Using group replication for resilience on exascale systems

Marin Bougeret[1], Henri Casanova[2], Yves Robert[3,4], Frédéric Vivien[3] and Dounia Zaidouni[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

February 2012

Abstract

High performance computing applications must be resilient to faults, which are common occurrences especially in post-petascale settings. The traditional fault-tolerance solution is checkpoint-recovery, by which the application saves its state to secondary storage throughout execution and recovers from the latest saved state in case of a failure. An oft studied research question is that of the optimal checkpointing strategy: when should state be saved? Unfortunately, even using an optimal checkpointing strategy, the checkpointing frequency 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-recovery. Using replication, multiple processors perform the same computation so that a processor failure does not necessarily imply application failure. While at first glance replication may seem wasteful, it may be significantly more efficient than using solely checkpoint-recovery at large scale. In this work we investigate a simple approach where entire application instances are replicated. We provide a theoretical study of checkpoint-recovery with replication in terms of expected application execution time, under an exponential distribution of failures. We design dynamic-programming based algorithms to define checkpointing dates that work under any failure distribution. We also conduct simulation experiments assuming that failures follow Exponential or Weibull distributions, the latter being more representative of real-world systems, and using failure logs from production clusters. Our results show that replication is useful in a variety of realistic application and checkpointing cost scenarios for future exascale platforms.

1 Introduction

As plans are made for deploying post-petascale high performance computing (HPC) systems [10, 22], solutions need to be developed to ensure resilience to failures that occur because not all faults can be automatically detected and corrected in hardware. For instance, the 224,162-core Jaguar platform is reported to experience on the order of 1 failure per day [20, 2], and its scale is modest compared to platforms in the plans for the next decade. For applications that enroll large numbers of, or perhaps all, processors a failure is the common case rather than the exception. One can recover 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 *checkpointing strategy* specifies when checkpoints should be taken.

A large literature is devoted to developing efficient checkpointing strategies, i.e., ones that minimize expected job execution time, 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 latter 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 in the general case.

A major issue with checkpoint-recovery is scalability: the necessary checkpoint frequency for tolerating failures in large-scale platforms is so large that processors spend more time saving state than computing. It is thus expected that future platforms will lead to unacceptably low parallel efficiency if only checkpoint-recovery 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-recovery.

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 our *group replication* approach, multiple application instances are executed. One could, for instance, execute 2 distinct n-process application instances on a $2n$-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 lower checkpointing frequency. Furthermore, once an instance saves a checkpoint, the other instance can 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-recovery execution of a parallel application, for various parallel workload models for Exponential failure distributions;
- An effective approach for group replication, with a theoretical analysis bounding expected execution time for Exponential failure distribution, and several dynamic programming solutions working for general failure distributions;
- Extensive simulations showing that group replication can indeed lower application running times, and that some of our proposed strategies deliver good performance.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 defines the theoretical framework and states key assumptions. Section 4 discusses the optimal number of processors for a checkpoint-recovery execution of a parallel application under an Exponential distribution of failures. Section 5 provides several approaches for group replication, and the theoretical analysis of one of them. Section 6 describes the experimental methodology and Section 7 presents simulation results. Finally, Section 8 concludes with a summary of our findings and with future perspectives.

2 Related work

Checkpointing policies have been widely studied in the literature. In [9], Daly studies periodic checkpointing policies for Exponentially distributed failures, generalizing the well-known bound obtained by Young [28]. Daly extended his work in [15] to study the impact of sub-optimal checkpointing periods. In [25], the authors develop an "optimal" checkpointing policy, based on the popular assumption that optimal checkpointing must be periodic. In [6], 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 failures. We have developed optimal solutions for Exponential failures and dynamic programming solutions for any failure distribution, demonstrating performance improvements over checkpointing approaches proposed in the literature in the case of Weibull and log-based failures. The Weibull distribution is recognized as a reasonable approximation of failures in real-world systems [14, 24]. 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 in [4].

In spite of all the above advances, several studies have questioned the feasibility of pure checkpoint-recovery for large-scale systems (see [12] for a discussion of this issue and for references to such studies). In this work, we study the use of replication as a mechanism complementary to checkpoint-recovery. Replication has long been used as a fault-tolerance mechanism in distributed systems [13] and more recently in the context of volunteer computing [17]. The idea to use replication together with checkpoint-recovery has been studied in the context of grid computing [27].One concern about replication in HPC is the induced resource waste. However, given the scalability limitations of pure checkpoint-recovery, replication has recently received more attention in the HPC literature [23, 29, 11].

In this work we study "group replication," by which multiple application instances are executed on different groups of processors.An orthogonal approach, "process replication," was recently studied by Ferreira et al. [12] in the context of MPI applications. To achieve fault-tolerance, each MPI process is replicated in a way that is transparent to the application developer. While this approach can lead to good fault-tolerance, one of its drawbacks is that it increases the number and the 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 $V_{tot} \times g^2$. Instead, with group replication using g groups, each original communication takes place g times, hence the total communication volume increases only to $V_{tot} \times g$. Another drawback of process replication is that it requires the use of a customized MPI library (such as the prototype developed by the authors in [12]). By contrast, group replication is completely agnostic to the parallel runtime system and thus does not even require MPI. Nevertheless, even for MPI applications, group replication provides an out of the box fault-tolerance solution that can be used until process replication possibly becomes a mainstream feature in MPI implementations.

3 Framework

We consider the execution of a tightly-coupled parallel application, or *job*, on a 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 applies regardless of the granularity of the platform. We assume that system-level checkpoint-recovery is enabled.

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 processors, we use 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 [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 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 [16, 8]) 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: $C(q) = R(q) = \alpha V/q = C/q$ 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 coordinated checkpointing [26], meaning that no message logging/replay is needed when recovering from failures. 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

Let $\mathbb{E}(q)$ be the expectation of the execution time, or *makespan*, when using q processors, and q_{opt} the value of q that minimizes $\mathbb{E}(q)$. Is it true that the optimal solution is to use all processors, i.e., $q_{opt} = p$? If not, what can we say about the value of q_{opt}? This question was partially and empirically addressed in [25], 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, $\mathbb{E}(q)$ may reach its minimum for some finite value of q (implying that q_{opt} is not necessarily equal to p).

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 and that the optimal expectation of the makespan is:

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

where $\mathbb{E}(X_D(q))$ denotes the expectation of the downtime. 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 $\mathbb{E}(X_D(q))$ is difficult to compute. This is because a processor can fail while another one is

down, thus prolonging the downtime. With a single processor ($q = 1$), $X_D(q)$ has constant value D, but with several processors there could be cascading downtimes. It turns out that 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*

$$D \leq \mathbb{E}(X_D(q)) \leq \frac{e^{(q-1)\lambda D} - 1}{(q-1)\lambda} \tag{2}$$

Proof. In [4], we have shown that th optimal expectation of the makespan is computed as:

$$\mathbb{E}^*(q) = K^*(q)\left(\frac{1}{q\lambda} + \mathbb{E}(T_{rec}(q))\right)\left(e^{\frac{q\lambda\mathcal{W}(q)}{K^*(q)} + q\lambda C(q)} - 1\right) \tag{3}$$

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. We write the following recursion:

$$T_{rec}(q) = \begin{cases} X_D(q) + R(q) & \text{if no processor fails} \\ & \text{during } R(q) \text{ units of time,} \\ X_D(q) + T_{lost}(R(q)) + T_{rec}(q) & \text{otherwise.} \end{cases} \tag{4}$$

$X_D(q)$ 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 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 4 leads to:

$$\begin{aligned} \mathbb{E}(T_{rec}(q)) = e^{-q\lambda R(q)}(\mathbb{E}(X_D(q)) + R(q)) \\ + (1 - e^{-q\lambda R(q)})\left(\mathbb{E}(X_D(q)) + \frac{1}{q\lambda} - \frac{R(q)}{e^{q\lambda R(q)} - 1} + \mathbb{E}(T_{rec}(q))\right) \end{aligned} \tag{5}$$

Equation 5 reads as follows: after the downtime $X_D(q)$, either the recovery succeeds for everybody, or there is a failure during the recovery and another attempt must be made. Both events are weighted by their respective probabilities. Simplifying the above expression we get:

$$\mathbb{E}(T_{rec}(q)) = \mathbb{E}(X_D(q))e^{q\lambda R(q)} + \frac{1}{q\lambda}(e^{q\lambda R(q)} - 1) \tag{6}$$

Plugging back this expression in Equation 3, we obtain the value given in Equation 1.

Now we establish the desired bounds on $\mathbb{E}(X_D(q))$ 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, $1 \le i \le q$, let t_i denote the remaining duration of the downtime of processor i. We have $0 \le t_i \le D$ for $1 \le i \le q$, $t_{i_0} = D$, 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, $1 \le i \le q$, 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 $X_D^{t_1,..,t_q}(q)$:

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

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 $T_{lost}^{t_1,..,t_q}(D) \le T_{lost}^{D,0,...,0}(D)$ and $X_D^{t_1,..,t_q}(q) \le X_D^{D,0,..,0}(q)$ 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, $\mathbb{E}(X_D^{t_1,..,t_q}(q)) \le \mathbb{E}(X_D^{D,0,..,0}(q))$. Following the same line of reasoning, we derive an upper-bound for $X_D^{D,0,..,0}(q)$:

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

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

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

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

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

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

While in a failure-free environment $\mathbb{E}^*(q)$ would always decrease as q increases, using the above lower bound on $\mathbb{E}(X_D(q))$ we obtain the following results:

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

Note that the only open scenario is with perfectly parallel jobs and proportional overhead. In this case the lower bound on $\mathbb{E}^*(q)$ decreases to some constant value while the upper bound goes to $+\infty$ as q increases.

Proof. We show that $\lim_{q\to+\infty} \mathbb{E}^*(q) = +\infty$ for the relevant scenarios. We first plug the lower-bound of Equation 2 into Equation 6 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 $K_0(q) = \frac{q\lambda\mathcal{W}(q)}{1+\mathbb{L}(-e^{-q\lambda C(q)-1})}$.

With constant overhead. Let us consider the case of perfectly parallel jobs ($\mathcal{W}(q) = \mathcal{W}/q$) with constant checkpointing 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 $K_0(q) = \frac{\lambda\mathcal{W}}{1+\mathbb{L}(-e^{-q\lambda C-1})}$. When q tends to $+\infty$, $K_0(q)$ goes to $\lambda\mathcal{W}$, while $(\frac{1}{q\lambda}+D)e^{q\lambda C}\left(e^{\frac{\lambda\mathcal{W}}{K_0(q)}+q\lambda C} - 1\right)$ goes to $+\infty$. Consequently, $\mathbb{E}^*(q)$ is bounded below by a quantity that goes to $+\infty$, which 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 those cases than with perfectly 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. □

5 Group replication

Since using all processors to run a single application instance may not make sense in light of Theorem 1, the *group replication* approach consists in executing multiple application instances on different processor groups, where the number of processors in a group is closer to q_{opt}. 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. In this section we formalize group replication as an execution protocol called ASAP (As Soon As Possible), and analyze its performance for Exponential failures. We then introduce dynamic programming solutions that work with general failure distributions.

5.1 The ASAP execution protocol

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, 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 processor failure at time t, the group is *down* between t and $t + X_D(q)$.

The ASAP algorithm proceeds in k macro-steps. 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 $j < k$). Note that the value of k, the total number of chunks, as well as the chunk sizes, the ω_j's, are inputs to the algorithm (we always have $\sum_{j=1}^{k}\omega_j = \mathcal{W}(q)$). We provide an analytical evaluation of ASAP for Exponential failure laws, and discuss how to choose these values, in Section 5.2.

Two important aspects must be mentioned. 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 a successful group. Second, this recovery may start later than time t_j^{end}, in the case where the group is down at time t_j^{end}. An example is shown in Figure 1, in which group 1 cannot start the recovery at time t_j^{end}. The only groups that do not need to recover at the beginning of the next step are the groups that were successful for the previous step, except during the first step at which all groups can start computing right away.

We now provide an analytical evaluation of ASAP for Exponential failure laws, and show how to compute the number of macro-steps k and the values of the chunk sizes ω_j.

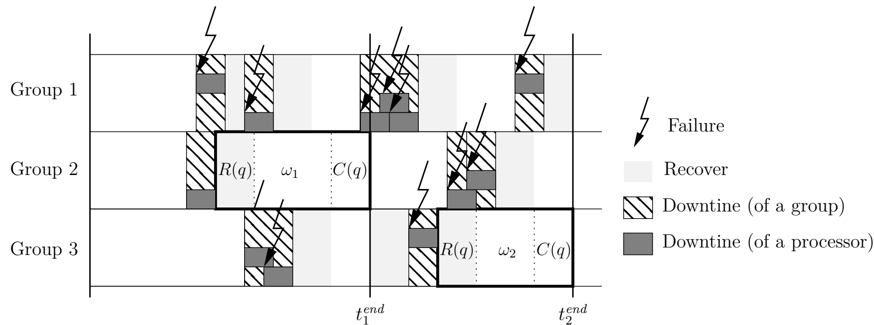


Figure 1: Execution of chunks ω_1 and ω_2 (macro-steps 1 and 2) using the ASAP protocol. At time t_1^{end}, Group 1 is not ready, and Group 2 is the only one that does not need to recover.

5.2 Exponential failures

Let use assume that 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 in which all groups, including the successful ones, execute a recovery at the beginning of all macro-steps, including the first one. This new 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 $R(q) + \omega_j + C(q)$.

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

```
1 for j = 1 to k do
2 |   for each group do in parallel
3 |   |   repeat
4 |   |   |   Finish current downtime (if any)
5 |   |   |   Try to perform a recovery, then a chunk of size ω_j, and finally to checkpoint
6 |   |   |   if execution successful then
7 |   |   |   |   Signal other groups to immediately stop their attempts
8 |   |   until one of the groups has a successful attempt
```

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. In Figure 2, we have $j = 2$, the successful chunk of size $R + \omega_2 + C$ is the fourth attempt, so $N_2 = 4$. To represent each attempt, we sample random variables X_i^j and Y_i^j, $1 \leq i \leq N_j$, that correspond respectively to the ith tentative execution of the chunk and to the ith 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's follow the same distribution D_X, namely an Exponential law of parameter $q\lambda$. And all the Y_i^j's 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 using the classical online *list scheduling* algorithm for independent jobs [21, section 5.6]. This formulation (see Proposition 2) 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). We then derive an upper bound for the expected execution time of ASAP (see Theorem 2).

