{E}(X)$ by $\frac{1}{q\lambda}$, we get

$$\alpha_j = \frac{g-1}{g}\omega_j + \frac{1}{g}e^{\lambda q(R(q)+\omega_j+C(q))}\left(\frac{1}{q\lambda} + \mathbb{E}(Y)\right) + \frac{g-1}{g}\left(\mathbb{E}(Y) + R(q) + C(q)\right) - \frac{1}{g}\frac{1}{q\lambda}$$

$$T_{ASAP} = \frac{g-1}{g}\mathcal{W} + \frac{1}{g}\left(\frac{1}{q\lambda} + \mathbb{E}(Y)\right)e^{\lambda q(R(q)+C(q))}\sum_{j=1}^{k} e^{\lambda q\omega_j} + k\left(\frac{g-1}{g}\left(\mathbb{E}(Y) + R(q) + C(q)\right) - \frac{1}{g}\frac{1}{q\lambda}\right)$$

By convexity, the expression $\sum_{j=1}^{k} e^{\lambda q\omega_j}$ is minimal when all ω_j's are equal (to $\mathcal{W}(q)/k$). Hence all the chunks should be equal for T_{ASAP} to be minimal. We obtain:

$$T_{ASAP} = \frac{g-1}{g}\mathcal{W} + \frac{1}{g}\left(\frac{1}{q\lambda} + \mathbb{E}(Y)\right)e^{\lambda q(R(q)+C(q))}ke^{\lambda q\frac{\mathcal{W}(q)}{k}} + k\left(\frac{g-1}{g}\left(\mathbb{E}(Y) + R(q) + C(q)\right) - \frac{1}{g}\frac{1}{q\lambda}\right).$$

Let $f(x) = \tau_1 x e^{\lambda q\frac{\mathcal{W}(q)}{x}} + \tau_2 x$, where

$$\tau_1 = \frac{1}{g}\left(\frac{1}{q\lambda} + \mathbb{E}(Y)\right)e^{\lambda q(R(q)+C(q))}$$

and

$$\tau_2 = \left(\frac{g-1}{g}\left(\mathbb{E}(Y) + R(q) + C(q)\right) - \frac{1}{g}\frac{1}{q\lambda}\right).$$

A simple analysis using differentiation shows that f has a unique minimum, and solving $f'(x) = 0$ leads to $\tau_1 e^{\lambda q\frac{\mathcal{W}(q)}{k}}\left(1 - \frac{\lambda q\mathcal{W}(q)}{k}\right) + \tau_2 = 0$, and thus to $k = \frac{\lambda q\mathcal{W}(q)}{1+\mathbb{L}\left(\frac{\tau_2}{\tau_1\cdot e}\right)} = k^*$, which concludes the proof. □

Using the upper-bound of $\mathbb{E}(Y) = \mathbb{E}(X_D(q))$ provided in Proposition 1, we can compute numerically the number of chunks and the expectation of the upper bound of *ASAP*'s makespan given by Theorem 7 .

7 Conclusion

In this paper we have presented a rigorous study of replication techniques for large-scale platforms. These platforms are subject to failures, the frequencies

of which increase dramatically with platform scale. For a variety of job types (embarrassingly parallel, generic or numerical) and checkpoint cost models (constant or proportional overhead), we show that using the largest possible number of processors does not always lead to the smallest execution time. This is because using more resources implies facing more failures during execution, hence wasting more time tolerating them (via an increase in checkpointing frequency) and recovering from them. This waste results in a slow-down despite the additional hardware resources.

This observation leads us to investigate replication as a technique to better use all the resources provided by the platform. Replication comes in two flavors, Group replication and Process replication. Group replication consists in partitioning the platform into several groups, which each executes an instance of the application concurrently in phases. All groups synchronize as soon as one of them completes a phase. Instead, Process replication replicates each application process onto several processors (a replica-group), thereby reducing the need to recover from a failure only when all processors in a replica-group have failed. Process replication is the approach followed in [10] with two processors per replica-group.

While both replication techniques improve reliability, they have very different characteristics. Group replication can be used for any kind of parallel application, while Process replication is much more intrusive than Group replication, in that it requires a sophisticated replication-aware implementation of the MPI library. Also, the total communication volume is increased by a factor proportional to the square of the replication degree, while the increase is only linear for Group replication.

