Using group replication for resilience on exascale systems

Marin Bougeret[1], Henri Casanova[2], Yves Robert[3,4], Frédéric Vivien[5,3] and Dounia Zaidouni[5,3]
1. LIRMM Montpellier, France, `Marin.Bougeret@lirmm.fr`
2. Univ. of Hawai'i at Manoa, Honolulu, USA, `henric@hawaii.edu`
3. Ecole Normale Supérieure de Lyon, France
`{Yves.Robert|Frederic.Vivien|Dounia.Zaidouni}@ens-lyon.fr`
4. University of Tennessee Knoxville, USA
5. INRIA, France

Abstract

High performance computing applications must be resilient to faults. The traditional fault-tolerance solution is checkpoint-recovery, by which application state is saved to and recovered from secondary storage throughout execution. It has been shown that, even when using an optimal checkpointing strategy, the checkpointing overhead precludes high parallel efficiency at large scale. Additional fault-tolerance mechanisms must thus be used. Such a mechanism is replication, i.e., multiple processors performing the same computation so that a processor failure does not necessarily imply an application failure. In spite of resource waste, replication can lead to higher parallel efficiency when compared to using only checkpoint-recovery at large scale.

We propose to execute and checkpoint multiple application instances concurrently, an approach we term *group replication*. For Exponential failures we give an upper bound on the expected application execution time. This bound corresponds to a particular checkpointing period that we derive. For general failures, we propose a Dynamic Programming algorithm to determine non-periodic checkpoint dates as well as an empirical periodic checkpointing solution whose period is found via a numerical search. Using simulation we evaluate our proposed approaches, including comparison to the non-replication case, for both Exponential and Weibull failure distributions. Our broad finding is that group replication is useful in a range of realistic application and checkpointing overhead scenarios for future exascale platforms.

Keywords: checkpointing; replication; exascale platforms; resilience.

Corresponding author
Frédéric.Vivien
Laboratoire LIP, ENS Lyon, 69364 Lyon Cedex 07
Phone +33472728586, Fax +33472728080
E-mail: `Frederic.Vivien@inria.fr`

1 Introduction

As plans are made for deploying post-petascale high performance computing (HPC) systems [1, 2], solutions need to be developed to ensure resilience to failures. Failures occur because not all faults are automatically detected and corrected in the hardware components used to build production systems. For instance, the 224,162-core Jaguar platform experienced on the order of 1 failure per day [3]. On this platform, failures were thus common rather than exceptional for applications that enroll large numbers of processors. One recovers from a failure by resuming execution from a previously saved fault-free execution state, or *checkpoint*. Checkpoints are saved to resilient storage throughout execution, usually periodically. More frequent checkpoints lead to less loss when a failure occurs but to higher overhead during fault-free execution. A large literature is devoted to developing checkpointing strategies that minimize expected job execution time, or *makespan*, including both theoretical and practical contributions [4, 5, 6, 7, 8, 9].

In spite of these efforts, the necessary checkpoint frequency for tolerating failures in large-scale platforms can become so large that processors spend more time checkpointing than computing. Consider an ideal moldable parallel application that can be executed on an arbitrary number of processors and that is perfectly parallel. The makespan with p processors is the sequential makespan divided by p. In a failure-free execution, the larger p the faster the execution. But in the presence of failures, as p increases so does the frequency of processor failures, leading to (i) more time spent in recovering from these failures and (ii) more time spent in more frequent checkpoints to allow for an efficient recovery after each failure. Beyond some threshold values, increasing p actually increases the expected makespan [10, 11, 12, 13]. This is because the MTBF (Mean Time Between Failures) of the platform becomes so small that the application experiences too many failures, hence too many recovery and re-execution delays, to progress efficiently.

One possible solution to this problem is to increase the reliability of individual components, e.g., with more hardware redundancy. But this increase comes at a higher cost. Since system acquisition costs are typically constrained when designing a parallel platform, vendors must instead use commercial-of-the-shelf (COTS) components. The reliability of these COTS components is defined by the product lifetime, as driven by the market. HPC systems with COTS components will thus experience higher failure rates at higher scales [14], thereby limiting parallel efficiency if only checkpoint-recovery is used at these scales. Furthermore, even if the MTBF of an individual component is a high μ_{ind}, then the MTBF of a platform with p components is $\mu = \frac{\mu_{ind}}{p}$. No matter how reliable the individual components, there is thus a value of p above which errors are so frequent that they can prevent any application progress.

An age-old fault-tolerance mechanism is *replication*, by which several processors perform the same computation synchronously so that a fault on one of these processors does not lead to an application failure. Because it leads to resource waste, replication has gained traction in the HPC context only relatively recently [15, 16, 17, 13]. The authors in [13] propose "process replication" by which each process in a parallel MPI application is replicated on multiple physical processors while maintaining synchronous execution of the replicas. This approach is effective because the Mean Time Between Failures of a set of replicas (which is the average delay for failures to strike *both* processors in the replica set) becomes much larger than the MTBF of a single processor, even when only two replicas are used.

Process replication may not always be a feasible option. Process replication features must be provided transparently as part of the MPI implementation, which is not the case for the most widely used MPI implementations today. However, the work in [13] is a convincing proof of concept and shows that process replication can provide at least a partial answer to the fault-tolerance challenge for upcoming large-scale platforms. Therefore, it is reasonable to expect that production

MPI implementations will provide process replication in the future. Another reason why process replication may not be usable is that not all parallel applications are implemented using MPI or MPI-like frameworks, but can instead be based on other parallel programming models and accompanying runtime systems (e.g., concurrent objects, distributed components, workflows, algorithmic skeletons). Most existing such systems do not provide an equivalent to transparent process replication for the purpose of fault-tolerance, and enhancing them with this capability may be non trivial. When transparent replication is not (yet) provided by the runtime system, one solution could be to implement it explicitly within the application, but this is a labor-intensive process especially for legacy applications.

In this work we propose and study a technique that we call *group replication* and that can be used whenever process replication is not available. This approach is agnostic to the parallel programming model, and thus views the application as an unmodified black box. The only requirement is that the application be moldable and startable from a saved checkpoint file. Group replication consists in executing multiple application instances concurrently. For exampe, 2 distinct n-process application instances could be executed on a $2n$-processor platform. We note that (process or group) replication prevents the execution of an application that requires the aggregate memory of the full platform, and in this sense limits the scale of the application execution. However, such full-scale execution is likely impractical in the first place due to the need for a high checkpointing frequency. The processors would spend more time saving state than computing state, thus leading to low parallel efficiency.

At first glance, it may seem paradoxical that better performance can be achieved by using (process or group) replication. After all in the above example, 50% of the platform is "wasted" to perform redundant computation. The key point here is that each application instance runs at a smaller scale. As a result each instance can use lower checkpointing frequency, and can thus have better parallel efficiency when compared to a single application instance running at full scale. The application makespan can then be comparable to or even shorter than that obtained when running a single application instance. In the end, the cost of wasting processor power for redundant computation can be offset by the benefit of the reduced checkpointing frequency. Furthermore, in group replication, once an instance saves a checkpoint, the other instance can use this checkpoint immediately to "jump ahead" in its execution. Hence group replication is more efficient than the mere independent execution of several instances: each time one instance successfully completes a given "chunk of work", all the other instances immediately benefit from this success.

To implement group replication the runtime system needs to perform the typical operations needed for system-level checkpointing: determining checkpointing frequencies for each application instance, causing checkpoints to be saved, detecting application failures, and restarting an application instance from a saved checkpoint after a failure. The only additional feature is that the system must be able to stop an instance and cause it to resume execution from a checkpoint file produced by another instance as soon as it is produced.

