Comparing the performance of rigid, moldable and grid-shaped applications on failure-prone HPC platforms

Valentin Le Fèvre, Thomas Herault, Yves Robert, Aurelien Bouteiller, Atsushi Hori, George Bosilca, Jack Dongarra

Version of Record: https://www.sciencedirect.com/science/article/pii/S0167819118302230
Manuscript_f41d63531937af3cb485f9a173738937

Comparing the performance of rigid, moldable and grid-shaped applications on failure-prone HPC platforms

Valentin Le Fèvre[a], Thomas Herault[b], Yves Robert[a,b,*], Aurelien Bouteiller[b], Atsushi Hori[c], George Bosilca[b], Jack Dongarra[b,d]

[a] *Ecole Normale Superieure de Lyon, France*
[b] *University of Tennessee Knoxville, USA*
[c] *RIKEN Center for Computational Science, Japan*
[d] *University of Manchester, UK*

Abstract

This paper compares the performance of different approaches to tolerate failures for applications executing on large-scale failure-prone platforms. We study (i) RIGID applications, which use a constant number of processors throughout execution; (ii) MOLDABLE applications, which can use a different number of processors after each restart following a fail-stop error; and (iii) GRIDSHAPED applications, which are moldable applications restricted to use rectangular processor grids (such as many dense linear algebra kernels). We start with checkpoint/restart, the de-facto standard approach. For each application type, we compute the optimal number of failures (i.e. that maximizes the yield of the application) to tolerate before relinquishing the current allocation and waiting until a new resource can be allocated, and we determine the optimal yield that can be achieved. For GRIDSHAPED applications, we also investigate *Application Based Fault Tolerance* (ABFT) techniques and perform the same analysis, computing the optimal number of failures to tolerate and the associated yield. We instantiate our performance model with realistic applicative scenarios and make it publicly available for further usage. We show that using spare nodes grants a much better yield than currently used strategies that restart after each failure. Moreover, the yield is similar for RIGID, MOLDABLE and GRIDSHAPED applications, while the optimal number of failures to tolerate is very high, even for a short wait time in between allocations. Finally, MOLDABLE applications have the advantage to restart less frequently than RIGID applications.

1. Introduction

Consider a long-running job that requests N processors from the batch scheduler. Resilience to fail-stop errors[1] is typically provided by a Checkpoint/Restart

[☆] A preliminary version of this paper has appeared in the Proceedings of the 2018 *Resilience* workshop co-located with EuroPar.
[*] Corresponding author
[1] We use the terms *fail-stop error* and *failure* indifferently.

(C/R) mechanism, the de-facto standard approach for High-Performance Computing (HPC) applications. After each failure on one of the nodes used by the application, the application restarts from the last checkpoint but the number of available processors decreases, assuming the application can continue execution after a failure (e.g., using ULFM [4]). Until which point should the execution proceed before requesting a new allocation with N fresh nodes from the batch scheduler?

The answer depends upon the nature of the application. For a RIGID application, the number of processors must remain constant throughout the execution. The question is then to decide the number F of processors (out of the N available initially) that will be used as spares. With F spares, the application can tolerate F failures. The application always executes with $N - F$ processors: after each failure, then it restarts from the last checkpoint and continues executing with $N - F$ processors, the faulty processor having been replaced by a spare. After F failures, the application stops when the $(F + 1)$st failure strikes, and relinquishes the current allocation. It then asks for a new allocation with N processors, which takes a *wait time*, D, to start (as other applications are most likely using the platform concurrently). The optimal value of F obviously depends on the value of D, in addition to the application and resilience parameters. The wait time typically ranges from several hours to several days if the platform is over-subscribed (up to 10 days for large applications on the K-computer [29]). The metric to optimize here is the (expected) application yield, which is the fraction of useful work per second, averaged over the N resources, and computed in steady-state mode (expected value for multiple batch allocations of N resources).

For a MOLDABLE application, the problem is different: here we assume that the application can use a different number of processors after each restart. The application starts executing with N processors; after the first failure, the application recovers from the last checkpoint and is able to continue with only $N - 1$ processors, albeit with a slowdown factor $\frac{N-1}{N}$. After how many failures F should the application decide to stop[2] and accept to produce no progress during D, in order to request a new allocation? Again, the metric to optimize is the application yield.

Finally, consider an application which must have a given shape (or a set of given shapes) in terms of processor layout. Typically, these shapes are dictated by the application algorithm. In this paper, we use the example of a GRIDSHAPED application, which is required to execute on a rectangular processor grid whose size can dynamically be chosen. Most dense linear algebra kernels (matrix multiplication, LU, Cholesky and QR factorizations) are GRIDSHAPED applications, and perform more efficiently on square processor grids than on elongated rectangle ones. The application starts with a (logical) square $p \times p$ grid of $N = p^2$ processors. After the first failure, execution continues on a $p \times (p - 1)$ rectangular grid, keeping $p - 1$ processors as spares for the next $p - 1$ failures (Figure 1, b). After p failures, the grid is shrunk again to a

[2]Another limit is induced by the total application memory Mem_{tot}. There must remain at least ℓ live processors such that $Mem_{tot} \leq \ell \times Mem_{ind}$, where Mem_{ind} is the memory of each processor. We ignore this contraint in the paper but it would be straightforward to take it into account.