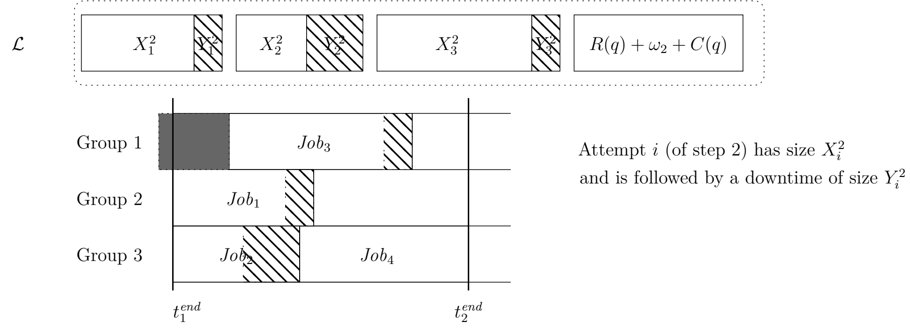


Figure 2: Zoom on macro-step 2 of the execution depicted in Figure 1, using the (X, Y) notation of Algorithm 2. Recall that Job_i has size $X_i^2 + Y_i^2$ for $1 \leq i \leq 3$, and Job_4 has size $R(q) + \omega_2 + C(q)$.

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

1 $i \leftarrow 1$ /* i represents the number of attempts for the job */
2 $\mathcal{L} \leftarrow \emptyset$ /* $\mathcal{L}$ represents the list of attempts for the job */
3 Sample X_i^j and Y_i^j using D_X and $D_{X_D(q)}$ respectively
4 **while** $X_i^j < R(q) + \omega_j + C(q)$ **do**
5 Add Job_i, with processing time $X_i^j + Y_i^j$, to $\mathcal{L}$
6 $i \leftarrow i + 1$
7 Sample X_i^j and Y_i^j using D_X and $D_{X_D(q)}$ respectively
8 $N_j \leftarrow i$
9 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 $X_{N_j}^j + Y_{N_j}^j$ */
10 From time t_{j-1}^{end} on, execute a List Scheduling algorithm to distribute jobs of $\mathcal{L}$ to the different groups (recall that some groups may not be ready at time t_{j-1}^{end})

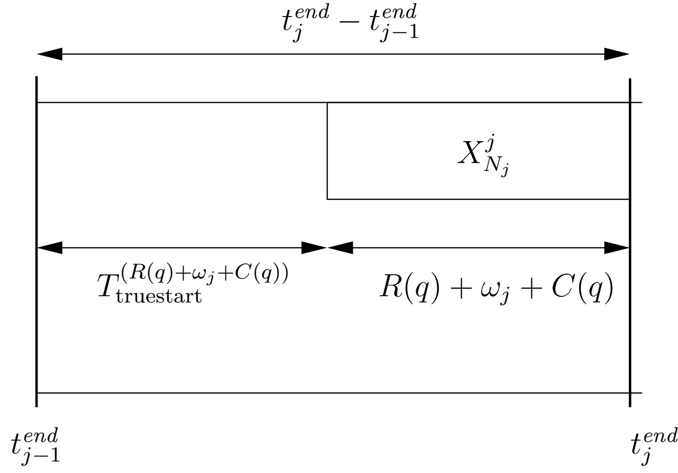


Figure 3: Notations used in Proposition 3.

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

Proof. The List 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. □

Proposition 3. *Let $T_{truestart}^{(R(q)+\omega_j+C(q))}$ be the time elapsed between t_{j-1}^{end} and the beginning of Job_{N_j} (see Figure 3). We have $\mathbb{E}\left(T_{truestart}^{(R(q)+\omega_j+C(q))}\right) \leq \mathbb{E}(Y) + \frac{\mathbb{E}(N_j)\mathbb{E}(X) - \mathbb{E}(X_j^{N_j}) + (\mathbb{E}(N_j)-1)\mathbb{E}(Y)}{g}$ where X and Y are random variables corresponding to an attempt (sampled using D_X and $D_{X_{D(q)}}$ respectively). Moreover, we have $\mathbb{E}(N_j) = e^{\lambda q(R(q)+\omega_j+C(q))}$ and $\mathbb{E}(X_j^{N_j}) = \frac{1}{q\lambda} + R(q) + \omega_j + C(q)$.*

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_{j-1}^{end}), while $\tilde{Y_2} = \tilde{Y_3} = 0$ (groups 2 and 3 are ready to compute at time t_{j-1}^{end}). Proposition 2 has shown that executing macro-step j can be simulated by executing a List Schedule on a job list $\mathcal{L}$ (see Algorithm 2). We now consider g "jobs" $\tilde{Job_x}$, $x = 1, \ldots, g$, so that $\tilde{Job_x}$ has duration $\tilde{Y_x}$. We now consider the augmented job list $\mathcal{L}' = \mathcal{L} \cup \bigcup_{x=1}^{g} \tilde{Job_x}$. Note that $\mathcal{L}'$ may contain more 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 start and completion dates. Thus, we have $T_{\text{truestart}}^{(R(q)+\omega_j+C(q))} \leq \frac{\sum_{x=1}^{g}(\tilde{Y_x}) + \sum_{i=1}^{N_j-1}(X_i^j + Y_i^j)}{g}$: this key inequality is due to the property of list 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

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

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 $q\lambda$. □

Theorem 2. *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:*

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

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

$$\tau_1 = \frac{1}{g}\left(\frac{1}{q\lambda} + \mathbb{E}(Y)\right)e^{\lambda q(R(q)+C(q))} \quad \text{and}$$
$$\tau_2 = \left(\frac{g-1}{g}\left(\mathbb{E}(Y) + R(q) + C(q)\right) - \frac{1}{g}\frac{1}{q\lambda}\right).$$

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 $\mathbb{E}(Y) = \mathbb{E}(X_D(q))$ in Proposition 1, we can compute numerically the number of chunks and the expectation of the upper bound given by Theorem 2.

5.3 Group replication heuristics for general failure distributions

The results of the previous section are limited to Exponential failures. We now address turn to the general case. In Section 5.3.1 we recall the dynamic program designed in [4] to define checkpointing dates in a context *without* replication. We then discuss how to use this dynamic program in the context of ASAP (Section 5.3.2). Finally, in Section 5.3.3 we propose heuristics that correspond to more general execution protocols than ASAP.

5.3.1 Solution without replication

According to [4], the most efficient algorithm to define checkpointing dates, for *general* failure distributions and when no replication is used, is the dynamic programming approach called DPNext-Failure, shown in Algorithm 3. This algorithm works on a failure by failure basis, maximizing the expectation of the amount of work completed (and checkpointed) before the next failure occurs. This algorithm only provides an approximation of the optimal solution as it relies on a time discretization. (For the sake of simplicity, we present all dynamic programs as recursive algorithms.)

Algorithm 3: DPNextFailure (W, τ_1, ..., τ_p)

```
1  if W = 0 then return 0
2  best ← 0;       chunksize ← 0
3  for ω = quantum to W step quantum do
4  |  (expected_work, 1st_chunk) ← DPNextFailure(W − ω, τ_1 + ω + C(p), ..., τ_p + ω + C(p))
5  |  cur_exp_work ← P_suc(τ_1 + ω + C(p), ..., τ_p + ω + C(p) | τ_1, ..., τ_p) × (ω + expected_work)
6  |  if cur_exp_work > best then best ← cur_exp_work;    chunksize ← ω
7  return (best, chunksize)
```

5.3.2 Implementing the ASAP execution protocol

A key question when implementing ASAP is that of the chunk size. A naive approach would be to use DPNextFailure. Each group would call DPNextFailure to compute what would be the optimal chunk size for itself, as if there were no other groups. Then we have to *merge* these individual chunk sizes to obtain a common chunk size. This heuristic, DPNextFailureAsap, is shown in Algorithm 4 (the Alive function returns, for a list of q processors, the amount of time each has been up and running since its last downtime). For the Merge operator, one could be pessimistic and take the minimum of the chunk sizes, or be optimistic and take the maximum, or attempt a trade-off by taking the average. Two important limitations of this heuristic are: 1) the Merge operator that has no theoretical justification; and 2) the use of chunk sizes defined using an obsolete failure history. The latter limitation shows after a group is victim of a failure: the failure history has changed significantly but the chunk size is not recomputed (this has no consequences with an Exponential distribution as the Exponential is memoryless).

Algorithm 4: DPNextFailureAsap(W).

```
1  while W ≠ 0 do
2  |  for each group x = 1..g do in parallel
3  |  |  (τ_{(x−1)q+1}, ..., τ_{(x−1)q+q}) ← Alive((x − 1)q + 1, ..., (x − 1)q + g)
4  |  |  (exp_work_x, ω_x) ← DPNextFailure(W, τ_{(x−1)q+1}, ..., τ_{(x−1)q+q})
5  |  ω ← Merge(ω_1, ..., ω_g)
6  |  for each group do in parallel
7  |  |  repeat
8  |  |  |  Try to execute a chunk of size ω and then checkpoint
9  |  |  |  if successful then
10 |  |  |  |  Signal other groups to immediately stop their attempts
11 |  |  |  else if failure then Complete downtime and perform recovery
12 |  |  until One of the groups signals its success
13 |  W ← W − ω
14 |  for each group do in parallel
15 |  |  if not successful on last chunk then  Perform recovery from last successfully completed
   |  |  checkpoint
```

5.3.3 Other execution protocols

In order to circumvent both limitations of DPNextFailureAsap, we relax the constraint that all groups work with the same chunk size. Each group now works with its own chunk size that is recomputed each time the group fails, and each time one of the groups successfully complete its own chunk. This leads to the heuristic DPNextFailureSynchro in Algorithm 5. Each time a

group successfully works for the duration of its chunk size and checkpoints its work, it signals its success to all groups which then interrupt their own work. This behavior is clearly sub-optimal if the successful completion is for a very small chunk and an interrupted group that was computing a large chunk was close to completion. The problems are that: 1) chunk sizes are defined as if each group was alone; and 2) the definition of the chunk sizes does not account for the fact that the first group to complete its checkpoint defines a mandatory checkpointing date for all groups.

Algorithm 5: DPNextFailureSynchro(W).

```
1  for each group x = 1..g do in parallel
2  |  while W ≠ 0 do
3  |  |  (τ_{(x-1)q+1}, ..., τ_{(x-1)q+q}) ← Alive((x-1)q+1, ..., (x-1)q+g)
4  |  |  (exp_work_x, ω_x) ← DPNextFailure(W, τ_{(x-1)q+1}, ..., τ_{(x-1)q+q})
5  |  |  Try to execute a chunk of size ω_x and then checkpoint
6  |  |  if successful then
7  |  |  |  Signal other groups to immediately stop their attempts
8  |  |  |  W ← W - ω_x
9  |  |  if failure then Complete downtime
10 |  |  if failure or signal then
11 |  |  |  Perform recovery from last successfully completed checkpoint
```

To address these problems, rather than doing another attempt at reusing DPNextFailure, we design a brand new dynamic program. In [4], without replication, we found that a dynamic program that minimizes the expectation of the makespan would require an intractable exponential number of states to record which processors fail and when. Hence we aimed instead at maximizing the expectation of the work completed before the next failure. We now extend this approach to the context of replication. The first failure will only interrupt a single group. Therefore, the objective should be to maximize the expectation of the work completed before all groups have failed. This approach ignores that once a group has failed, it will eventually restart and resume computing. However, keeping track of such restarts would require recording which processors have failed and when, thus once again leading to an exponential number of states.

To avoid having the first completed checkpoint force a checkpoint for all other groups we design a new dynamic program, DPNextCheckpoint (Algorithm 6). DPNextCheckpoint does not define chunk sizes, i.e., amount of work to be processed before a checkpoint is taken, but instead it defines checkpoint dates. The rationale is that one checkpointing 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 based on another group's checkpoint. The function WorkAlreadyDone (Line 3) returns, for each group, the time since it started processing its current chunk.

DPNextCheckpoint proceeds as follows. 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 8), that is, which group x, among the successful groups, has done the most work. We compute the probability of each case (Line 11). 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 12 and 13). We then sum the contributions of all the cases (Line 14) and record the checkpointing date leading to the largest expectation (Line 15). Note that the probability computed at Line 11 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 13). This is to avoid keeping track of which group had failed, thereby lowering the complexity of the dynamic program. This explains why the conditions do not evolve in the conditional probability at Line 11.

Algorithm 6: DPNextCheckpoint(W, T, T_0, τ_1, ..., τ_{gq})

1 **if** $W = 0$ **then return** 0
2 $best_work \leftarrow 0$; $next_chkpt \leftarrow date$
3 $(W_1, ..., W_g) \leftarrow$ WorkAlreadyDone(T) /* Time since last recovery or checkpoint */
4 Reorder groups in non-increasing availabilities (W_1 is maximum)
5 **for** $t = T$ *to* $T + W - W_g$ *step* quantum /* Loop on checkpointing date */
6 **do**
7 | $cur_work \leftarrow 0$
8 | **for** $x = 1$ *to* g /* Loop on the first group to successfully work until $t + C(q)$ */
9 | **do**
10 | | $\delta \leftarrow (t + C(q)) - T_0$ /* Total time elapsed until the checkpoint completion */
11 | | $proba \leftarrow \prod_{y=1}^{x-1} P_{fail}(\tau_{(y-1)q+1} + \delta, ..., \tau_{(y-1)q+p} + \delta \,|\, \tau_{(y-1)q+1}, ..., \tau_{(y-1)q+p})$ $\times P_{suc}(\tau_{(x-1)q+1} + \delta, ..., \tau_{(x-1)q+p} + \delta \,|\, \tau_{(x-1)q+1}, ..., \tau_{(x-1)q+p})$
12 | | $\omega \leftarrow \min\{W - W_x, t - T\}$ /* Work done between T and t by group x */
13 | | $(rec_\omega, rec_t) \leftarrow$ DPNextCheckpoint($W - W_x - \omega,\ T + \omega + C(q) + R(q),\ T_0,\ \tau_1, ..., \tau_p$)
14 | | $cur_work \leftarrow cur_work + proba \times (W_x + \omega + rec_\omega)$
15 | **if** $cur_work > best_work$ **then** $best_work \leftarrow cur_work$; $next_chkpt \leftarrow t$
16 **return** $(best_work, next_chkpt)$

Algorithm 7: DPNextFailureGlobal(W).