We have provided a thorough analysis of Process replication, providing recursive formulas for the MNFTI and MTTI, analytical expressions for arbitrary distributions, and closed-form expressions for Exponential and Weibull distributions. We have explained why the MNFTI and MTTI values determined in [10] are not accurate, leading to a different of roughly 30% with our own calulations, which are validated via simulation experiments.

We also have provided a detailed analysis of Group replication for Exponential failures, owing to an analogy with a Greedy Schedule to bound the number of attempts and the execution time of each group. We have derived the optimal number of chunks, together with their sizes. We do not have a closed-form formula because we do not know the expectation of the downtime of a processor group, but we have provided lower and upper bounds.

Ongoing work is devoted to conducting an extensive set of simulations for Exponential, Weibull and trace-based failures. We use a realistic set of failure rates and checkpoint/recovery overheads, and we explore all the combinations of job types and checkpoint cost models that we have presented in this report. Preliminary results confirm that both Group replication and Process replication do reduce total execution time for a wide range of typical exascale parameters. Within a few weeks, we expect to be able to produce an extended version of this report with comprehensive simulations.

References

[1] G. Amdahl. The validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485. AFIPS Press, 1967.

[2] L. Bautista Gomez, A. Nukada, N. Maruyama, F. Cappello, and S. Matsuoka. Transparent low-overhead checkpoint for GPU-accelerated clusters. `https://wiki.ncsa.illinois.edu/download/attachments/17630761/INRIA-UIUC-WS4-lbautista.pdf?version=1&modificationDate=1290470402000`.

[3] 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.

[4] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien. Checkpointing strategies for parallel jobs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 33:1–33:11, New York, NY, USA, 2011. ACM.

[5] M.-S. Bouguerra, T. Gautier, D. Trystram, and J.-M. Vincent. A flexible checkpoint/restart model in distributed systems. In *PPAM*, volume 6067 of *LNCS*, pages 206–215, 2010.

[6] V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan, and W. P. Zeggert. Proactive management of software aging. *IBM J. Res. Dev.*, 45(2):311–332, 2001.

[7] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, 2004.

[8] 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.*, 23(4):309–322, 2009.

[9] C. Engelmann, H. H. Ong, and S. L. Scorr. The case for modular redundancy in large-scale highh performance computing systems. In *Proc. of the 8th IASTED Infernational Conference on Parallel and Distributed Computing and Networks (PDCN)*, pages 189–194, 2009.

[10] K. Ferreira, J. Stearley, J. H. Laros, III, 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 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11. ACM, 2011.

[11] F. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1), 1999.

[12] T. Heath, R. P. Martin, and T. D. Nguyen. Improving cluster availability using workstation validation. *SIGMETRICS Perf. Eval. Rev.*, 30(1):217–227, 2002.

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

[14] N. Kolettis and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *FTCS '95*, page 381, Washington, DC, USA, 1995. IEEE CS.

[15] D. Kondo, A. Chien, and H. Casanova. Scheduling Task Parallel Applications for Rapid Application Turnaround on Enterprise Desktop Grids. *Journal of Grid Computing*, 5(4):379–405, 2007.

[16] E. Meneses. Clustering Parallel Applications to Enhance Message Logging Protocols. `https://wiki.ncsa.illinois.edu/download/attachments/17630761/INRIA-UIUC-WS4-emenese.pdf?version=1&modificationDate=1290466786000`.

[17] V. Sarkar and others. Exascale software study: Software challenges in extreme scale systems, 2009. White paper available at: `http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/ECSS%20report%20101909.pdf`.

[18] B. Schroeder and G. Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78(1), 2007.

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

[20] K. Venkatesh. Analysis of Dependencies of Checkpoint Cost and Checkpoint Interval of Fault Tolerant MPI Applications. *Analysis*, 2(08):2690–2697, 2010.

[21] L. Wang, P. Karthik, Z. Kalbarczyk, R. Iyer, L. Votta, C. Vick, and A. Wood. Modeling Coordinated Checkpointing for Large-Scale Supercomputers. In *Proc. of the International Conference on Dependable Systems and Networks*, pages 812–821, June 2005.

[22] 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.

[23] J. W. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, 1974.

[24] Z. Zheng and Z. Lan. Reliability-aware scalability models for high performance computing. In *Proc. of the IEEE Conference on Cluster Computing*, 2009.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399