Our contributions in this work are as follows:

- We propose group replication, an approach that can be implemented in practice with simple enhancements to a checkpointing infrastructure and that is applicable to blackbox applications regardless of the parallel programming model used. It can thus serve as a useful alternative when process replication is not feasible.
- For exponentially distributed failures we derive a checkpointing period that minimizes a upper bound on application makespan.
- For non-exponentially distributed failures we propose a Dynamic Programming approach that computes non-periodic checkpoint dates in a view to minimizing makespan.
- For non-exponentially distributed failures we also propose a periodic checkpointing approach

in which the period is computed based on a numerical search.

- We compare all our approaches in simulation, both for Exponential and Weibull failure distributions, and compare them to the no-replication case.
- Our results demonstrate that group replication can be beneficial at large scale. Importantly, this conclusion holds for Weibull failures that are aknowledged as representative of real-world failure behaviors [18, 19, 20, 21].

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 defines our theoretical framework. Section 4 describes our group replication approach. Section 5 gives our theoretical analysis assuming Exponential failures. Section 6 presents our Dynamic Programming and our periodic checkpointing approaches in the case of general failures. Our simulation methodology is detailed in Section 7. All results are presented and discussed in Section 8. Section 9 concludes with a summary of our findings and future perspectives.

2 Related work

Checkpointing policies have been widely studied in the literature. In [4], Daly studies periodic checkpointing policies for exponentially distributed failures, generalizing the well-known bound obtained by Young [5]. Daly extended his work in [6] to study the impact of sub-optimal checkpointing periods. In [7], the authors develop an "optimal" checkpointing policy, based on the popular assumption that optimal checkpointing must be periodic. In [8], Bouguerra et al. *prove* that the optimal checkpointing policy is periodic for either Exponential or Weibull failures, but with the assumption that *all* processors begin a new lifetime after each recovery and each checkpoint. We extended their results in [9], where we dropped this assumption and developed optimal solutions for Exponential failures and Dynamic Programming solutions for general failures, which have been shown effective for Weibull failures. The Weibull distribution is of interest because recognized as a reasonable approximation of failures in real-world systems [18, 19, 20, 21]. Some of the results in this work build on our work in [9].

In spite of all the above advances, the scalability of pure checkpoint-recovery approaches is limited [10, 11, 12]. Replication has long been used as a fault-tolerance mechanism in distributed systems [22] and more recently in the context of volunteer computing [23] or, together with checkpoint-recovery, in the context of grid computing [24]. Even though it induces resource waste, because of the scalability limitations of pure checkpoint-recovery replication has recently received more attention in the HPC literature [15, 16, 17, 13].

In this work we study group replication, by which multiple application instances are executed on different groups of processors. Ferreira et al. [13] recently studied an orthogonal approach, process replication, where each process of an MPI application is transparently replicated. Group replication cannot hope to outperform process replication, simply because process replication leads to dramatically increased MTBF for each replica set. However, the advantage of group replication is that it is agnostic to the parallel programming model and runtime system, at the cost of only minor modifications to the checkpointing infrastructure. Consequently, it is a useful alternative when process replication is not (yet) available to an application, as discussed in Section 1.

3 Framework

We consider the execution of a parallel application, or *job*, on a platform with p processors. We use the term processor to indicate any individually scheduled compute resource (a core, a multicore processor, a cluster node). We assume that system-level coordinated checkpoint-recovery is enabled [25].

The job consists of $\mathcal{W}$ units of (divisible) work, which can be split arbitrarily into *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 processors, we consider three models:

- Perfectly parallel jobs: $\mathcal{W}(q) = \mathcal{W}/q$.
- Generic parallel jobs: $\mathcal{W}(q) = (1-\gamma)\mathcal{W}/q + \gamma\mathcal{W}$. As in Amdahl's law [26], $\gamma < 1$ 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 [27], $q = r^2$ 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 checkpoint. 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 [28, 29]) or for the logical 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: $C(q) = R(q) = \alpha V/q = C/q$ with α some constant, for cases where the bandwidth of the network card/link at each processor is the I/O bottleneck.
- Constant overhead: $C(q) = R(q) = \alpha V = C$ with α some constant, for cases where the bandwidth to/from the resilient storage system is the I/O bottleneck.

We assume that failures can happen during recovery or checkpointing. 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 (i.i.d.).

4 Group replication execution protocol

Group replication consists in executing multiple application instances on different processor groups. All groups compute the same chunk simultaneously, and do so until one of them succeeds, potentially after several failed trials. Then all other groups stop executing that chunk and recover from the checkpoint stored by the successful group. All groups then attempt to compute the next chunk. Group replication can be implemented easily with no modification to the application, provided that the recovery implementation allows a group to recover immediately from a checkpoint produced by another group. Hereafter we formalize group replication as an execution protocol we call ASAP (As Soon As Possible).

We consider g groups, where each group has q processors, with $g \times q \leq p$. A group is available for execution if and only if all its q processors are available. In case of a failure at a processor in a group, the downtime of this group is a random variable $X_D(q) \geq D$. This random variable can take values strictly larger than D because while a processor in a group is experiencing a downtime, another processor in that group can experience a failure, thus prolonging the groups' downtime beyond D seconds. If a group encounters a first processor failure at time t, we say that the group

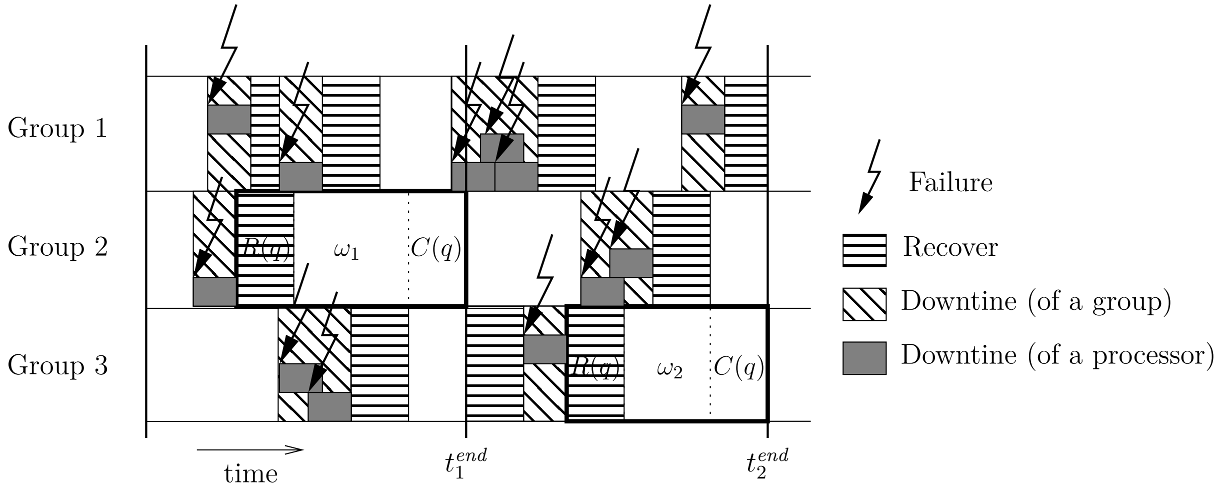


Figure 1: Execution of chunks ω_1 and ω_2 (macro-steps 1 and 2) using the ASAP protocol. At time t_1^{end}, Group 1 is not ready, and Group 2 is the only one that does not need to recover.

is *down* between times t and $t + X_D(q)$.