1 **for** *each group* $x = 1..g$ **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 | | $date \leftarrow$ 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 ω be the work performed by y since its last recovery or checkpoint
11 | | | $W \leftarrow W - \omega$
12 | | | **if** *group x 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

Finally, Algorithm 7 shows the algorithm, called DPNextFailureGlobal, that uses DP-NextCheckpoint. 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 signals the result of its computation to the other groups (e.g., by broadcasting it to the g group leaders). Hence, a group may have computed the next checkpoint date to be t, and that date can be either un-modified, or postponed, or advanced by events occurring on other groups and by their re-computation of the best next checkpoint date. In practice, as these algorithms rely on a time discretization, at each time quantum a group can check whether the current time is a checkpoint date or not.

6 Simulation framework

In this section we detail our simulation methodology. We use both synthetic and real-world failure distributions. The source code and all simulation results are publicly available at: `http://perso.ens-lyon.fr/frederic.vivien/Data/Resilience/SPAA2012/`.

6.1 Heuristics

Our simulator implements the following eight checkpointing policies:

- Two versions of the ASAP protocol: OptExp, that uses the optimal and periodic policy established in [4] for Exponential failure distributions and no replication; OptExpGroup, that uses the periodic policy defined by Theorem 2 for Exponential distributions.
- The six dynamic programming approaches in Section 5.3: DPNextFailure, the 3 variants of DPNextFailureAsap, DPNextFailureSynchro, and DPNextFailureGlobal.

Our simulator also implements PeriodLB, which is a numerical search for the optimal period for ASAP. We evaluate each candidate period on 50 randomly generated scenarios. To build the candidate periods, the period computed for OptExp is multiplied and divided by $1 + 0.05 \times i$ with $i \in \{1, ..., 180\}$, and by 1.1^j with $j \in \{1, ..., 60\}$. PeriodLB corresponds to the periodic policy that uses the best period found by the search. The evaluation of PeriodLB on a configuration requires running 24,000 simulations (which would be prohibitive in practice), but we include it for reference. Based on the results in [4], we do not consider any additional checkpointing policy, such as those defined by Young [28] or Daly [9] for instance. We point out that OptExp and OptExpGroup compute the checkpointing period based solely on the MTBF, implicitly assuming that failures are exponentially distributed. For the sake of completeness we nevertheless include them in all our simulations, simply using the MTBF value even when failures are not exponentially distributed. To use OptExp with g groups we use the period from [4] computed with $\lfloor p/g \rfloor$ processors.

6.2 Platforms

We target two types of platforms, depending on the type of the failure distribution. For synthetic distributions, we consider platforms containing from 32,768 to 1,048,576 processors. For platforms with failures based on failure logs from production clusters, because of the limited scale of those clusters, we restrict the size of the platforms to a maximum of 131,072 processors, starting with 4,096 processors. For both platform types, 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 ≈ 3.5 days on the largest platforms for synthetic failures ($\mathcal{W} = 10,000$ years), and ≈ 2.8 days on those for log-based failures ($\mathcal{W} = 1,000$ years). Otherwise, we use the same parameters as in [4]: $C = R = 600$ s, $D = 60$ s, $\gamma = 10^{-6}$ for generic parallel jobs, and $\gamma = 0.1$ for numerical kernels. Note that the checkpointing overheads come from the scenarios in [7] and are smaller than those used in [12].

6.3 Failure scenarios

Synthetic failure distributions – To choose failure distribution parameters that are representative of realistic systems, we use failure statistics from the Jaguar platform. Jaguar contains $45,208$ processors and is said to experience on the order of 1 failure per day [20, 2]. Assuming a 1-day platform MTBF gives us a processor MTBF equal to $\frac{45,208}{365} \approx 125$ years. For the Exponential distribution, we then have λ as $\lambda = \frac{1}{MTBF}$ and for Weibull, which requires two parameters k and

λ, we have $\lambda = MTBF/\Gamma(1+1/k)$ and we fix k to 0.7 based on the results of [24]. (We have shown in [4] that the general trends were not influenced by the exact values used for k nor for the MTBF.)

Log-based failure distributions – We also consider failure distributions based on failure logs from production clusters. We used logs from the largest clusters among the preprocessed logs in the *Failure trace archive* [18], i.e., from clusters at the Los Alamos National Laboratory [24]. In these logs, each failure is tagged by the node —and not just the processor— on which the failure occurred. Among the 26 possible clusters, we opted for the only two clusters with more than 1,000 nodes, as we needed a sample history sufficiently large to simulate platforms with more than 10,000 nodes. The two chosen logs are for clusters 18 and 19 in the archive (referred to as 7 and 8 in [24]). For each log, we record the set $\mathcal{S}$ of availability intervals. A discrete failure distributionfor the simulation is then generated as follows: the conditional probability $\mathbb{P}(X \geq t \mid X \geq \tau)$ that a node stays up for a duration t, knowing that it has been up for a duration τ, is set to the ratio of the number of availability durations in $\mathcal{S}$ greater than or equal to t, over the number of availability durations in $\mathcal{S}$ greater than or equal to τ.

Scenario generation – Given a p-processor job, a failure trace is a set of failure dates for each processor over a fixed time horizon h (set to 2 years). The job start time is assumed to be 1 year for synthetic distribution platforms, and 0.25 year for log-based distribution platforms. We use a non-null 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.For simulations where the only varying parameter is the number of processors $a \leq p \leq b$, we first generate traces for b processors. For experiments with p processors we then simply select the first p traces. This ensures that simulation results are coherent when varying p. Finally, the two clusters used for computing our log-based failure distributions consist of 4-processor nodes. Hence, to simulate a 131,072-processor platform we generate 32,768 failure traces, one for each four-processor node.

7 Simulation results

In this section we discuss simulation results, but only show graphs for perfectly parallel applications under the constant overhead scenario. All results can be found in the appendix. All simulations with replication are for two groups (using three groups never leads to improvements in our experiments).

Optimal number of processors for execution – For Weibull and log-based failures with the constant overhead scenario, using all the available processors to run a single application instance leads to significantly larger makespans. This is seen in Figures 5-7 where the curves for OptExp, PeriodLB, and DPNextFailure, the no-replication strategies, shoot upward when p reaches a large enough value. For instance, with traces based on the logs of LANL cluster 18, the increase in makespan is more than 37% when going from $p = 2^{16}$ to $p = 2^{17}$. This behavior is in line with the result in Theorem 1. However, in our experiments with Exponential failures the lowest makespan when running a single application instance is obtained when using all the processors, regardless of the application model. This is seen plainly in Figure 4 at least up to 2^{20} processors. This likely means that our experimental scenarios have not reached the scale at which the makespan would increase. Furthermore, running a single application instance is not always best in our results even for Exponential failures. For instance, for generic parallel jobs OptExp (with $g = 1$) begins being outperformed by strategies that use replication at 2^{20} processors (see the full results in [5]). We conclude that on exascale platforms the optimal makespan may not be obtained by running a

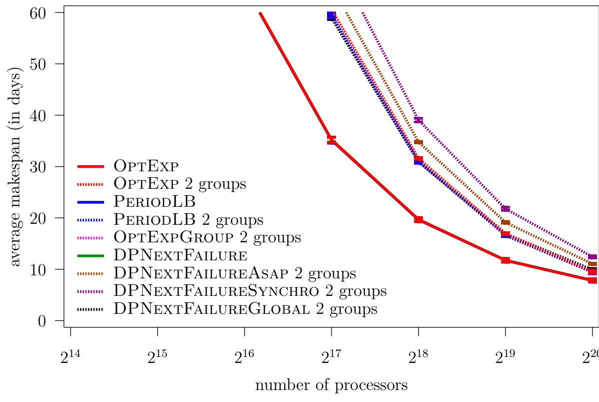


Figure 4: Exponential failures with MTBF=125 years.

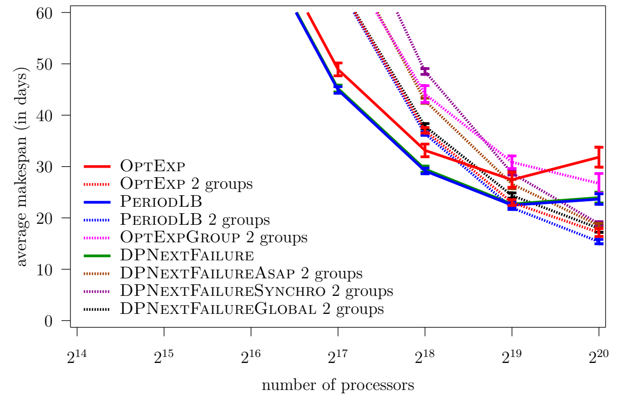


Figure 5: Weibull failures with MTBF=125 years and $k = 0.70$.

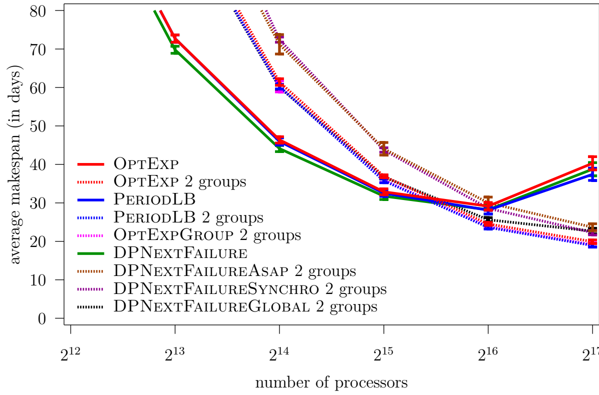


Figure 6: Failures based on the failure log of LANL cluster 18.

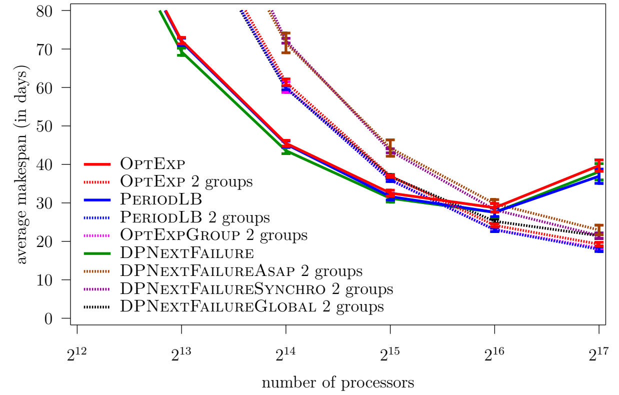


Figure 7: Failures based on the failure log of LANL cluster 19.

single application instance. This is similar to the result obtained in [12] in the context of process replication.

The ASAP protocol with Exponential – Considering only those results obtained for Exponential failures and using group replication, we find that OptExpGroup with $g = 2$ delivers performance very close to that of the numerical lower bound on any periodic policy under ASAP (PeriodLB with $g = 2$) and slightly better than the performance of OptExp with $g = 2$.These results corroborates our theoretical study. Therefore, determining chunk sizes according to Theorem 2 is more efficient than using two instances that each use the chunk size determined by OptExp.

The dynamic programming approaches - The performance of the three variants of DPNextFailureAsap (with Merge being the minimum, the average, or the maximum) are indistinguishable on Exponential and Weibull failure distributions. For log-based failure distributions, their performance are still very close, with the average and the maximum seemingly achieving better performance (the performance differences, however, may not be significant in light of the standard deviations).

DPNextFailureSynchro achieves worse performance than DPNextFailureAsap for Ex-

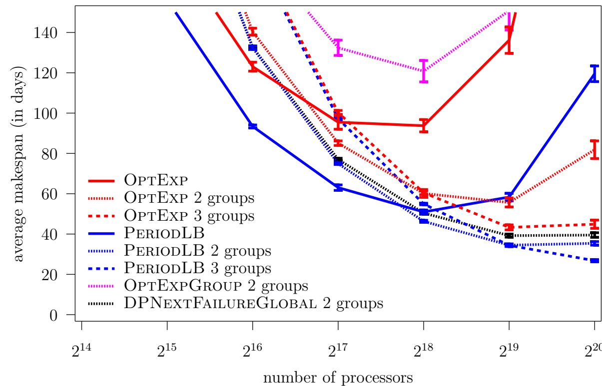


Figure 8: Weibull failures with MTBF=125 years and $k = 0.50$.

ponential and Weibull failure distributions. For log-based failure distributions, in some rare settings, the opposite is true. Overall, this heuristic provides disappointing results. By contrast, depending on the scenario, DPNextFailureGlobal is equivalent to the other dynamic programming approaches or outperforms them significantly. Furthermore, its performance is relatively close to that of PeriodLB with $g = 2$.