ASAP proceeds in k macro-steps, with a chunk of work processed during each macro-step. More formally, during macro-step j, $1 \leq j \leq k$, each group independently attempts to execute the j-th chunk of size ω_j and to checkpoint, restarting as soon as possible in case of a 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 $j < k$). The only two necessary inputs to the algorithm are (i) the number of chunks, k, and (ii) all chunk sizes, the ω_j's, chosen so that $\sum_{j=1}^{k} \omega_j = \mathcal{W}(q)$.

Before being able to start macro-step $(j+1)$, a group that has been stopped must execute a recovery so that it can resume execution from the checkpoint saved by a successful group. Furthermore, this recovery may start later than time t_j^{end}, in the case where the group is down at time t_j^{end}. This is shown on an example execution in Figure 1. At time t_1^{end}, Group 2 completes the computation and checkpointing of the chunk for macro-step 1. During that macro-step, Group 1 experiences two downtimes, each of duration D, while Group 3 experiences a single downtime of duration $> D$ due to a failure at a first processor followed by a failure at a second processor before the end of the first processor's downtime. At time t_1^{end}, Group 1 is down (experiencing a downtime caused by a sequence of three processor failures), so it cannot begin the recovery from the checkpoint saved by Group 2 immediately. Group 3, instead, can begin the recovery immediately a time t_1^{end}, but due to a failure it must reattempt the recovery. At time t_2^{end} it is Group 3 that completes the chunk for macro-step 2. As seen in the figure, the only groups that do not need to recover at the beginning of the next macro-step are the groups that were successful for the previous macro-step (except for the first macro-step for which all groups can start computing right away).

5 Exponential failures

In this section we provide an analytical evaluation of ASAP assuming Exponential failures. More specifically, we are able to compute the optimal number of macro-steps k and the optimal values of the chunk sizes ω_j. Assume that individual processor failures are distributed following an Exponential distribution of parameter λ. For the sake of the theoretical analysis, we introduce a slightly modified version of the ASAP protocol in which all groups, including the successful ones, execute

Algorithm 1: ASAP $(\omega_1, \ldots, \omega_k)$

```
for $j = 1$ to $k$ do
    for each group do in parallel
        repeat
            Finish current downtime (if any)
            Try to perform a recovery, then a chunk of size $\omega_j$, and finally to checkpoint
            if execution successful then
                Signal other groups to immediately stop their attempts
        until one of the groups has a successful attempt
```

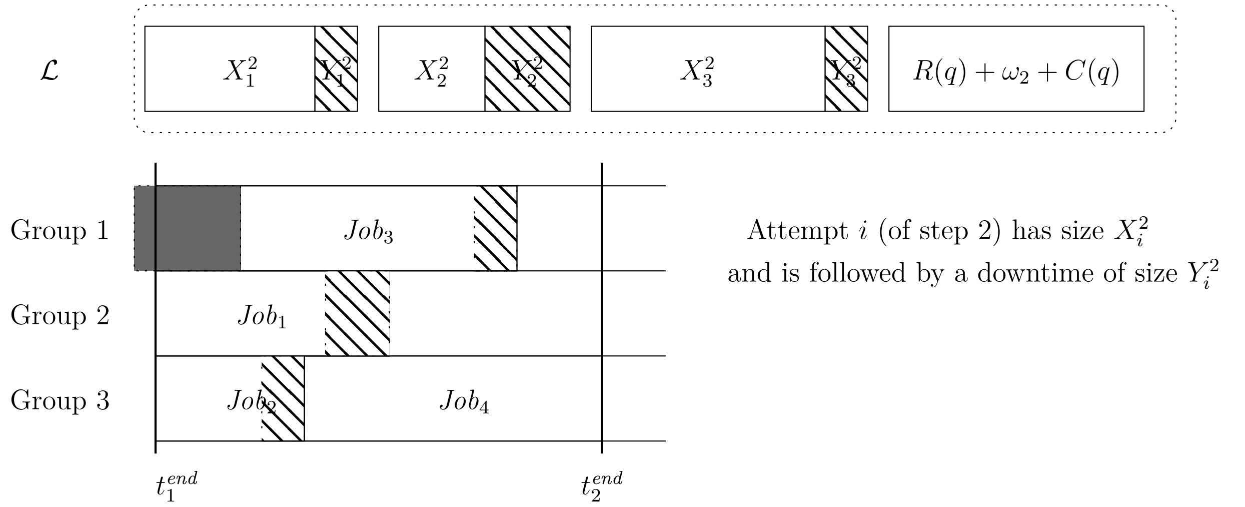


Figure 2: Zoom on macro-step 2 of the execution depicted in Figure 1, using the (X, Y) notation of Algorithm 2. Recall that Job_i has size $X_i^2 + Y_i^2$ for $1 \leq i \leq 3$, and Job_4 has size $R(q) + \omega_2 + C(q)$.

a recovery at the beginning of all macro-steps, including the first one. This version of ASAP is described in Algorithm 1. It is completely symmetric, which renders its analysis easier: for macro-step j to be successful, one of the groups must be up and running for a duration of $R(q)+\omega_j+C(q)$. Note however that all experiments reported in Section 8 use the original version of ASAP, without any superfluous recovery during execution (as depicted in Figure 1).

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 successfully computes chunk ω_j. Figure 2 zooms in on the execution of the second macro-step ($j = 2$). Each attempt is called Job_i in the order of its start time, and is followed by a downtime but for the last attempt, which is successful. In that example the successful computation of the chunk of size $R+\omega_2+C$ is the fourth attempt, Job_4, executed by Group 3. Consequently, $N_2 = 4$, meaning that macro-step 2 requires 4 attempts. The duration of each attempt is the sum of a sample of two random variables X_i^j and Y_i^j, $1 \leq i \leq N_j$. X_i^j corresponds to the duration of the i^{th} attempt at executing the chunk. Y_i^j corresponds to the duration of the i^{th} downtime that follows the i^{th} attempt (if $i \neq N_j$). Note that $X_i^j < R(q) + \omega_j + C(q)$ for $i < N_j$, and $X_{N_j}^j = R(q) + \omega_j + C(q)$. All the X_i^j's follow the same distribution D_X, an Exponential distribution of parameter $q\lambda$. And all the Y_i^j's follow the same distribution $D_{X_D}(q)$, that of the random variable $X_D(q)$ corresponding to the downtime of a group of q processors. The main idea is to view the N_j execution attempts as jobs, where the size of job

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

```
$i \leftarrow 1$                                            /* number of attempts for the job */
$\mathcal{L} \leftarrow \emptyset$                                             /* list of attempts for the job */
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 List Scheduling algorithm to distribute jobs in $\mathcal{L}$ to the different groups
(recall that some groups may not be ready at time $t_{j-1}^{end}$)
```

i is $X_i^j + Y_i^j$, and to distribute them across the g groups using the classical online *list scheduling* algorithm for independent jobs [30, Section 5.6], as stated in the following proposition:

Proposition 1. *The j-th ASAP macro-step can be simulated using Algorithm 2: the last job scheduled by Algorithm 2 ends exactly at time t_j^{end}.*

This formulation makes it possible to derive the following theorem that gives an upper bound for the expected makespan of ASAP.

Theorem 1. *The expected makespan of ASAP has the following upper bound:* $\frac{g-1}{g}\mathcal{W}(q) + \frac{1}{g}\left(\frac{1}{q\lambda} + \mathbb{E}(Y)\right) e^{\lambda q(R(q)+C(q))} k^* e^{\lambda q \frac{\mathcal{W}(q)}{k^*}} + k^*\left(\frac{g-1}{g}\left(\mathbb{E}(Y) + R(q) + C(q)\right) - \frac{1}{g}\frac{1}{q\lambda}\right)$, *where Y is a random variable with distribution $D_{X_D(q)}$. This bound 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*

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

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

This theorem can in turn be used to compute numerically the number of chunks and an upper bound on the expected makespan, provided that $\mathbb{E}(Y) = \mathbb{E}(X_D(q))$ can be itself bounded. The following proposition provides such a bound:

Proposition 2. *Let $X_D(q)$ denote the downtime of a group of q processors. Then*

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

All proofs are available in a technical report [31].

6 General failures

The analytical derivations in Section 5 hold only for Exponential failures. In the case of non-Exponential failures we propose two algorithms for determining an execution of ASAP that achieves good makespan in practice: a "brute-force" approach called BestPeriod and a Dynamic Programming approach called DPNextFailure.

6.1 Brute-force algorithm

The BestPeriod algorithm enforces a periodic execution of ASAP, meaning that all chunk sizes are identical. For a given number of groups, the period is computed via a numerical search among a set of candidate periods generated as follow. The work in [9] makes it possible to compute an optimal period, τ, for an application executed without replication on n processors subjected to Exponential failures. In our case, with g groups and p processors, we compute this period for $n = \lfloor p/g \rfloor$ processors. Besides τ, we then generate 360 candidates as $\tau(1 + 0.05 \times i)$ and $\tau/(1 + 0.05 \times i)$ for $i \in \{1, ..., 180\}$, and 120 candidates as $\tau \times 1.1^j$ and $\tau/1.1^j$ for $j \in \{1, ..., 60\}$, for a total of 481 candidate periods. When then evaluate each candidate period in simulation (see Section 7 for details on our simulation methodology) over 50 randomly generated experimental scenarios. We pick the candidate period that achieves the best average makespan over these 50 scenarios.

BestPeriod has two potential drawbacks. First, it enforces a periodic execution even though there is no theoretical reason why the optimal should correspond to a periodic execution if failures are non-Exponential. Second, it requires running a large number of simulations ($50 \times 481 = 24,050$). With our current implementation each individual set of 481 simulations requires between 3 and 24 minutes on one core of a Quad-core AMD Opteron running at 2400 MHz. While this may indicate that BestPeriod is impractical, when compared to application makespans that can be several days the overhead of searching for the period may not be significant. Furthermore, the search for the period can be done in parallel since all simulations are independent. The search for the best period to execute an application on a large-scale platform can thus be done in a few seconds on that same large-scale platform.

6.2 Dynamic Programming algorithm

Algorithm 3: DPNextCheckpoint(W, T, T_0, τ_1, ..., τ_{gq})

```
if $W = 0$ then return 0
$best\_work \leftarrow 0$
$next\_chkpt \leftarrow T$
$(W_1, ..., W_g) \leftarrow$ WorkAlreadyDone($T$) /* Work done since last recovery or checkpoint */
Sort groups by non-increasing of work done ($W_1$ is maximum)
for $t = T$ to $T + W - W_g$ step quantum                /* Loop on checkpointing date */
do
    $cur\_work \leftarrow 0$
    for $x = 1$ to $g$      /* Loop on the first group to successfully work until $t + C(q)$ */
    do
        $\delta \leftarrow (t + C(q)) - T_0$      /* Total time elapsed until the checkpoint completion */
        $proba \leftarrow \left(\prod_{y=1}^{x-1} P_{fail}(\tau_{(y-1)q+1} + \delta, ..., \tau_{(y-1)q+q} + \delta \mid \tau_{(y-1)q+1}, ..., \tau_{(y-1)q+q})\right)$
                $\times P_{suc}(\tau_{(x-1)q+1} + \delta, ..., \tau_{(x-1)q+q} + \delta \mid \tau_{(x-1)q+1}, ..., \tau_{(x-1)q+q})$
        $\omega \leftarrow \min\{W - W_x, t - T\}$                /* Work done between $T$ and $t$ by group $x$ */
        $(rec\_\omega, rec\_t) \leftarrow$ DPNextCheckpoint($W - W_x - \omega,\ T + \omega + C(q) + R(q),\ T_0,\ \tau_1, ..., \tau_{gq}$)
        $cur\_work \leftarrow cur\_work + proba \times (W_x + \omega + rec\_\omega)$
    if $cur\_work > best\_work$ then
        $best\_work \leftarrow cur\_work$
        $next\_chkpt \leftarrow t$
return $(best\_work, next\_chkpt)$
```

As an alternative to the brute-force algorithm in the previous section, one can resort to Dynamic Programming (DP). We initially developed a DP algorithm to compute chunk sizes for each group at each step of the application execution. Even though this seems like a natural approach, it is only tractable (in terms of number of DP states) if the chunk sizes for each group are computed independently of those for the other groups. As a result, we found that the resulting algorithm does not achieve good results in practice.

In our previous work [9], when faced with an exponential number of DP states when using DP to minimize expected makespan, we opted for maximizing the expected amount of completed work before the next failure. We generalize this idea to the context of replication, doing away with the concept of chunk sizes altogether. More specifically, since the first failure only interrupts a single group, the objective is to maximize the expected amount of work completed before all groups have failed. This can be achieved with the DP algorithm presented hereafter. We make one simplifying assumption: we ignore that once a group has failed, it will eventually restart and resume computing. This is because keeping track of such restarts would again lead to an exponential number of DP states. The hope is that our approach will work well in spite of this simplifying assumption.

Our DP algorithm, DPNextCheckpoint, is shown in Algorithm 3. It does *not* define chunk sizes, i.e., amounts of work to be processed before a checkpoint is taken, but instead it defines *checkpoint dates*. The rationale is that one checkpoint date can correspond to different amounts of work for each group, depending on when the group has started to process its chunk, after either its last failure and recovery, or its last checkpoint, or its last recovery from another group's checkpoint. Input to the algorithm is the amount of work that remains to be done (W), the current time (T), the time at which the application started (T_0), and the times since the latest failure at each processor before time T_0 (the τ_i's). The output is the next checkpoint date and the expected amount of work completed before the next failure occurs.

DPNextCheckpoint proceeds as follows. At Line 4 function WorkAlreadyDone is called which returns, for each group, the time since it has started processing its current chunk (i.e., the amount of work it has done to date). The groups are sorted in decreasing order of work performed to date (Line 5). The algorithm then picks the next checkpoint date for all possible dates between the current time T and time $T + W - W_g$, i.e., the time at which the last group would finish computing if no failure were to occur (Line 6). At the checkpointing date, the amount of work completed is the maximum of the amount of work done by the different groups that successfully complete the checkpoint. Therefore, we consider all the different cases (Line 9), that is, which group x, among the successful groups, has done the most work. We compute the probability of each case (Line 12). All groups that started to work earlier than group x have failed (i.e., at least one processor in each of them has failed) but not group x (i.e., none of its processors have failed). We compute the expectation of the amount of work completed in each case (Lines 13 and 14). We then sum the contributions of all the cases (Line 15) and record the checkpointing date leading to the largest expectation (Line 16). Note that the probability computed at Line 12 explicitly states which groups have successfully completed the checkpoint, and which groups have not. We choose not to take this information into account when computing the expectation (recursive call at Line 14) so as to avoid keeping track of which groups have failed, thereby lowering the complexity of the dynamic program. This is why the conditions do not evolve in the conditional probability at Line 12.