The very perplexing result here is that, regardless of the underlying failure distribution, DP-NextFailureGlobal is outperformed by the periodic policy OptExp with $g = 2$. Recall that this policy erroneously assumes Exponential failures and computes the checkpointing period ignoring that replication is used! By contrast, DPNextFailureGlobal is based on strong theoretical foundations and was designed specifically to handle multiple groups efficiently. And yet, in non-Exponential failure cases, OptExp with $g = 2$ outperforms DPNextFailureGlobal (see Figures 5-7). In the case of log-based experiments (Figures 6 and 7), it may be that our datasets are not sufficiently large. We use failure data from the largest production clusters in the *Failure trace archive* [18], but these clusters are orders of magnitude smaller than our simulated platforms. Consequently, we are forced to "replay" the same failure data many times for many of our simulated nodes, which biases experiments. In the case of Weibull failures with $k = 0.7$ (Figure 5), it turns out that the failure distribution is sufficiently close to an Exponential, i.e., that k is sufficiently close to 1, so that OptExp does well. However, smaller values of k are reasonable as seen for instance in [19] ($k \approx 0.5$) and in [24] ($0.33 \leq k \leq 0.49$). In Table 1 we show results for smaller values of k for a platform with 2^{20} processors. We see that, as k decreases, the performance of OptExp with $g = 2$ degrades when compared to PeriodLB, reaching more than 100% degradation for $k = 0.5$. However, DPNextFailureGlobal still achieves close-to-optimal makespans in such cases, as can be seen on Figure 8 (for $k = 0.5$). Overall, DPNextFailureGlobal achieves good to very good performance all across the board. Finally, remark on Figure 8 that PeriodLB achieves better performance using a replication factor of 3 ($g = 3$) than with a replication factor of 2.

8 Conclusion

In this paper we have presented a study of replication techniques for large-scale platforms. These platforms are subject to failures, the frequency of which increase dramatically with platform scale. For a variety of job types (perfectly parallel, generic, or numerical) and checkpoint cost models (constant or proportional overhead), we have shown 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 led us to investigate replication as a technique to better use all the resources provided by the platform. While others have studied process replication [12], in this work we have focused on group replication, a more general and less intrusive approach. Group replication consists in partitioning the platform into several groups, which each executes an instance of the application concurrently. All groups coordinate to take advantage of the most advanced application state available. We have conducted an analytical study of group replication for Exponential failures when using the ASAP execution protocol, using an analogy with a list schedule to bound the number of attempts and the execution time of each group. This analogy makes it possible to compute an upper bound on the expected execution time, which can be computed numerically. We have also proposed several dynamic programming algorithms to minimize application makespan, whatever the failure distribution. We have compared all these approaches, along with no-replication approaches proposed in previous work, in simulation for both synthetic failure distributions and failure data from production clusters. Our main findings are 1) that replication can significantly lower the execution time of applications on very large scale platforms, for failure and checkpointing characteristics corresponding to today's platforms; 2) that our DPNextFailureGlobal dynamic programming approach is close to the optimal periodic solution (in fact close to PeriodLB, which uses a prohibitively expensive numerical search for the best period).

A clear direction for future work, which we are currently exploring and that could lead to results in the final version of the article, is the reasons for the unexpected good results for OptExp with $g = 2$. In terms of longer-term objectives, it would be interesting to generalize this work beyond the case of coordinated checkpointing. For instance, we plan to study hierarchical checkpointing schemes with message logging. Another interesting direction would be to study process replication and compare it to group replication, both theoretically and through simulations.

Acknowledgments

This work was supported in part by the ANR RESCUE project, and by the INRIA-Illinois Joint Laboratory for Petascale Computing.

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] Marin Bougeret, Henri Casanova, Mikael Rabie, Yves Robert, and Frédéric 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] Marin Bougeret, Henri Casanova, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. Using group replication for resilience on exascale systems. Research Report RR-7872, INRIA, ENS Lyon, France, 2012. Available at `graal.ens-lyon.fr/~yrobert/`.

[6] Mohamed-Slim Bouguerra, Thierry Gautier, Denis Trystram, and Jean-Marc Vincent. A flexible checkpoint/restart model in distributed systems. In *PPAM*, volume 6067 of *LNCS*, pages 206–215, 2010.

[7] Franck Cappello, Henri Casanova, and Yves Robert. Checkpointing vs. migration for post-petascale supercomputers. In *ICPP'2010*. IEEE Computer Society Press, 2010.

[8] 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.

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

[10] Jack Dongarra, Pete Beckman, Patrick Aerts, Frank Cappello, Thomas Lippert, Satoshi Matsuoka, Paul Messina, Terry Moore, Rick Stevens, Anne Trefethen, and Mateo 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.

[11] 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.

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

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

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

[15] W.M. Jones, J.T. Daly, and N. DeBardeleben. Impact of sub-optimal checkpoint intervals on application efficiency in computational clusters. In *HPDC'10*, pages 276–279. ACM, 2010.

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

[17] 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.

[18] Derrick Kondo, Bahman Javadi, Alexandru Iosup, and Dick Epema. The failure trace archive: Enabling comparative analysis of failures in diverse distributed systems. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:398–407, 2010.

k	PeriodLB ($g = 2$)	OptExp ($g = 2$)	Relative degradation
0.5	35.57	81.20	128.30%
0.6	21.50	30.10	39.99%
0.7	15.38	17.14	11.44%
0.8	12.27	12.45	1.46%
0.9	10.45	10.47	0.21%

Table 1: Makespans (in days) obtained by PeriodLB with $g = 2$ and OptExp with $g = 2$, and relative degradation of OptExp, as a function of k, the shape parameter of the Weibull distribution of failures, on platforms with $1,048,576$ processors.

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

[20] 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`.

[21] Michael Pinedo. *Scheduling: theory, algorithms, and systems (3rd edition)*. Springer, 2008.

[22] Vivek 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`.

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

[24] 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.

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

[26] L. Wang, P. Karthik, Z. Kalbarczyk, R.K. 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.

[27] 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.

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

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

A Detailed simulation results

	$g=1$			$g=2$							
p	OptExp	PeriodLB	DPNextFailure	OptExp	OptExpGroup	PeriodLB	DPNextFailureAsap			DPNextFailureSynchro	DPNextFailureGlobal
							min	avg	max		
32768	124.14 ± 0.86	124.14 ± 0.86	124.12 ± 0.77	231.72 ± 0.33	228.74 ± 0.56	228.53 ± 0.58	238.74 ± 0.49	238.75 ± 0.56	238.75 ± 0.29	254.27 ± 1.32	226.89 ± 0.59
65536	65.21 ± 0.60	65.21 ± 0.60	65.38 ± 0.57	117.96 ± 0.18	116.11 ± 0.51	116.07 ± 0.54	122.99 ± 0.28	122.99 ± 0.31	122.93 ± 0.26	133.61 ± 0.88	114.94 ± 0.34
131072	35.16 ± 0.53	35.12 ± 0.56	35.27 ± 0.47	60.61 ± 0.15	59.53 ± 0.44	59.49 ± 0.34	65.14 ± 0.17	65.14 ± 0.26	65.20 ± 0.39	71.65 ± 0.50	58.94 ± 0.30
262144	19.71 ± 0.37	19.63 ± 0.35	19.73 ± 0.32	31.60 ± 0.16	31.01 ± 0.36	30.98 ± 0.29	34.77 ± 0.15	34.80 ± 0.24	34.80 ± 0.27	39.05 ± 0.35	30.89 ± 0.24
524288	11.74 ± 0.33	11.72 ± 0.32	11.79 ± 0.35	16.96 ± 0.18	16.68 ± 0.33	16.62 ± 0.20	19.14 ± 0.17	19.10 ± 0.16	19.12 ± 0.27	21.79 ± 0.31	16.92 ± 0.20
1048576	7.82 ± 0.31	7.82 ± 0.31	7.87 ± 0.31	9.55 ± 0.23	9.45 ± 0.39	9.42 ± 0.17	11.05 ± 0.13	11.07 ± 0.19	11.07 ± 0.17	12.42 ± 0.22	9.93 ± 0.22

(a) Perfectly parallel jobs.

	$g=1$			$g=2$							
p	OptExp	PeriodLB	DPNextFailure	OptExp	OptExpGroup	PeriodLB	DPNextFailureAsap			DPNextFailureSynchro	DPNextFailureGlobal
							min	avg	max		
32768	128.22 ± 0.86	128.22 ± 0.86	128.22 ± 0.76	235.53 ± 0.35	232.45 ± 0.56	232.29 ± 0.62	242.71 ± 0.25	242.59 ± 0.46	242.70 ± 0.34	258.59 ± 1.30	230.65 ± 0.60
65536	69.48 ± 0.60	69.48 ± 0.60	69.64 ± 0.61	121.81 ± 0.18	119.94 ± 0.52	119.87 ± 0.53	127.01 ± 0.32	126.98 ± 0.28	126.96 ± 0.22	137.95 ± 0.86	118.68 ± 0.34
131072	39.78 ± 0.49	39.76 ± 0.60	39.90 ± 0.46	64.58 ± 0.16	63.46 ± 0.49	63.40 ± 0.34	69.45 ± 0.27	69.42 ± 0.23	69.40 ± 0.24	76.37 ± 0.53	62.82 ± 0.33
262144	24.81 ± 0.41	24.75 ± 0.41	24.90 ± 0.39	35.76 ± 0.19	35.07 ± 0.38	35.05 ± 0.26	39.43 ± 0.31	39.42 ± 0.39	39.35 ± 0.16	44.23 ± 0.40	34.97 ± 0.28
524288	17.83 ± 0.37	17.83 ± 0.35	17.90 ± 0.39	21.40 ± 0.21	21.05 ± 0.42	20.97 ± 0.26	24.12 ± 0.19	24.13 ± 0.29	24.13 ± 0.25	27.47 ± 0.38	21.33 ± 0.28
1048576	15.95 ± 0.48	15.92 ± 0.48	16.02 ± 0.46	14.54 ± 0.32	14.38 ± 0.55	14.33 ± 0.20	16.83 ± 0.25	16.77 ± 0.15	16.86 ± 0.30	18.95 ± 0.28	15.07 ± 0.27

(b) Generic parallel jobs.

	$g=1$			$g=2$							
p	OptExp	PeriodLB	DPNextFailure	OptExp	OptExpGroup	PeriodLB	DPNextFailureAsap			DPNextFailureSynchro	DPNextFailureGlobal
							min	avg	max		
32768	124.44 ± 0.83	124.44 ± 0.83	124.49 ± 0.87	232.14 ± 0.34	229.18 ± 0.57	228.95 ± 0.60	239.16 ± 0.47	239.27 ± 0.37	239.20 ± 0.56	254.78 ± 1.28	227.36 ± 0.59
65536	65.46 ± 0.60	65.46 ± 0.60	65.64 ± 0.58	118.26 ± 0.17	116.43 ± 0.49	116.39 ± 0.51	123.34 ± 0.42	123.25 ± 0.23	123.29 ± 0.29	134.02 ± 0.84	115.22 ± 0.35
131072	35.38 ± 0.51	35.36 ± 0.56	35.48 ± 0.47	60.86 ± 0.17	59.80 ± 0.49	59.75 ± 0.45	65.42 ± 0.25	65.39 ± 0.31	65.35 ± 0.17	71.98 ± 0.52	59.19 ± 0.34
262144	19.86 ± 0.37	19.77 ± 0.35	19.87 ± 0.34	31.78 ± 0.16	31.15 ± 0.36	31.15 ± 0.31	35.00 ± 0.25	34.97 ± 0.16	35.02 ± 0.27	39.25 ± 0.36	31.06 ± 0.25
524288	11.88 ± 0.34	11.88 ± 0.33	11.92 ± 0.35	17.09 ± 0.18	16.81 ± 0.32	16.73 ± 0.19	19.28 ± 0.17	19.32 ± 0.32	19.31 ± 0.25	21.98 ± 0.32	17.05 ± 0.21
1048576	7.95 ± 0.32	7.95 ± 0.32	7.99 ± 0.31	9.65 ± 0.23	9.56 ± 0.40	9.52 ± 0.17	11.16 ± 0.13	11.19 ± 0.21	11.19 ± 0.20	12.55 ± 0.21	10.03 ± 0.22

(c) Numerical kernels.

Table 2: Evaluation of the different heuristics on a platform with **Exponential failures** ($MTBF$ = 125 years) under the constant overhead model.

	$g=1$			$g=2$							
p	OptExp	PeriodLB	DPNextFailure	OptExp	OptExpGroup	PeriodLB	DPNextFailureAsap			DPNextFailureSynchro	DPNextFailureGlobal
							min	avg	max		
65536	123.19 ± 5.87	123.08 ± 6.04	123.62 ± 5.87	185.64 ± 4.80	185.88 ± 7.68	182.76 ± 5.73	218.73 ± 4.51	218.27 ± 4.67	218.31 ± 4.69	236.11 ± 6.68	202.78 ± 7.13
131072	61.89 ± 2.91	61.79 ± 2.49	61.96 ± 2.89	92.79 ± 2.20	92.55 ± 3.60	91.22 ± 3.45	109.71 ± 2.14	109.45 ± 2.11	109.48 ± 2.14	118.43 ± 2.55	101.39 ± 3.43
262144	30.55 ± 1.21	30.55 ± 1.21	30.66 ± 1.14	46.65 ± 1.01	46.69 ± 1.68	45.70 ± 1.50	54.74 ± 1.14	54.58 ± 1.15	54.57 ± 1.15	58.85 ± 1.44	50.90 ± 1.69
524288	15.46 ± 0.58	15.45 ± 0.47	15.51 ± 0.58	23.14 ± 0.60	23.22 ± 0.89	22.89 ± 0.71	27.33 ± 0.60	27.26 ± 0.60	27.26 ± 0.60	29.43 ± 0.55	25.27 ± 0.64
1048576	7.82 ± 0.31	7.82 ± 0.31	7.87 ± 0.31	9.55 ± 0.23	9.45 ± 0.39	9.42 ± 0.17	11.05 ± 0.13	11.07 ± 0.19	11.07 ± 0.17	12.42 ± 0.22	9.93 ± 0.22

(a) Perfectly parallel jobs.