Algorithm 4 shows the overall algorithm, DPNextFailure, which uses DPNextCheckpoint (the Alive function returns, for a list of processors, the amount of time each has been up and running since its last downtime). Each time a group is affected by an event (a failure, a successful checkpoint by itself or by another group), it computes the next checkpoint date and broadcasts it to the g group leaders. Hence, a group may have computed the next checkpoint date to be t, and

Algorithm 4: DPNextFailure(W).

```
1  for each group $x = 1$ to $g$ do in parallel
2      while $W \neq 0$ do
3          $(\tau_1, ..., \tau_{gq}) \leftarrow$ Alive$(1, ..., gq)$
4          $T_0 \leftarrow$ Time()                                        /* Current time */
5          $(work, date) \leftarrow$ DPNextCheckpoint$(W,\ T_0,\ T_0,\ \tau_1, \ldots, \tau_{gq})$
6          Signal all processors that the next checkpoint date is now date
7          Try to work until date and then checkpoint
8          if successful work until date and checkpoint then
9              Let $y$ be the longest running group without failure among the successful groups
10             Let $\omega$ be the work performed by $y$ since its last recovery or checkpoint
11             $W \leftarrow W - \omega$
12             if group $x$'s last recovery or checkpoint was strictly later than that of $y$ then
13                 Perform a recovery
14         if failure then Complete downtime
15         if failure or signal then  Perform recovery from last successfully completed checkpoint
```

that date can be either un-modified, postponed, or advanced by events occurring at other groups and by their re-computation of the best next checkpoint date. In practice, as time is discretized, at each time quantum a group can check whether the current date is a checkpoint date or not.

Both Algorithms 3 and 4 have a complexity in $O\left(gq\left(\frac{W}{\mathsf{quantum}}\right)^2\right)$. The term in gq comes from the computation of the probabilities at Line 12. This complexity can be lowered using the methodology outlined in [9].

7 Simulation methodology

In this section we detail our simulation methodology. Source codes and simulation scenarios are publicly available at `http://perso.ens-lyon.fr/frederic.vivien/Data/Resilience/Replication`.

7.1 Evaluated algorithms

Our simulator implements two versions of the ASAP protocol in the case of exponentially distributed failures. The first version, OptExp, simply uses for each group the optimal and periodic policy established in [9] for Exponential failure distributions and no replication. To use OptExp with g groups we use the period from [9] computed with $\lfloor p/g \rfloor$ processors. The second, OptExpGroup, uses the periodic policy defined by Theorem 1. Both OptExp and OptExpGroup compute the checkpointing period based solely on the MTBF, assuming that failures are exponentially distributed. We nevertheless include them in all our experiments, simply using the MTBF value even when failures are not exponentially distributed. The simulator also implements BestPeriod (Section 6.1) and DPNextFailure (Section 6.2). Note that the execution times reported when using DPNextFailure include the time needed to run Algorithms 3 and 4. Based on the results in [9], we do not consider any additional checkpointing policy, such as those defined by Young [5] or Daly [4] for instance.

7.2 Platform and job parameters

We consider platforms containing from 32,768 to 4,194,304 processors. We determine the job size $\mathcal{W}$ so that a job using the whole platform would use it for a significant amount of time in the absence of failures, namely ≈ 21 hours on the largest platforms ($\mathcal{W} = 10,000$ years). In all experiments we use $D = 60\ s$, and $C = R = 60\ s$, $600\ s$, and $6000\ s$, thus spanning the spectrum from relatively fast to relatively slow checkpointing/recovery. We also ran experiments with a very short $C = R = 6\ s$, but the results are virtually identical to those obtained with $C = R = 60\ s$ and we do not present them. Finally, we use $\gamma = 10^{-6}$ for generic parallel jobs, and $\gamma = 0.1$ for numerical kernels (see Section 3). Here, we only present and discuss the constant overhead scenario ($C(q) = R(q) = C$). Results from the proportional overhead scenario are consistent with those for the constant overhead scenario and can be found in the companion research report [31].

7.3 Failure distributions

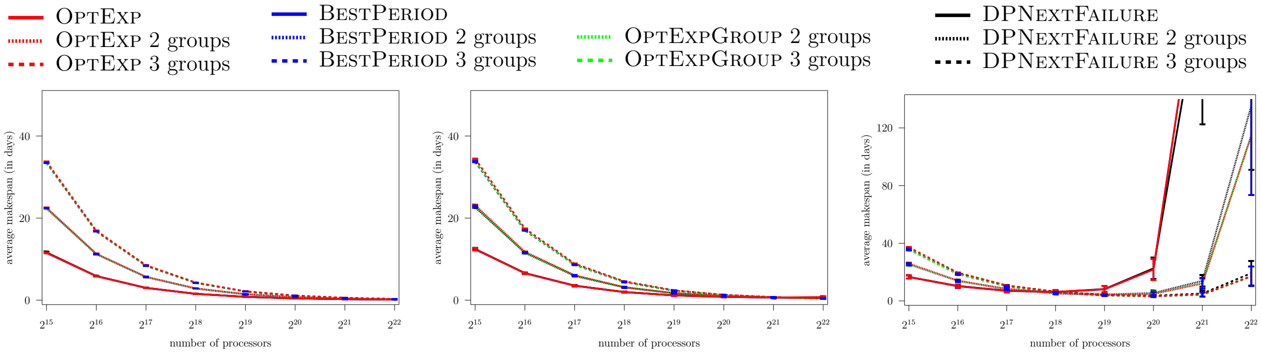
To choose failure distribution parameters that are representative of realistic systems, we use failure statistics from the Jaguar platform. Jaguar contained $45,208$ processors and is said to have experienced on the order of 1 failure per day [3]. Assuming a 1-day platform MTBF leads to a processor MTBF equal to $\frac{45{,}208}{365} \approx 125$ years. We generate both Exponential and Weibull failures, the former serving as a best case yet unrealistic scenario and the latter being representative of failure behavior in production systems [18, 19, 20, 21]. For the Exponential distribution of failure inter-arrival times, we simply set $\lambda = \frac{1}{MTBF}$. For the Weibull distribution, which requires two parameters, a shape parameter k and a scale parameter λ, and has density $\frac{k}{\lambda}(\frac{x}{\lambda})^{k-1}e^{-(x/\lambda)^k}$ for $x \geq 0$, we have $\lambda = MTBF/\Gamma(1+1/k)$. Based on the results in [18, 19, 20, 21] we use for the value of k either 0.5 or 0.7. For small values of the shape parameter k, the Weibull distribution is far from an Exponential distribution, meaning that it is far from being memoryless. We resort to generating synthetic failure traces because it is unclear how to extrapolate production failure logs for current platforms, e.g., as available in [32], to post-petascale platforms in a reasonable manner. One option is to use available smaller failure logs and use oversampling to simulate failures on larger platforms. Unfortunately, such oversampling introduces biases, and the validity of the obtained results would be questionable.

7.4 Failure scenario generation

Given a p-processor job, a failure trace is a set of failure dates for each processor over a fixed time horizon, which we set to 2 years in our simulations. The job start time is assumed to be at 1 year. We use a non-zero start time to avoid side-effects related to the synchronous initialization of all processors. Given the distribution of inter-arrival times at a processor, for each processor we generate a trace via independent sampling until the target time horizon is reached.

8 Simulation results

In this section, we only present simulation results for perfectly parallel applications under the constant overhead model. All trends and conclusions are similar regardless of the application and overhead models. For completeness, we provide the full results in a technical report [31]. All results are averages over at least 50 instances, and all graphs show one-standard-deviation error bars.



(a) $C = R = 60$ s (b) $C = R = 600$ s (c) $C = R = 6000$ s

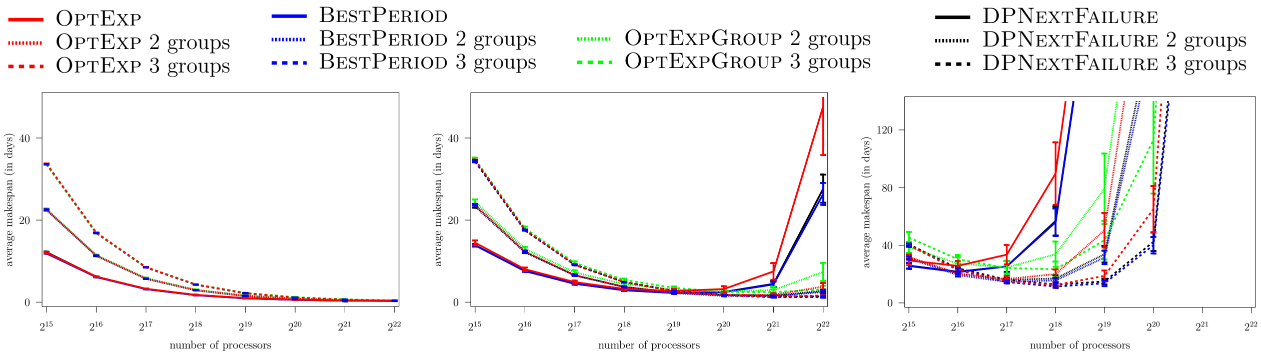
Figure 3: Average makespan vs. number of processors, Exponential failures, $MTBF = 125$ years.

8.1 Exponential failures

Figure 3 shows average makespan vs. the number of processors for our algorithms, each used assuming $g = 1$, 2, or 3 groups, assuming Exponential failures. A first observation is that many curves overlap each other: for a given g all algorithms lead to similar average makespan. For instance, for $C = R = 600$ s and $g = 2$, and taking OptExp as a reference, the relative difference between the average makespan of OptExp and that of the other three algorithms is at most 6.81% (and only 2.31% when averaged over all considered numbers of processors). In spite of such small differences, several trends emerge. OptExp almost always leads to higher average makespan than OptExpGroup (note that for $g = 1$ the two algorithms are equivalent). Over the 8 numbers of processors considered, the 3 values for $R = C$, and the 3 values for g, i.e., 72 scenarios, OptExp leads to average makespans shorter than that of OptExpGroup only 4 times (for $R = C = 6000$ s, for 2^{18} to 2^{21} processors, and by at most 3.27%). BestPeriod never leads to an average makespan higher than that of OptExp or OptExpGroup, and outperforms them by up to several percents across all the $R = C$ and g values. DPNextFailure leads to mixed results, with equal or shorter average makespan than OptExpGroup, resp. BestPeriod, for 31, resp. 24, of the 72 different scenarios.

A second observation is that the use of $g > 1$ (i.e., multiple groups) often does not help and can even lead to larger average makespans. For $R = C = 60$ s, increasing g from 1 to 2, or from 2 to 3, never leads to a lower average makespan for any of our algorithms. For $R = C = 600$ s, the only improvements are seen when going from 1 to 2 groups, for the OptExp, OptExpGroup, and BestPeriod algorithms, and only with more than 2^{21} processors. The relative improvements are at most 7.75% for 2^{21} processors, and between 25.40% and 41.09% for 2^{22} processors. No improvements are achieved when going from 2 to 3 groups. More improvements are seen for $C = R = 6000$ s. When going from 1 to 2 groups, improvements are achieved starting at 2^{18} processors, with improvements up to between 93.64% and 95.17% at large scale, for all four algorithms. When going from 2 to 3 groups, relative improvements are seen starting at 2^{19} processors, reaching up to between 85.09% and 85.78% for all four algorithms.

For low and moderate checkpointing overheads, $C = R = 60$ s or 600 s, the average makespan decreases as the number of processors increases. Instead, for high checkpointing overheads, $C = R = 6000$ s, the average makespan initially decreases but starts increasing at large scale. This is particularly noticeable when using $g = 1$ group. For instance, the average makespan using OptExp goes from 21.83 s with 2^{20} processors to 249.39 s with 2^{21} processors, or an increase by a factor 11.42. The increase is similar with BestPeriod and marginally lower with DPNextFailure



(a) $C = R = 60$ s

(b) $C = R = 600$ s

(c) $C = R = 6000$ s

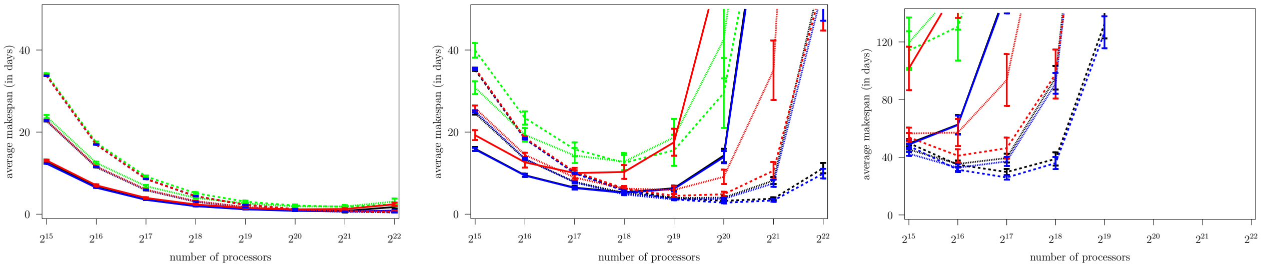
Figure 4: Average makespan vs. number of processors, Weibull failures, $k = 0.7$, $MTBF = 125$ years.

(a factor 9.72). The reason for this makespan increase is simply that with a high checkpointing overhead, the parallel efficiency is low as processors spend more time in checkpointing activities than in actual computation. This observation is precisely the motivation for using $g > 1$ (see Section 1). With $g = 2$, we still see increases in average makespans, but only by a factor between 2.46 and 2.53 when going from 2^{20} processors to 2^{21} processors for all algorithms. With $g = 3$, this factor is between 1.34 and 1.39 for all algorithms. Therefore, the use of group replication improves parallel efficiency and can lead to scalability improvements. For instance, with $g = 1$ or $g = 2$, regardless of the algorithm in use, it is not advisable to use 2^{20} processors as the makespan is lower when using 2^{19} processors. With $g = 3$, instead, there is a reduction in average makespan when going from 2^{19} processors to 2^{20} processors for all our algorithms (the relative percentage reductions are between 14.58% and 18.81%).

Based on the above, we conclude that for Exponential failures group replication can be useful when the checkpointing overhead is relatively large and/or when the scale of the execution is large. While large checkpointing overheads decrease parallel efficiency, the use of group replication makes it possible to limit this decrease or even to increase parallel efficiency at some scales. All our algorithms lead to comparable performance, with BestPeriod leading to good results even though marginally outperformed by DPNextFailure in some instances. While these results are interesting, and although Exponential failures have been studied in all previously published works, their relevance to practice is not clear given that real-world failures follow non-memoryless distributions. In the next section we present results for Weibull failures, which are more representative of real-world failure scenarios.