	$g=1$			$g=2$							
p	OptExp	PeriodLB	DPNextFailure	OptExp	OptExpGroup	PeriodLB	DPNextFailureAsap			DPNextFailureSynchro	DPNextFailureGlobal
							min	avg	max		
65536	131.05 ± 6.31	130.99 ± 5.79	131.29 ± 6.10	192.02 ± 4.74	191.80 ± 7.74	188.77 ± 5.90	225.85 ± 4.83	225.77 ± 4.89	225.78 ± 4.89	243.36 ± 6.37	209.88 ± 6.78
131072	69.65 ± 2.83	69.62 ± 2.36	69.82 ± 2.89	99.09 ± 2.18	98.82 ± 3.90	97.18 ± 3.25	116.71 ± 2.26	116.68 ± 2.21	116.68 ± 2.20	126.29 ± 2.76	107.85 ± 4.11
262144	38.61 ± 1.40	38.55 ± 1.29	38.63 ± 1.32	52.79 ± 1.16	52.64 ± 1.82	51.71 ± 1.53	61.83 ± 1.29	61.76 ± 1.35	61.76 ± 1.35	66.41 ± 1.43	57.51 ± 2.03
524288	23.59 ± 0.81	23.55 ± 0.78	23.62 ± 0.78	29.27 ± 0.60	29.27 ± 0.87	28.85 ± 0.74	34.52 ± 0.69	34.47 ± 0.65	34.47 ± 0.65	37.20 ± 0.70	31.90 ± 0.83
1048576	15.95 ± 0.48	15.92 ± 0.48	16.02 ± 0.46	14.54 ± 0.32	14.38 ± 0.55	14.33 ± 0.20	16.83 ± 0.25	16.77 ± 0.15	16.86 ± 0.30	18.95 ± 0.28	15.07 ± 0.27

(b) Generic parallel jobs.

	$g=1$			$g=2$							
p	OptExp	PeriodLB	DPNextFailure	OptExp	OptExpGroup	PeriodLB	DPNextFailureAsap			DPNextFailureSynchro	DPNextFailureGlobal
							min	avg	max		
65536	123.76 ± 5.94	123.60 ± 6.08	124.05 ± 5.98	186.39 ± 4.73	186.73 ± 7.69	183.65 ± 5.74	219.24 ± 4.56	219.10 ± 4.63	219.10 ± 4.63	236.52 ± 6.59	203.63 ± 7.03
131072	62.33 ± 2.83	62.26 ± 2.76	62.37 ± 2.89	93.32 ± 2.17	93.03 ± 3.72	91.67 ± 3.32	110.01 ± 2.22	109.97 ± 2.20	109.95 ± 2.18	118.77 ± 2.60	102.19 ± 3.63
262144	30.83 ± 1.21	30.83 ± 1.21	30.88 ± 1.14	46.91 ± 1.04	46.89 ± 1.71	45.91 ± 1.51	55.05 ± 1.20	54.89 ± 1.20	54.90 ± 1.20	59.21 ± 1.41	51.24 ± 1.82
524288	15.65 ± 0.58	15.65 ± 0.52	15.69 ± 0.59	23.37 ± 0.58	23.44 ± 0.92	23.08 ± 0.70	27.51 ± 0.60	27.48 ± 0.61	27.48 ± 0.61	29.64 ± 0.59	25.45 ± 0.68
1048576	7.95 ± 0.32	7.95 ± 0.32	7.99 ± 0.31	9.65 ± 0.23	9.56 ± 0.40	9.52 ± 0.17	11.16 ± 0.13	11.19 ± 0.21	11.19 ± 0.20	12.55 ± 0.21	10.03 ± 0.22

(c) Numerical kernels.

Table 3: Evaluation of the different heuristics on a platform with **Exponential failures** ($MTBF = 125$ years) under the proportional overhead model.

	$g=1$			$g=2$							
p	OptExp	PeriodLB	DPNextFailure	OptExp	OptExpGroup	PeriodLB	DPNextFailureAsap			DPNextFailureSynchro	DPNextFailureGlobal
							MIN	AVG	MAX		
32768	142.66 ± 1.91	137.19 ± 1.14	137.51 ± 1.22	236.16 ± 0.87	241.59 ± 2.61	236.06 ± 0.82	251.98 ± 1.46	252.17 ± 1.06	252.38 ± 1.52	277.78 ± 1.27	232.66 ± 0.75
65536	80.44 ± 1.45	76.17 ± 0.79	76.37 ± 0.81	122.54 ± 0.85	128.82 ± 2.50	122.49 ± 0.49	133.43 ± 0.49	133.52 ± 0.52	133.53 ± 0.55	150.89 ± 0.91	121.11 ± 0.63
131072	48.93 ± 1.25	44.92 ± 0.64	45.21 ± 0.66	65.51 ± 0.95	72.56 ± 2.60	65.29 ± 0.45	74.34 ± 0.84	74.41 ± 0.94	74.22 ± 0.57	84.39 ± 0.72	65.51 ± 0.56
262144	33.15 ± 1.25	29.16 ± 0.56	29.52 ± 0.59	37.07 ± 0.53	44.13 ± 1.63	36.41 ± 0.33	42.83 ± 0.50	42.76 ± 0.34	42.87 ± 0.88	48.52 ± 0.56	37.85 ± 0.52
524288	27.43 ± 1.45	22.49 ± 0.62	22.72 ± 0.70	23.00 ± 0.58	30.85 ± 1.23	21.98 ± 0.34	26.60 ± 0.89	26.43 ± 0.58	26.43 ± 0.56	28.73 ± 0.38	24.37 ± 0.50
1048576	31.83 ± 1.93	23.67 ± 1.01	23.96 ± 1.04	17.16 ± 0.77	26.73 ± 1.92	15.38 ± 0.43	18.53 ± 0.43	18.52 ± 0.43	18.71 ± 0.98	18.84 ± 0.44	17.98 ± 0.76

(a) Perfectly parallel jobs.

	$g=1$			$g=2$							
p	OptExp	PeriodLB	DPNextFailure	OptExp	OptExpGroup	PeriodLB	DPNextFailureAsap			DPNextFailureSynchro	DPNextFailureGlobal
							MIN	AVG	MAX		
32768	147.35 ± 1.87	141.72 ± 1.18	142.06 ± 1.22	240.03 ± 0.87	245.59 ± 2.66	239.98 ± 0.84	255.91 ± 0.80	256.44 ± 0.96	256.59 ± 0.76	282.32 ± 1.29	236.52 ± 0.69
65536	85.76 ± 1.43	81.10 ± 0.81	81.30 ± 0.78	126.54 ± 0.89	132.99 ± 2.50	126.52 ± 0.52	137.70 ± 0.37	137.74 ± 0.36	137.79 ± 0.35	155.87 ± 1.02	125.06 ± 0.62
131072	55.16 ± 1.29	50.76 ± 0.66	51.13 ± 0.74	69.78 ± 1.00	77.40 ± 2.75	69.54 ± 0.49	79.25 ± 1.06	79.06 ± 0.40	79.01 ± 0.29	89.92 ± 0.77	69.77 ± 0.59
262144	41.62 ± 1.47	36.69 ± 0.68	37.16 ± 0.77	41.84 ± 0.61	50.12 ± 1.71	41.11 ± 0.40	48.43 ± 0.57	48.38 ± 0.40	48.54 ± 0.82	54.77 ± 0.59	42.77 ± 0.53
524288	41.25 ± 1.93	34.07 ± 0.78	34.37 ± 0.86	29.02 ± 0.50	38.49 ± 1.37	27.70 ± 0.34	33.16 ± 0.32	33.22 ± 0.31	33.22 ± 0.33	36.17 ± 0.45	30.66 ± 0.49
1048576	63.14 ± 2.71	47.57 ± 1.38	48.12 ± 1.48	25.99 ± 0.98	39.94 ± 2.13	23.41 ± 0.52	28.47 ± 1.46	28.26 ± 1.02	28.18 ± 0.46	28.70 ± 0.51	27.28 ± 0.73

(b) Generic parallel jobs.

	$g=1$			$g=2$							
p	OptExp	PeriodLB	DPNextFailure	OptExp	OptExpGroup	PeriodLB	DPNextFailureAsap			DPNextFailureSynchro	DPNextFailureGlobal
							MIN	AVG	MAX		
32768	143.09 ± 1.86	137.57 ± 1.14	137.93 ± 1.27	236.61 ± 0.88	242.03 ± 2.62	236.51 ± 0.85	252.38 ± 0.71	252.74 ± 0.93	252.92 ± 1.29	278.28 ± 1.32	233.11 ± 0.74
65536	80.83 ± 1.45	76.51 ± 0.79	76.67 ± 0.78	122.85 ± 0.84	129.19 ± 2.47	122.83 ± 0.50	133.61 ± 0.34	133.80 ± 0.47	133.75 ± 0.46	151.34 ± 0.90	121.45 ± 0.63
131072	49.15 ± 1.29	45.14 ± 0.62	45.39 ± 0.64	65.77 ± 0.94	72.96 ± 2.63	65.53 ± 0.45	74.52 ± 0.68	74.57 ± 0.62	74.71 ± 1.28	84.73 ± 0.71	65.74 ± 0.54
262144	33.37 ± 1.27	29.36 ± 0.60	29.70 ± 0.63	37.27 ± 0.54	44.42 ± 1.61	36.59 ± 0.36	43.14 ± 0.76	43.04 ± 0.51	43.07 ± 0.82	48.75 ± 0.51	38.10 ± 0.48
524288	27.73 ± 1.45	22.72 ± 0.64	22.99 ± 0.75	23.18 ± 0.58	31.09 ± 1.22	22.13 ± 0.32	26.47 ± 0.29	26.59 ± 0.35	26.64 ± 0.48	28.95 ± 0.38	24.53 ± 0.53
1048576	32.34 ± 1.96	24.00 ± 1.06	24.37 ± 1.09	17.34 ± 0.78	27.07 ± 2.02	15.56 ± 0.41	18.74 ± 0.44	18.68 ± 0.41	18.75 ± 0.46	19.02 ± 0.42	18.23 ± 0.68

(c) Numerical kernels.

Table 4: Evaluation of the different heuristics on a platform with **Weibull failures** ($MTBF = 125$ years and $k = 0.70$) under the constant overhead model.

	$g=1$			$g=2$							
p	OptExp	PeriodLB	DPNextFailure	OptExp	OptExpGroup	PeriodLB	DPNextFailureAsap			DPNextFailureSynchro	DPNextFailureGlobal
							MIN	AVG	MAX		
131072	217.06 ± 12.68	167.58 ± 7.60	167.82 ± 7.82	237.38 ± 8.86	350.20 ± 9.66	203.08 ± 8.24	238.97 ± 5.09	242.39 ± 5.46	242.39 ± 5.46	227.19 ± 6.11	222.85 ± 6.33
262144	115.85 ± 7.53	87.20 ± 3.61	87.41 ± 3.84	129.28 ± 6.33	205.92 ± 13.43	106.70 ± 3.79	125.47 ± 3.52	126.01 ± 3.85	126.06 ± 3.83	118.59 ± 4.07	118.00 ± 4.89
524288	59.86 ± 4.36	45.14 ± 2.15	45.36 ± 2.03	66.97 ± 3.79	113.18 ± 9.20	54.61 ± 1.82	64.42 ± 1.44	64.41 ± 1.44	64.40 ± 1.44	60.66 ± 1.93	60.17 ± 2.20
1048576	31.83 ± 1.93	23.67 ± 1.01	23.96 ± 1.04	17.16 ± 0.77	26.73 ± 1.92	15.38 ± 0.43	18.53 ± 0.43	18.52 ± 0.43	18.71 ± 0.98	18.84 ± 0.44	17.98 ± 0.76

(a) Perfectly parallel jobs.

	$g=1$			$g=2$							
p	OptExp	PeriodLB	DPNextFailure	OptExp	OptExpGroup	PeriodLB	DPNextFailureAsap			DPNextFailureSynchro	DPNextFailureGlobal
							MIN	AVG	MAX		
131072	241.53 ± 13.87	188.05 ± 7.78	188.28 ± 8.08	247.34 ± 5.93	351.98 ± 10.85	211.96 ± 5.08	252.06 ± 3.99	255.12 ± 3.11	255.12 ± 3.11	239.30 ± 4.57	232.91 ± 5.22
262144	143.31 ± 8.11	108.86 ± 3.98	109.18 ± 4.25	144.93 ± 6.64	228.57 ± 12.62	120.47 ± 4.12	141.29 ± 3.54	142.12 ± 3.66	142.10 ± 3.67	133.53 ± 3.84	132.56 ± 4.34
524288	90.43 ± 5.74	67.80 ± 2.38	68.04 ± 2.58	83.63 ± 4.15	140.11 ± 10.47	68.58 ± 2.13	80.41 ± 1.79	80.80 ± 1.71	80.77 ± 1.74	76.21 ± 2.09	75.59 ± 2.54
1048576	63.14 ± 2.71	47.57 ± 1.38	48.12 ± 1.48	25.99 ± 0.98	39.94 ± 2.13	23.41 ± 0.52	28.47 ± 1.46	28.26 ± 1.02	28.18 ± 0.46	28.70 ± 0.51	27.28 ± 0.73

(b) Generic parallel jobs.

	$g=1$			$g=2$							
p	OptExp	PeriodLB	DPNextFailure	OptExp	OptExpGroup	PeriodLB	DPNextFailureAsap			DPNextFailureSynchro	DPNextFailureGlobal
							MIN	AVG	MAX		
131072	217.99 ± 12.89	168.14 ± 7.49	168.34 ± 7.78	237.60 ± 9.32	349.39 ± 10.05	204.15 ± 7.91	239.94 ± 4.67	243.39 ± 5.39	243.41 ± 5.41	228.14 ± 5.83	223.56 ± 6.34
262144	116.18 ± 7.16	87.85 ± 3.52	88.00 ± 3.78	129.85 ± 6.32	206.03 ± 13.37	107.25 ± 3.84	125.80 ± 3.67	126.77 ± 3.78	126.80 ± 3.77	119.17 ± 4.04	118.53 ± 4.84
524288	60.63 ± 4.37	45.61 ± 2.06	45.85 ± 2.02	67.27 ± 3.76	113.66 ± 9.18	54.99 ± 1.88	64.67 ± 1.53	64.85 ± 1.51	64.86 ± 1.49	61.13 ± 1.88	60.70 ± 2.22
1048576	32.34 ± 1.96	24.00 ± 1.06	24.37 ± 1.09	17.34 ± 0.78	27.07 ± 2.02	15.56 ± 0.41	18.74 ± 0.44	18.68 ± 0.41	18.75 ± 0.46	19.02 ± 0.42	18.23 ± 0.68

(c) Numerical kernels.

Table 5: Evaluation of the different heuristics on a platform with **Weibull failures** ($MTBF = 125$ years and $k = 0.70$) under the proportional overhead model.

	$g = 1$			$g = 2$							
p	OptExp	PeriodLB	DPNextFailure	OptExp	OptExpGroup	PeriodLB	DPNextFailureAsap			DPNextFailureSynchro	DPNextFailureGlobal
							MIN	AVG	MAX		
4096	124.15 ± 1.33	124.07 ± 1.42	120.94 ± 1.45	203.13 ± 0.58	198.94 ± 1.41	198.95 ± 0.90	221.99 ± 3.39	218.18 ± 4.22	216.48 ± 4.02	234.28 ± 1.75	197.38 ± 0.87
8192	72.71 ± 0.94	72.71 ± 0.94	69.80 ± 0.92	109.17 ± 0.74	107.04 ± 1.49	106.88 ± 0.69	122.11 ± 2.80	119.21 ± 3.65	117.79 ± 2.63	131.25 ± 1.58	106.44 ± 0.79
16384	46.38 ± 0.82	45.97 ± 0.94	44.12 ± 0.80	61.41 ± 0.87	60.27 ± 1.43	60.23 ± 0.59	71.23 ± 2.52	68.32 ± 1.75	67.34 ± 1.67	72.44 ± 0.67	60.08 ± 0.65
32768	32.88 ± 0.78	32.53 ± 0.84	31.76 ± 0.88	36.92 ± 0.39	35.91 ± 0.59	35.74 ± 0.50	44.06 ± 1.63	42.26 ± 1.54	41.55 ± 1.32	43.76 ± 0.56	36.71 ± 0.51
65536	29.12 ± 0.96	28.12 ± 1.00	28.22 ± 1.15	24.53 ± 0.36	23.82 ± 0.45	23.64 ± 0.45	30.02 ± 1.49	28.43 ± 1.23	28.06 ± 1.24	28.57 ± 0.55	25.62 ± 0.54
131072	40.30 ± 1.70	37.42 ± 1.59	38.75 ± 1.70	19.92 ± 0.45	19.09 ± 0.53	18.94 ± 0.44	23.67 ± 0.91	22.99 ± 1.08	22.74 ± 1.18	22.28 ± 0.58	22.68 ± 0.72

(a) Perfectly parallel jobs.

	$g = 1$			$g = 2$							
p	OptExp	PeriodLB	DPNextFailure	OptExp	OptExpGroup	PeriodLB	DPNextFailureAsap			DPNextFailureSynchro	DPNextFailureGlobal
							MIN	AVG	MAX		
4096	124.27 ± 1.32	124.22 ± 1.41	121.15 ± 1.63	203.23 ± 0.57	199.05 ± 1.40	199.02 ± 0.89	222.15 ± 4.37	218.33 ± 3.89	216.09 ± 3.64	234.33 ± 1.78	197.50 ± 0.87
8192	72.88 ± 0.93	72.88 ± 0.93	69.95 ± 0.92	109.27 ± 0.74	107.14 ± 1.49	107.00 ± 0.67	122.45 ± 3.50	119.00 ± 3.09	117.90 ± 3.01	131.26 ± 1.44	106.57 ± 0.72
16384	46.57 ± 0.80	46.17 ± 0.93	44.33 ± 0.82	61.54 ± 0.87	60.37 ± 1.44	60.35 ± 0.47	70.85 ± 1.97	68.71 ± 1.77	67.99 ± 1.70	72.69 ± 0.77	60.21 ± 0.70
32768	33.13 ± 0.79	32.76 ± 0.85	31.97 ± 0.88	37.07 ± 0.40	36.08 ± 0.58	35.91 ± 0.51	44.31 ± 1.74	42.74 ± 1.63	41.87 ± 1.37	43.91 ± 0.58	36.88 ± 0.52
65536	29.94 ± 1.01	28.56 ± 1.02	28.67 ± 1.18	24.71 ± 0.36	23.98 ± 0.44	23.77 ± 0.45	30.21 ± 1.74	28.72 ± 1.37	28.47 ± 1.55	28.79 ± 0.58	25.78 ± 0.54
131072	41.80 ± 1.71	38.82 ± 1.78	40.27 ± 1.85	20.40 ± 0.48	19.39 ± 0.54	19.15 ± 0.60	24.05 ± 1.07	23.29 ± 1.11	22.95 ± 1.18	22.59 ± 0.59	22.97 ± 0.74

(b) Generic parallel jobs.

	$g = 1$			$g = 2$							
p	OptExp	PeriodLB	DPNextFailure	OptExp	OptExpGroup	PeriodLB	DPNextFailureAsap			DPNextFailureSynchro	DPNextFailureGlobal
							MIN	AVG	MAX		
4096	124.28 ± 1.32	124.23 ± 1.40	121.01 ± 1.48	203.37 ± 0.57	199.19 ± 1.40	199.15 ± 0.89	221.89 ± 3.01	218.15 ± 3.97	216.89 ± 3.71	234.46 ± 1.61	197.66 ± 0.82
8192	72.90 ± 0.92	72.90 ± 0.92	70.03 ± 0.94	109.33 ± 0.73	107.21 ± 1.49	107.01 ± 0.68	122.93 ± 2.61	119.05 ± 3.12	118.16 ± 3.06	131.47 ± 1.58	106.73 ± 0.89
16384	46.58 ± 0.80	46.19 ± 0.94	44.28 ± 0.83	61.56 ± 0.85	60.40 ± 1.42	60.36 ± 0.59	71.36 ± 1.90	68.65 ± 1.92	67.83 ± 1.58	72.60 ± 0.68	60.22 ± 0.66
32768	32.99 ± 0.80	32.64 ± 0.84	31.86 ± 0.88	37.05 ± 0.39	36.00 ± 0.60	35.85 ± 0.51	44.58 ± 1.37	42.34 ± 1.56	41.88 ± 1.49	43.90 ± 0.54	36.84 ± 0.50
65536	29.63 ± 0.98	28.24 ± 0.99	28.40 ± 1.17	24.63 ± 0.36	23.90 ± 0.45	23.72 ± 0.45	29.87 ± 1.39	28.56 ± 1.27	28.07 ± 1.26	28.68 ± 0.57	25.73 ± 0.51
131072	40.73 ± 1.69	37.86 ± 1.66	39.14 ± 1.73	20.22 ± 0.46	19.23 ± 0.53	18.99 ± 0.60	23.89 ± 0.98	23.18 ± 1.17	22.83 ± 1.02	22.41 ± 0.58	22.85 ± 0.77

(c) Numerical kernels.

Table 6: Evaluation of the different heuristics on a platform with failures based on the failure log of **LANL cluster 18** and under under the constant overhead model.

	$g=1$			$g=2$							
p	OptExp	PeriodLB	DPNextFailure	OptExp	OptExpGroup	PeriodLB	DPNextFailureAsap			DPNextFailureSynchro	DPNextFailureGlobal
							min	avg	max		
32768	226.77 ± 12.39	218.36 ± 13.79	211.07 ± 11.97	292.56 ± 11.30	286.19 ± 13.42	285.64 ± 13.78	337.50 ± 20.77	339.30 ± 24.99	352.10 ± 25.20	310.68 ± 15.33	331.38 ± 17.67
65536	89.48 ± 4.12	84.15 ± 4.39	84.09 ± 4.40	119.10 ± 4.48	114.67 ± 4.69	112.43 ± 4.79	134.09 ± 5.64	128.67 ± 5.64	127.85 ± 5.90	123.99 ± 5.89	131.63 ± 7.76
131072	40.30 ± 1.70	37.42 ± 1.59	38.75 ± 1.70	19.92 ± 0.45	19.09 ± 0.53	18.94 ± 0.44	23.67 ± 0.91	22.99 ± 1.08	22.74 ± 1.18	22.28 ± 0.58	22.68 ± 0.72

(a) Perfectly parallel jobs.

	$g=1$			$g=2$							
p	OptExp	PeriodLB	DPNextFailure	OptExp	OptExpGroup	PeriodLB	DPNextFailureAsap			DPNextFailureSynchro	DPNextFailureGlobal
							min	avg	max		
32768	228.66 ± 12.25	219.93 ± 13.89	212.77 ± 12.42	293.69 ± 11.52	287.96 ± 13.35	287.54 ± 13.83	340.51 ± 20.16	342.01 ± 25.19	353.95 ± 25.44	312.25 ± 15.23	332.71 ± 18.34
65536	91.06 ± 4.29	86.16 ± 4.88	85.64 ± 4.55	120.10 ± 4.45	115.60 ± 4.65	113.28 ± 4.74	135.11 ± 5.63	129.59 ± 5.42	129.12 ± 5.76	124.70 ± 5.72	132.50 ± 7.13
131072	41.80 ± 1.71	38.82 ± 1.78	40.27 ± 1.85	20.40 ± 0.48	19.39 ± 0.54	19.15 ± 0.60	24.05 ± 1.07	23.29 ± 1.11	22.95 ± 1.18	22.59 ± 0.59	22.97 ± 0.74

(b) Generic parallel jobs.

	$g=1$			$g=2$							
p	OptExp	PeriodLB	DPNextFailure	OptExp	OptExpGroup	PeriodLB	DPNextFailureAsap			DPNextFailureSynchro	DPNextFailureGlobal
							min	avg	max		
32768	226.94 ± 12.48	219.04 ± 13.36	211.32 ± 12.86	293.58 ± 11.58	287.83 ± 13.45	287.23 ± 13.58	337.58 ± 19.12	340.34 ± 24.77	352.72 ± 25.39	311.75 ± 15.37	332.25 ± 18.59
65536	89.88 ± 4.15	85.12 ± 4.69	84.70 ± 4.42	119.55 ± 4.50	115.18 ± 4.72	112.79 ± 4.85	134.37 ± 5.35	129.21 ± 5.85	128.63 ± 5.90	124.56 ± 5.92	132.28 ± 8.05
131072	40.73 ± 1.69	37.86 ± 1.66	39.14 ± 1.73	20.22 ± 0.46	19.23 ± 0.53	18.99 ± 0.60	23.89 ± 0.98	23.18 ± 1.17	22.83 ± 1.02	22.41 ± 0.58	22.85 ± 0.77

(c) Numerical kernels.

Table 7: Evaluation of the different heuristics on a platform with failures based on the failure log of **LANL cluster 18** and under the proportional overhead model.

	$g = 1$			$g = 2$							
p	OptExp	PeriodLB	DPNextFailure	OptExp	OptExpGroup	PeriodLB	DPNextFailureAsap			DPNextFailureSynchro	DPNextFailureGlobal
							MIN	AVG	MAX		
4096	123.81 ± 1.28	123.67 ± 1.41	120.13 ± 1.46	203.31 ± 0.55	198.87 ± 1.39	198.97 ± 0.92	224.81 ± 5.58	219.66 ± 3.81	216.53 ± 4.32	233.54 ± 1.80	196.71 ± 0.83
8192	72.18 ± 0.82	71.84 ± 0.96	69.28 ± 0.95	109.42 ± 0.77	107.05 ± 1.36	106.98 ± 0.57	122.92 ± 2.72	119.69 ± 1.92	118.55 ± 2.32	130.38 ± 1.57	106.39 ± 0.82
16384	45.50 ± 0.74	45.27 ± 0.79	43.57 ± 0.78	61.36 ± 0.87	60.08 ± 1.37	59.88 ± 0.46	71.57 ± 2.56	68.21 ± 1.98	67.59 ± 2.00	72.15 ± 0.63	59.81 ± 0.59
32768	32.54 ± 0.76	31.63 ± 0.84	31.20 ± 0.96	36.97 ± 0.29	36.00 ± 0.42	35.96 ± 0.42	44.26 ± 2.14	41.86 ± 1.33	41.32 ± 1.21	43.54 ± 0.54	36.84 ± 0.53
65536	28.74 ± 1.10	27.57 ± 1.15	27.54 ± 1.25	24.18 ± 0.40	23.06 ± 0.50	23.01 ± 0.48	29.78 ± 1.08	28.40 ± 1.39	27.99 ± 1.40	28.22 ± 0.56	25.21 ± 0.57
131072	39.69 ± 1.47	36.93 ± 1.85	38.08 ± 2.12	19.26 ± 0.49	18.24 ± 0.52	17.92 ± 0.57	22.95 ± 1.24	21.68 ± 0.94	21.32 ± 0.85	21.48 ± 0.72	21.45 ± 0.75

(a) Perfectly parallel jobs.

	$g = 1$			$g = 2$							
p	OptExp	PeriodLB	DPNextFailure	OptExp	OptExpGroup	PeriodLB	DPNextFailureAsap			DPNextFailureSynchro	DPNextFailureGlobal
							MIN	AVG	MAX		
4096	123.93 ± 1.30	123.80 ± 1.44	120.24 ± 1.51	203.42 ± 0.55	199.00 ± 1.40	199.07 ± 0.91	224.16 ± 4.98	219.79 ± 3.74	216.99 ± 5.19	233.77 ± 1.86	196.76 ± 0.82
8192	72.33 ± 0.81	72.01 ± 0.95	69.43 ± 0.91	109.52 ± 0.76	107.20 ± 1.38	107.09 ± 0.58	123.09 ± 2.93	120.09 ± 1.92	118.02 ± 2.04	130.28 ± 1.52	106.34 ± 0.82
16384	45.73 ± 0.68	45.48 ± 0.78	43.79 ± 0.78	61.47 ± 0.88	60.21 ± 1.39	60.02 ± 0.47	71.60 ± 2.12	68.41 ± 2.00	67.49 ± 1.91	72.27 ± 0.63	59.89 ± 0.55
32768	32.82 ± 0.77	31.90 ± 0.85	31.49 ± 0.94	37.11 ± 0.29	36.07 ± 0.41	36.07 ± 0.41	44.16 ± 2.01	42.09 ± 1.65	41.51 ± 1.34	43.70 ± 0.59	36.98 ± 0.53
65536	29.27 ± 1.11	28.04 ± 1.20	27.98 ± 1.26	24.39 ± 0.40	23.26 ± 0.51	23.19 ± 0.50	30.06 ± 1.07	28.56 ± 1.43	28.16 ± 1.40	28.44 ± 0.57	25.40 ± 0.59
131072	40.93 ± 1.57	38.12 ± 1.79	39.26 ± 2.20	19.59 ± 0.48	18.54 ± 0.52	18.23 ± 0.56	23.35 ± 1.14	22.09 ± 0.96	21.56 ± 0.89	21.82 ± 0.73	21.90 ± 0.84

(b) Generic parallel jobs.