8.2 Weibull failures

Figures 4 and 5 show results for Weibul failures with $k = 0.7$ and $k = 0.5$, respectively. For low $R = C = 60$ s and for $k = 0.7$ (Figure 4(a)), results are similar to those seen in the previous section for Exponential failures: the use of multiple groups does not help, and all algorithms lead to sensibly the same performance. The gaps between the algorithms become larger for $k = 0.5$, i.e., when the failure distribution is farther from the Exponential distribution, with the advantage to BestPeriod (Figure 5(a)). For instance, for $k = 0.5$, 2^{20} processors, and using $g = 2$ groups, BestPeriod leads to an average makespan lower than that of OptExp, OptExpGroup, and DPNextFailure by 10.46%, 51.04%, and 2.08%, respectively. A general observation in all the results for replication ($g > 1$) with Weibull failures, regardless of the value of $C = R$, is that



(a) $C = R = 60$ s

(b) $C = R = 600$ s

(c) $C = R = 6000$ s

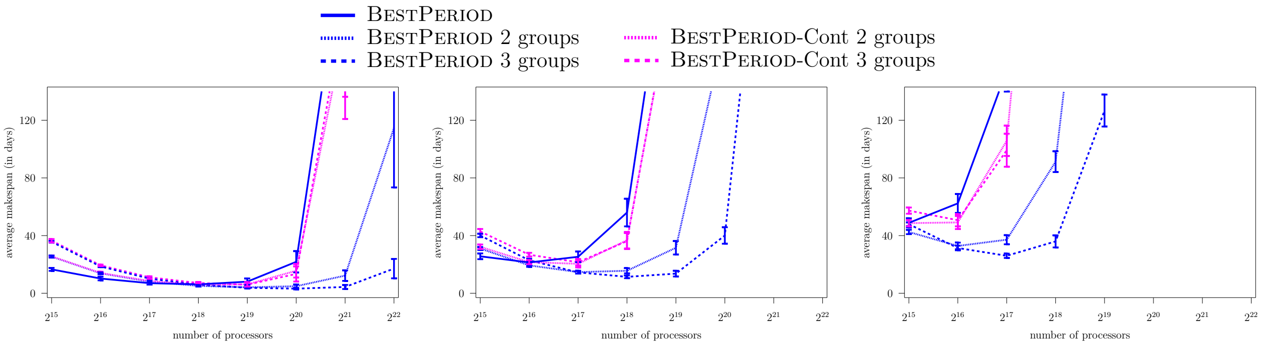
Figure 5: Average makespan vs. number of processors, Weibull failures, $k = 0.5$, $MTBF = 125$ years.

OptExpGroup leads to much poorer results than all the other algorithms. This is because the analytical development of Theorem 1 relies heavily on the Exponential failure assumption. As a result, OptExpGroup is even outperformed by OptExp, even though this algorithm also assumes Exponential failures. In all that follows we no longer discuss the results for OptExpGroup.

For $C = R = 600$ s and $k = 0.7$, and unlike the results for Exponential failures, at large scale the average makespan of the $g = 1$ executions increases sharply while the average makespans for $g > 1$ executions remain more stable (Figure 4(b)). In other words, even when checkpointing overheads are moderate, group replication is useful for increasing parallel efficiency once the scale is large enough. This result is amplified when failures are further from being Exponential, i.e., for $k = 0.5$ (Figure 5(b)). For $k = 0.5$, going from $g = 1$ to $g = 2$ groups is beneficial for OptExp starting at 2^{17} processors and for BestPeriod and DPNextFailure starting at 2^{18} processors. Going from $g = 2$ to $g = 3$ groups is beneficial for OptExp andBestPeriod starting at 2^{19} processors, and for DPNextFailure starting at 2^{20} processors. In terms of comparing the algorithms with each other, in Figure 5(b) all algorithms experience a makespan increase after the initial decrease. Only BestPeriod and DPNextFailure, when using $g = 3$ groups, have a decreasing makespan up to 2^{20} processors. When going to 2^{21} processors, these algorithms lead to relative increases in makespan of 18.50% and 14.99%, and larger increases when going from 2^{21} to 2^{22} processors. Across the board, BestPeriod with $g = 3$ groups leads to the lowest average makespan, with DPNextFailure with $g = 3$ groups a close second. The average makespan of DPNextFailure is at most 15.66% larger than that of BestPeriod, and in fact is shorter at low scales (for 2^{15} and 2^{16} processors).

Results for $C = R = 6000$ s show similar but accentuated trends. For $k = 0.7$ (Figure 4(c)) the main results are similar to those obtained for $k = 0.5$ with $C = R = 600$ s. The best two algorithms are BestPeriod and DPNextFailure using $g = 3$ groups, but both algorithms show an increase in makespan starting at 2^{19} processors. For $k = 0.5$ (Figure 5(c)) this increase occurs at 2^{18} processors and is sharper for DPNextFailure than BestPeriod. Even though group replication helps, with such large checkpointing overheads parallel efficiency cannot be maintained beyond 2^{17} processors.

We conclude that although with Exponential failures all our algorithms are more or less equivalent (see Section 8.1), with more realistic Weibul failures BestPeriod emerges as the best algorithm. The only algorithm that leads to makespans comparable to those of BestPeriod is DPNextFailure, but it never leads to a lower average makespan than BestPeriod at large scale. Even though DPNextFailure relies on a sophisticated DP approach, the brute-force but pragmatic approach used by BestPeriod turns out to be more effective. Even when using BestPeriod, our results show that application scalability is hindered by higher checkpoint overheads,



(a) Exponential failures (b) Weibull failures, $k = 0.7$ (c) Weibull failures, $k = 0.5$

Figure 6: Average makespan vs. number of processors, $C = R = 6000$ s, $MTBF = 125$ years.

which is expected, but also by lower k values, i.e., by less exponentially distributed failures.

8.3 Checkpointing contention

The results presented so far are obtained assuming that the checkpointing overhead ($R = C$) does not depend on the number of groups. There are cases in which this assumption could give an unfair advantage to group replication. Consider an application with a given memory footprint V, in bytes, running on a platform with a total of q processors. With no replication ($g = 1$) the total volume of data involved in a checkpoint is V. Assuming that V is no larger than the aggregate RAM capacity of q/g processors, then group replication can be used with $g > 1$ groups. In this case, since each group executes the application, the total volume of data involved in a checkpoint at each group is also V. Since groups may checkpoint/recover at the same time, the amount of data involved can be up to $g \times V$, or a factor g larger than in the no-replication case.

To evaluate the impact of group replication on checkpointing overhead, we introduce a checkpointing contention model in our simulation. Whenever multiple checkpointing/recovery operations are concurrent, they receive a fair share of the checkpointing/recovery bandwidth. For instance, if n checkpointing operations begin at the same time, and no other checkpointing or recovery occurs over the next $n \times C$ time units, then all n checkpointing operations finish after $n \times C$ time units. More generally, considering that a checkpointing/recovery operations requires C units of activity, over a time interval Δt during which there are n ongoing such operations each operation performs $\frac{1}{n}/\Delta t$ units of activity (if one of these operations requires fewer units of work to complete, consider a shorter Δt interval).

Our objective in this section is to determine whether group replication can still be beneficial when considering checkpointing contention. We repeated all the experiments presented in Sections 8.1 and 8.2. For $C = R = 60$ s, checkpointing contention has negligible impact on the results, and the impact for $C = R = 600$ s is lower than that for $C = R = 6000$ s. This is expected since the larger the checkpointing/recovery overhead, the more likely that more than one group is engaged in checkpointing or recovery at the same time. Thus, among all our results, those for $C = R = 6000$ s should be the most disadvantageous for group replication. These are the results presented in Figure 6, which shows average makespan vs. number of processors for BestPeriod without and with contention (denoted by BestPeriod-Cont), for $g = 1, 2$, and 3, for $C = R = 6000$ s, for Exponential failures and for Weibull failures with $k = 0.7$ and $k = 0.5$.

As expected the average makespan of BestPeriod is increased due to checkpointing contention when multiple groups are used. However, even with contention, group replication outperforms the no-replication case at large scale. For Exponential failures, using $g = 2$ groups outperforms using

$g = 1$ group as soon as the number of processors reaches 2^{18}, both with and without contention. Using $g = 3$ groups outperforms using $g = 2$ groups when there are either 2^{19} or 2^{20} processors with contention. The lowest average makespans with contention are achieved using either 2^{18} processors split in $g = 2$ groups, or 2^{19} processors split in $g = 3$ groups. For Weibull failures with $k = 0.7$, using $g = 2$ groups outperforms using $g = 1$ group starting at 2^{16} processors, with or without checkpointing contention. With contention, using $g = 3$ groups never outperforms using $g = 2$ groups, and ties its performance starting at 2^{18} processors. For Weibull failures with $k = 0.5$, using $g = 2$ groups outperforms using $g = 1$ group starting at 2^{15} processors with or without contention. With contention, using $g = 3$ groups is beneficial over using $g = 2$ groups when there are 2^{17} processors but the lowest makespan overall is achieved with $g = 2$ groups and 2^{15} processors.