	$g = 1$			$g = 2$							
p	OptExp	PeriodLB	DPNextFailure	OptExp	OptExpGroup	PeriodLB	DPNextFailureAsap			DPNextFailureSynchro	DPNextFailureGlobal
							MIN	AVG	MAX		
4096	124.01 ± 1.31	123.90 ± 1.44	120.39 ± 1.51	203.53 ± 0.56	199.13 ± 1.39	199.20 ± 0.90	224.18 ± 4.70	219.70 ± 3.98	217.95 ± 4.53	234.13 ± 1.92	196.94 ± 0.74
8192	72.31 ± 0.82	72.00 ± 0.95	69.42 ± 0.96	109.59 ± 0.76	107.24 ± 1.33	107.14 ± 0.56	123.37 ± 2.12	120.10 ± 2.08	118.64 ± 2.19	130.39 ± 1.53	106.51 ± 0.74
16384	45.68 ± 0.70	45.42 ± 0.79	43.73 ± 0.78	61.48 ± 0.86	60.21 ± 1.36	60.02 ± 0.46	71.17 ± 2.71	68.95 ± 1.83	67.73 ± 2.00	72.28 ± 0.65	59.87 ± 0.55
32768	32.69 ± 0.77	31.78 ± 0.85	31.38 ± 0.96	37.08 ± 0.29	36.07 ± 0.41	36.05 ± 0.36	44.01 ± 1.59	41.93 ± 1.28	41.54 ± 1.31	43.63 ± 0.54	36.92 ± 0.50
65536	28.98 ± 1.09	27.77 ± 1.18	27.70 ± 1.23	24.30 ± 0.40	23.17 ± 0.50	23.10 ± 0.49	29.94 ± 0.99	28.46 ± 1.33	28.14 ± 1.41	28.36 ± 0.57	25.33 ± 0.59
131072	40.00 ± 1.50	37.25 ± 1.82	38.43 ± 2.22	19.40 ± 0.48	18.34 ± 0.51	18.03 ± 0.57	23.21 ± 1.11	21.80 ± 0.87	21.37 ± 0.84	21.60 ± 0.73	21.60 ± 0.84

(c) Numerical kernels.

Table 8: Evaluation of the different heuristics on a platform with failures based on the failure log of **LANL cluster 19** and under the constant overhead model.

	$g = 1$			$g = 2$							
p	OptExp	PeriodLB	DPNextFailure	OptExp	OptExpGroup	PeriodLB	DPNextFailureAsap			DPNextFailureSynchro	DPNextFailureGlobal
							min	avg	max		
32768	202.75 ± 10.31	194.84 ± 9.98	191.61 ± 12.91	282.16 ± 15.71	270.90 ± 14.84	270.90 ± 14.84	319.55 ± 15.68	311.87 ± 16.09	319.79 ± 21.61	299.02 ± 16.21	329.11 ± 20.91
65536	83.80 ± 4.13	77.91 ± 4.04	77.98 ± 4.38	109.64 ± 4.65	105.29 ± 4.59	103.92 ± 5.44	125.88 ± 10.22	119.30 ± 5.83	117.21 ± 5.33	114.52 ± 5.61	121.70 ± 6.52
131072	39.69 ± 1.47	36.93 ± 1.85	38.08 ± 2.12	19.26 ± 0.49	18.24 ± 0.52	17.92 ± 0.57	22.95 ± 1.24	21.68 ± 0.94	21.32 ± 0.85	21.48 ± 0.72	21.45 ± 0.75

(a) Perfectly parallel jobs.

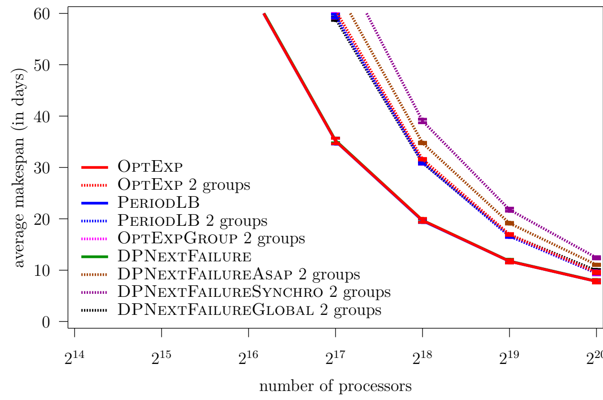
	$g = 1$			$g = 2$							
p	OptExp	PeriodLB	DPNextFailure	OptExp	OptExpGroup	PeriodLB	DPNextFailureAsap			DPNextFailureSynchro	DPNextFailureGlobal
							min	avg	max		
32768	205.09 ± 10.62	196.63 ± 10.59	192.73 ± 13.45	283.03 ± 15.83	272.17 ± 14.98	272.17 ± 14.98	320.80 ± 16.98	314.54 ± 16.59	321.32 ± 21.78	300.54 ± 16.25	329.95 ± 20.76
65536	85.36 ± 4.23	79.36 ± 4.30	79.50 ± 4.66	110.56 ± 4.61	108.24 ± 4.90	104.74 ± 5.04	125.85 ± 9.64	120.23 ± 7.00	118.34 ± 5.34	115.56 ± 5.61	123.05 ± 6.62
131072	40.93 ± 1.57	38.12 ± 1.79	39.26 ± 2.20	19.59 ± 0.48	18.54 ± 0.52	18.23 ± 0.56	23.35 ± 1.14	22.09 ± 0.96	21.56 ± 0.89	21.82 ± 0.73	21.90 ± 0.84

(b) Generic parallel jobs.

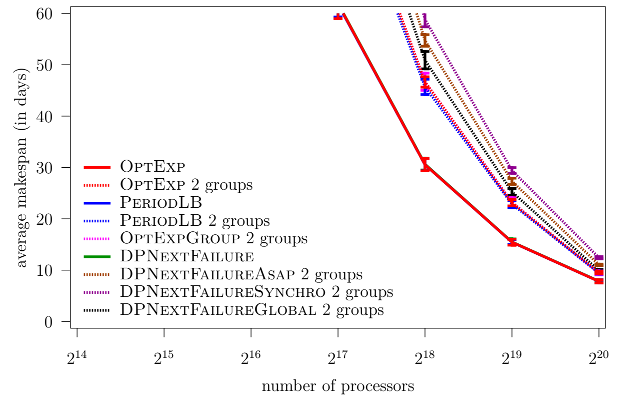
	$g = 1$			$g = 2$							
p	OptExp	PeriodLB	DPNextFailure	OptExp	OptExpGroup	PeriodLB	DPNextFailureAsap			DPNextFailureSynchro	DPNextFailureGlobal
							min	avg	max		
32768	204.05 ± 10.33	196.02 ± 10.15	192.61 ± 12.83	283.17 ± 15.92	272.25 ± 15.13	272.25 ± 15.13	322.35 ± 16.87	313.46 ± 16.92	322.73 ± 23.52	300.28 ± 16.44	329.87 ± 20.83
65536	84.39 ± 4.19	78.42 ± 4.07	78.57 ± 4.51	110.15 ± 4.60	107.79 ± 4.85	104.24 ± 5.41	125.56 ± 9.92	120.07 ± 6.93	117.84 ± 5.47	115.05 ± 5.59	122.37 ± 6.45
131072	40.00 ± 1.50	37.25 ± 1.82	38.43 ± 2.22	19.40 ± 0.48	18.34 ± 0.51	18.03 ± 0.57	23.21 ± 1.11	21.80 ± 0.87	21.37 ± 0.84	21.60 ± 0.73	21.60 ± 0.84

(c) Numerical kernels.

Table 9: Evaluation of the different heuristics on a platform with failures based on the failure log of **LANL cluster 19** and under the proportional overhead model.

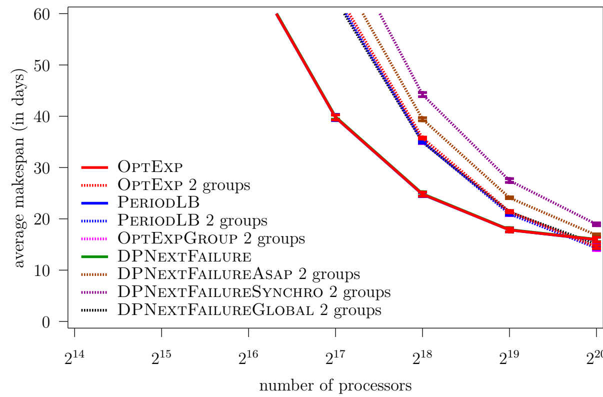


a) constant overhead model

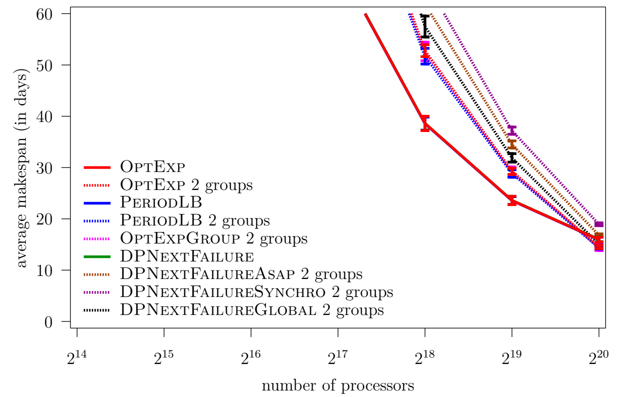


b) proportional overhead model

(1) Perfectly parallel jobs.

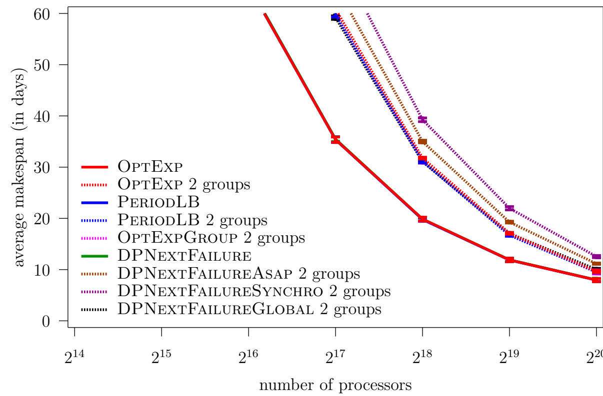


a) constant overhead model

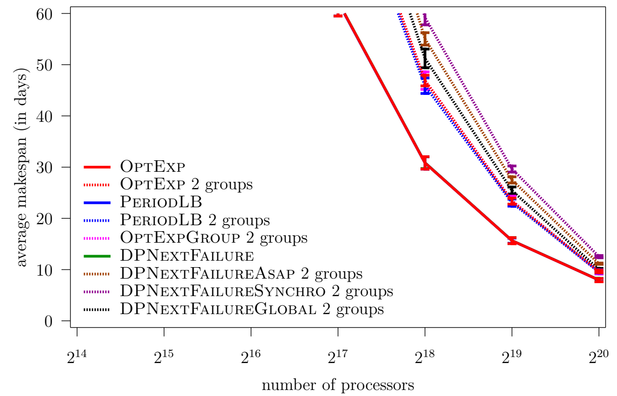


b) proportional overhead model

(2) Generic parallel jobs.



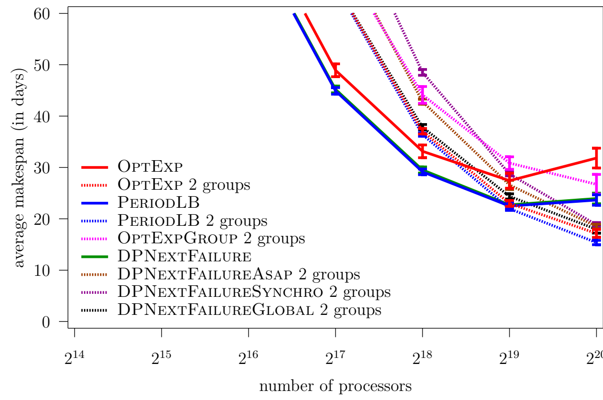
a) constant overhead model



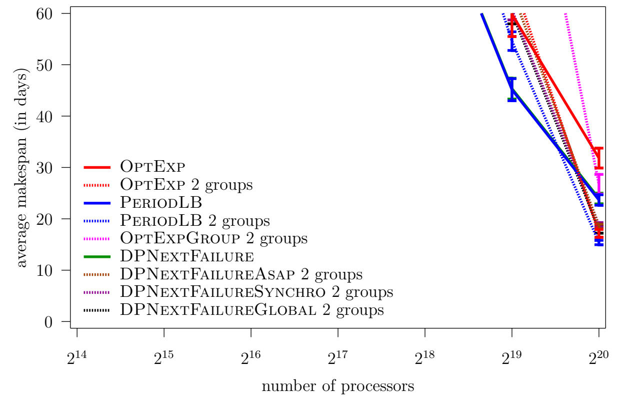
b) proportional overhead model

(3) Numerical kernels.

Figure 9: Evaluation of the different heuristics on a platform with **Exponential failures** ($MTBF = 125$ years).

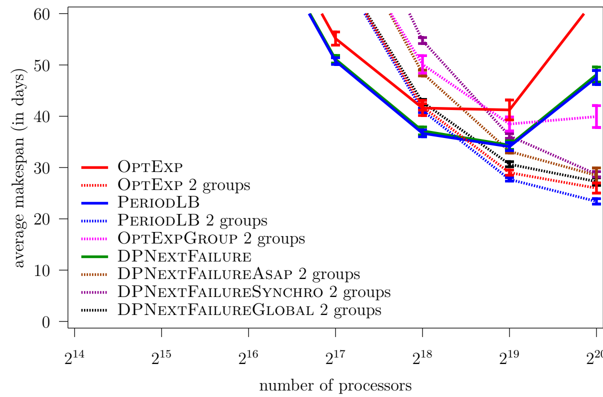


a) constant overhead model

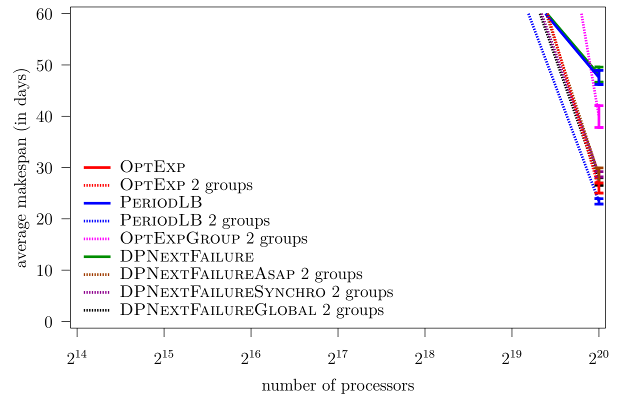


b) proportional overhead model

(1) Perfectly parallel jobs.

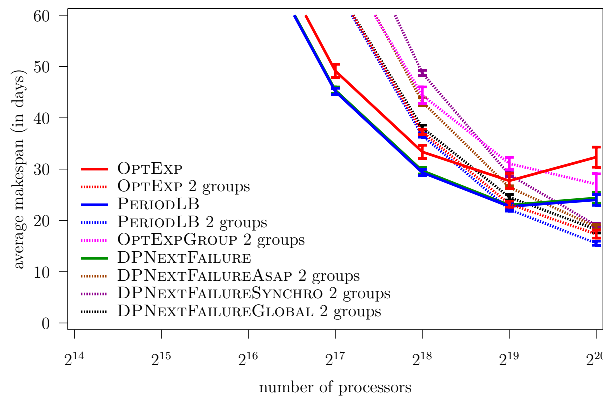


a) constant overhead model

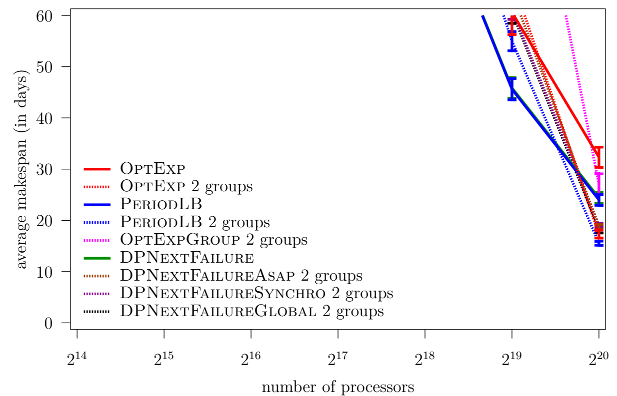


b) proportional overhead model

(2) Generic parallel jobs.



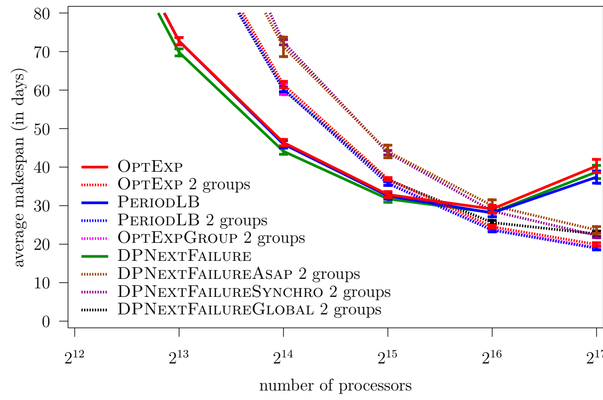
a) constant overhead model



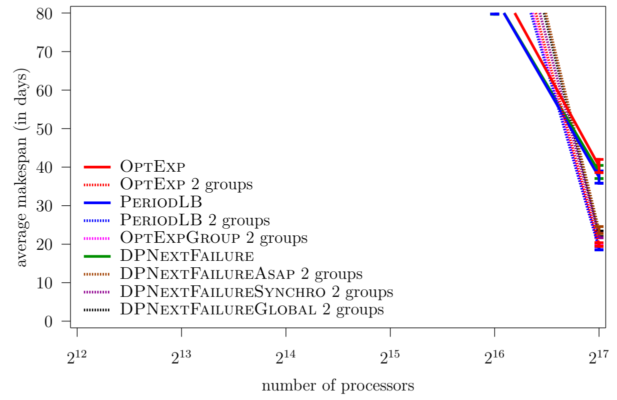
b) proportional overhead model

(3) Numerical kernels.

Figure 10: Evaluation of the different heuristics on a platform with **Weibull failures** ($MTBF =$ 125 years, and $k = 0.70$).

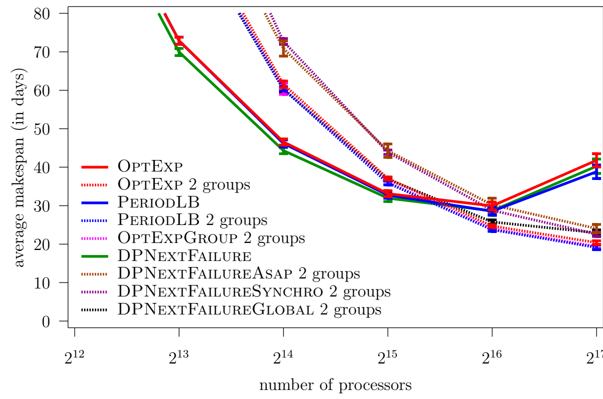


a) constant overhead model

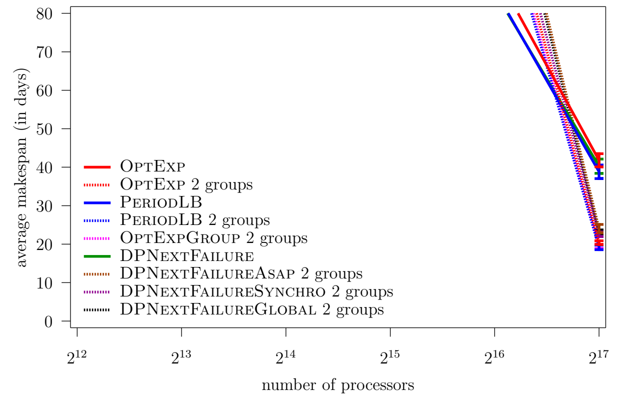


b) proportional overhead model

(1) Perfectly parallel jobs.

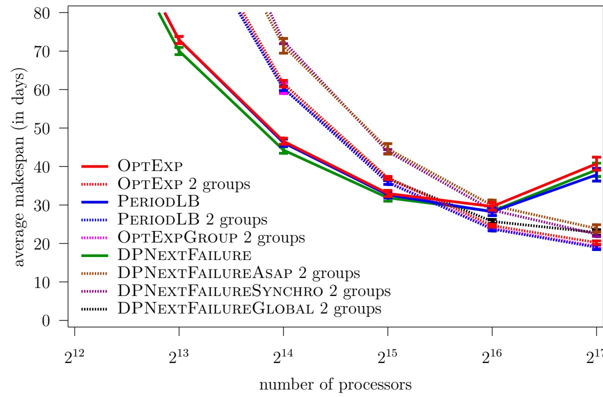


a) constant overhead model

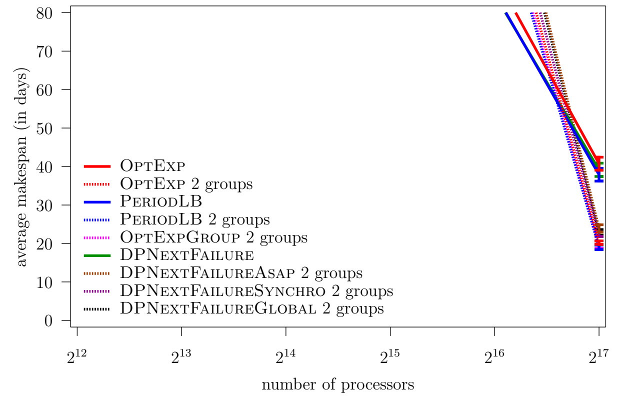


b) proportional overhead model

(2) Generic parallel jobs.



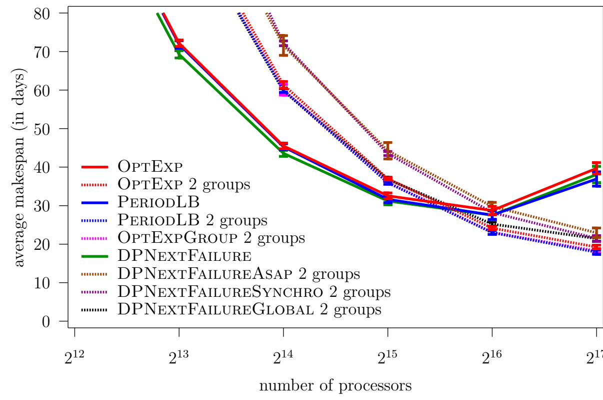
a) constant overhead model



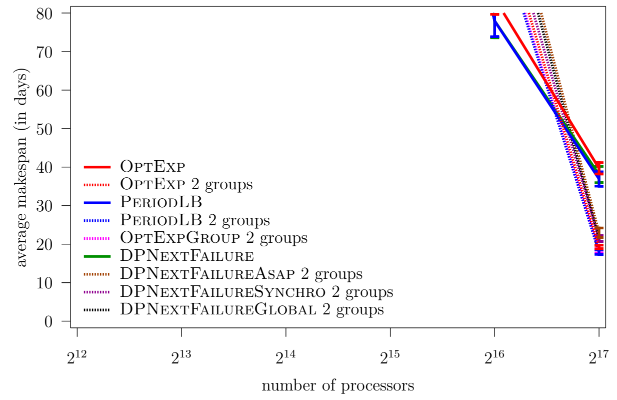
b) proportional overhead model

(3) Numerical kernels.

Figure 11: Evaluation of the different heuristics on a platform with failures based on the failure log of **LANL cluster 18**.

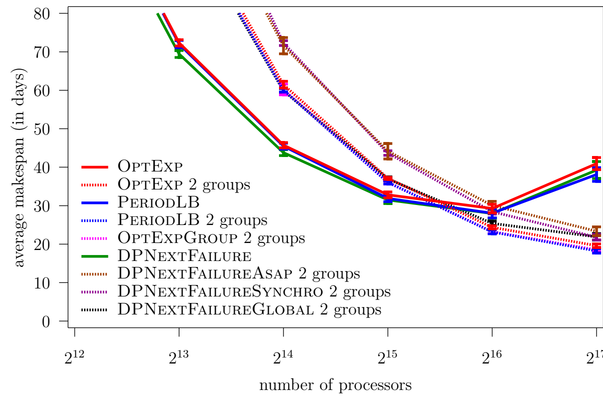


a) constant overhead model

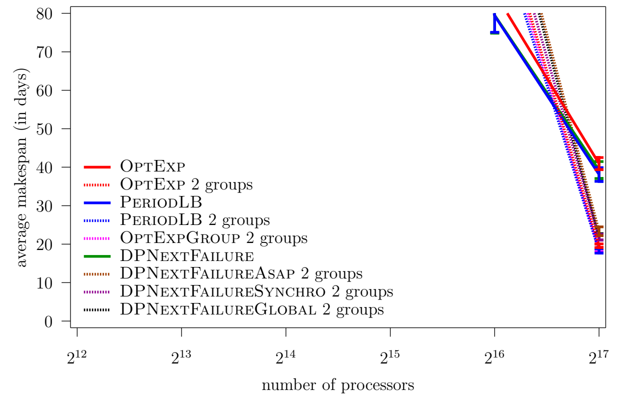


b) proportional overhead model

(1) Perfectly parallel jobs.

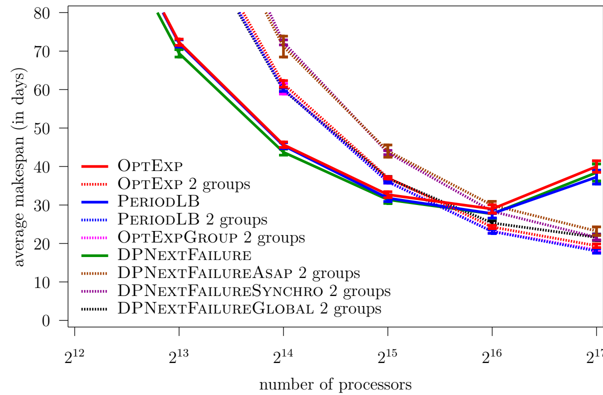


a) constant overhead model

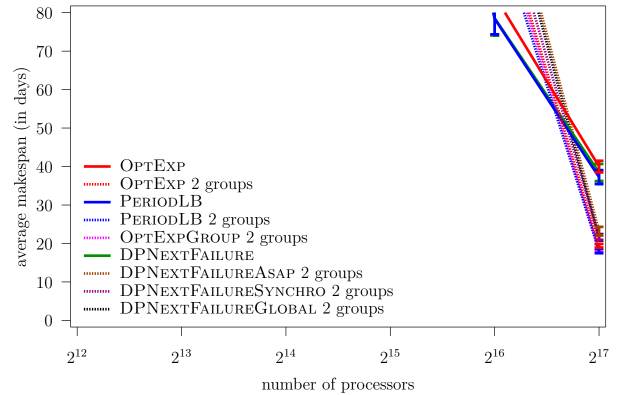


b) proportional overhead model

(2) Generic parallel jobs.



a) constant overhead model



b) proportional overhead model

(3) Numerical kernels.

Figure 12: Evaluation of the different heuristics on a platform with failures based on the failure log of **LANL cluster 19**.