We conclude that although checkpointing contention increases the makespan of group replication executions, the makespans of these executions are still shorter than that of no-replication execution at the same or slightly higher scales than when no contention takes place. One difference due to contention is that in our experiments using $g = 3$ groups is never worthwhile.

9 Conclusion

In this work we have studied group replication as a fault-tolerance mechanism for parallel applications on large-scale platforms. We have defined an execution protocol for group replication, ASAP. We have derived a bound on the expected application makespan using this protocol when failures are exponentially distributed, which suggests a checkpointing period that can be used in practice. We have also proposed two approaches to minimize application makespan that are applicable regardless of the failure distribution: (i) a brute-force search for a checkpointing period, called BestPeriod; and (ii) a Dynamic Programming algorithm, called DPNextFailure. Using simulation, and for a range of failure and checkpointing overheads, we have evaluated our proposed approaches and compared them to no-replication approaches from previous work. Our main findings are that (i) when considering realistic failures (e.g., Weibull distributed) group replication can significantly lower application makespan on large-scale platforms; (ii) our pragmatic BestPeriod approach outperforms the more sophisticated DPNextFailure Dynamic Programming approach; (iii) even when accounting for the contention due to concurrent checkpointing/recovery by multiple groups, group replication remains beneficial at large scale. Note that our group replication approaches lead to particularly good results when failures are far from being exponentially distributed, which several studies have shown to be the case in production platforms [18, 19, 20, 21].

An interesting direction for future work is to generalize this work beyond the case of coordinated checkpointing, for instance to deal with hierarchical checkpointing schemes based on message logging, or with containment domains [33]. Both these techniques alleviate the cost of checkpointing and recovery, and would dramatically decrease checkpointing contention costs. Another interesting direction would be to compare the checkpoints saved by multiple groups as a way to detect silent errors or corrupted data. This would require modifying our approach so that at least 2 groups among $g > 2$ groups compute a chunk of work successfully, thereby trading off performance for reliability.

Acknowledgments

This work was partly supported by the French ANR White Project *RESCUE*, and by the INRIA international team ALOHA. Yves Robert is with Institut Universitaire de France. We would like to thank the reviewers for their comments and suggestions, which greatly helped improve the final version of the paper.

References

[1] J. Dongarra, P. Beckman, P. Aerts, F. Cappello, T. Lippert, S. Matsuoka, P. Messina, T. Moore, R. Stevens, A. Trefethen, M. Valero, The international exascale software project: a call to cooperative action by the global high-performance community, International Journal of High Performance Computing Applications 23 (4) (2009) 309–322. doi:http://dx.doi.org/10.1177/1094342009347714.

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

[3] G. Zheng, X. Ni, L. Kale, A scalable double in-memory checkpoint and restart scheme towards exascale, in: Dependable Syst. and Networks Workshops, 2012. doi:10.1109/DSNW.2012.6264677.

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

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

[6] W. Jones, J. Daly, N. DeBardeleben, Impact of sub-optimal checkpoint intervals on application efficiency in computational clusters, in: Proc. of the International ACM Symposium on High-Performance Parallel and Distributed Computing, 2010, pp. 276–279.

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

[8] M.-S. Bouguerra, T. Gautier, D. Trystram, J.-M. Vincent, A flexible checkpoint/restart model in distributed systems, in: Proc. of PPAM, Vol. 6067 of LNCS, 2010, pp. 206–215.
URL http://dx.doi.org/10.1007/978-3-642-14390-8_22

[9] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, F. Vivien, Checkpointing strategies for parallel jobs, in: Proc. Int. Conf. High Performance Computing, Networking, Storage and Analysis SC'11, ACM Press, 2011.

[10] E. Elnozahy, J. Plank, Checkpointing for Peta-Scale Systems: A Look into the Future of Practical Rollback-Recovery, IEEE Transactions on Dependable and Secure Computing 1 (2) (2004) 97—108.

[11] R. A. Oldfield, S. Arunagiri, P. J. Teller, S. Seelam, M. R. Varela, R. Riesen, P. C. Roth, Modeling the Impact of Checkpoints on Next-Generation Systems, in: Proc. of the 24th IEEE Conference on Mass Storage Systems and Technologies, 2007, pp. 30—46.

[12] B. Schroeder, G. A. Gibson, Understanding Failures in Petascale Computers, Journal of Physics: Conference Series 78 (1).

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

[14] X.-J. Yang, Z. Wang, J. Xue, Y. Zhou, The Reliability Wall for Exascale Supercomputing, IEEE Transactions on Computers 61 (6) (2012) 767–779.

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

[16] Z. Zheng, Z. Lan, Reliability-aware scalability models for high performance computing, in: Proc. of the IEEE Conference on Cluster Computing, 2009.

[17] C. Engelmann, H. H. Ong, S. L. Scott, The case for modular redundancy in large-scale high performance computing systems, in: Proc. of the 8th IASTED International Conference on Parallel and Distributed Computing and Networks), 2009, pp. 189–194.

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

[19] B. Schroeder, G. A. Gibson, A Large-Scale Study of Failures in High-Performance Computing Systems, in: Proc. of International Conference on Dependable Systems and Networks, 2006, pp. 249–258.

[20] Y. Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Paun, S. Scott, An optimal checkpoint/restart model for a large scale high performance computing system, in: Proc. of International Parallel and Distributed Processing Symposium, 2008, pp. 1–9.

[21] R. Heien, D. Kondo, A. Gainaru, D. LaPine, B. Kramer, F. Cappello, Modeling and Tolerating Heterogeneous Failures on Large Parallel System, in: Proc. of the IEEE/ACM Supercomputing Conference (SC), 2011.

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

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

[24] S. Yi, D. Kondo, B. Kim, G. Park, 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.

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

[26] G. Amdahl, The validity of the single processor approach to achieving large scale computing capabilities, in: Proc. of AFIPS, Vol. 30, 1967, pp. 483–485.

[27] 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, R. C. Whaley, ScaLAPACK Users' Guide, SIAM, 1997.

[28] N. Kolettis, N. D. Fulton, Software Rejuvenation: Analysis, Module and Applications, in: Proc. of International Symposium on Fault-Tolerant Computing, 1995, p. 381.

[29] V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan, W. P. Zeggert, Proactive management of software aging, IBM Journal of Research and Development 45 (2) (2001) 311–332.

[30] M. Pinedo, Scheduling: theory, algorithms, and systems (3rd edition), Springer, 2008.

[31] M. Bougeret, H. Casanova, Y. Robert, F. Vivien, D. Zaidouni, Using group replication for resilience on exascale systems, Research Report RR-7876, INRIA, ENS Lyon, France, available at `http://hal.inria.fr/hal-00668016` (2012).

[32] D. Kondo, B. Javadi, A. Iosup, D. Epema, The Failure Trace Archive: Enabling Comparative Analysis of Failures in Diverse Distributed Systems, in: Proc. of the IEEE International Symposium on Cluster Computing and the Grid, IEEE Computer Society, Los Alamitos, CA, USA, 2010, pp. 398–407. doi:http://doi.ieeecomputersociety.org/10.1109/CCGRID.2010.71.

[33] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, M. Erez, Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems, in: Proc. Int. Conf. High Performance Computing, Networking, Storage and Analysis SC'12, ACM Press, 2012.