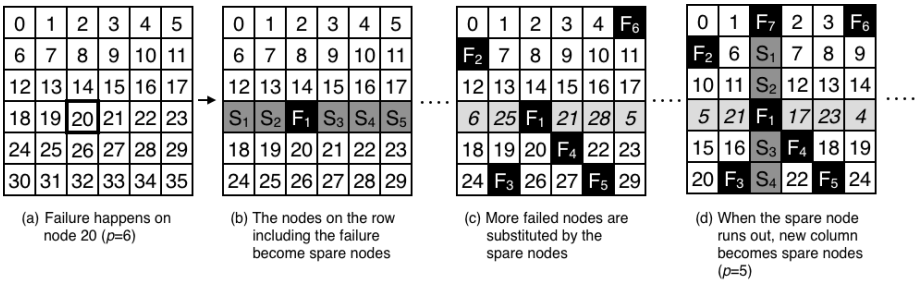


Figure 1: Example of node failures substituted by spare nodes in a 2-D GridShaped application.

$(p-1) \times (p-1)$ square grid (see Figure 1(d)), and so on. We address the same question: after how many failures F should the application stop working on a smaller processor grid and request a new allocation, in order to optimize the application yield?

Many GridShaped applications can also be protected from failures by using *Algorithm-Based Fault Tolerant* techniques (ABFT), instead of Checkpoint/Restart (C/R). ABFT is a widely used approach for linear algebra kernels [18, 5]. We present how we can model ABFT techniques instead of C/R and we perform the same analysis: we compute the optimal number of failures to tolerate before relinquishing the allocation, as well as the associated yield.

Altogether, the major contribution of this paper is to present a detailed performance model and to provide analytical formulas for the expected yield of each application type. We instantiate the model for several applicative scenarios, for which we draw comparisons across application types. Our model is publicly available [25] so that more scenarios can be explored. Notably, the paper quantifies the optimal number of spares for the optimal yield, and the optimal length of a period between two full restarts; it also qualifies how much the yield and total work done within a period are improved by deploying Moldable applications w.r.t. Rigid applications. Finally, for GridShaped applications, the paper compares the use of C/R and ABFT under various frameworks. Our main result is that using spare nodes grants a significantly higher yield for every kind of application, even for short wait times. We also show that the number of failures to tolerate before resubmitting the application is very high, meaning that it is possible that the application never needs to be resubmitted. Finally, we show the advantage of Moldable applications: while the yield obtained is similar for Rigid and Moldable applications, Moldable applications can tolerate more failures and thus restart more rarely than Rigid ones. This means that a Moldable application is more likely to terminate before being resubmitted.

The rest of the paper is organized as follows. Section 2 provides an overview of related work. Section 3 is devoted to formally defining the performance model. Section 4 provides formulas for the yield of Rigid, Moldable and GridShaped applications using the C/R approach, and for the yield of GridShaped applications using the ABFT approach . All these formulas are instantiated through the applicative scenarios in Section 5, to compare the different results. Finally, Section 6 provides final remarks and hints for future work.

2. Related work

We first survey related work on checkpoint-restart in Section 2.1. Then we discuss previous contributions on MOLDABLE applications in Section 2.2. Finally, we provide a few references for ABFT techniques in Section 2.3

2.1. Checkpoint-restart

Checkpoint/restart (C/R) is the most common strategy employed to protect applications from underlying faults and failures on HPC platforms. Generally, C/R periodically outputs snapshots (*i.e.*, checkpoints) of the application global distributed state to some stable storage device. When a failure occurs, the last stored checkpoint is retrieved and used to restart the application.

A widely-used approach for HPC applications is to use a fixed checkpoint period (typically one or a few hours), but it is sub-optimal. Instead, application-specific metrics can (and should) be used to determine the optimal checkpoint period. The well-known Young/Daly formula [31, 9] yields an application optimal checkpoint period, $\sqrt{2\mu C}$ seconds, where C is the time to commit a checkpoint and μ the application Mean Time Between Failures (MTBF) on the platform. We have $\mu = \frac{\mu_{ind}}{N}$, where N is the number of processors enrolled by the application and μ_{ind} is the MTBF of an individual processor [19].

The Young/Daly formula minimizes platform waste, defined as the fraction of job execution time that does not contribute to its progress. The two sources of waste are the time spent taking checkpoints (which motivates longer checkpoint periods) and the time needed to recover and re-execute after each failure (which motivates shorter checkpoint periods). The Young/Daly period achieves the optimal trade-off between these sources to minimize the total waste.

2.2. MOLDABLE *and* GRIDSHAPED *applications*

RIGID and MOLDABLE applications have been studied for long in the context of scientific applications. A detailed survey on various application types (RIGID, MOLDABLE, malleable) was conducted in [12]. Resizing application to improve performance has been investigated by many authors, including [20, 7, 27, 26] among others. A related recent study is the design of a MPI prototype for enabling tolerance in MOLDABLE MapReduce applications [14].

The TORQUE/Maui scheduler has been extended to support evolving, malleable, and MOLDABLE parallel jobs [23]. In addition, the scheduler may have system-wide spare nodes to replace failed nodes. In contrast, our scheme does not assume a change of behavior from the batch schedulers and resource allocators, but utilizes job-wide spare nodes: a node set including potential spare nodes is allocated and dedicated to a job at the time of scheduling, that can be used by the application to restart within the same job after a failure. At the application level, spare nodes have become common in HPC centers since more than a decade [28]. Recent work aims at sharing spare-nodes across the whole platform to achieve a better global resource utilization [22].

An experimental validation of the feasibility of shrinking application on the fly is provided in [3]. In this paper, the authors used an iterative solver application to compared two recovery strategies, shrinking and spare node substitution. They use ULFM, the fault-tolerant extension of MPI that offers the possibility of dynamically resizing the execution after a failure. Finally, in [13, 17], the

authors studied MOLDABLE and GRIDSHAPED applications that continue executing after some failures. They focus on the performance degradation incurred after shrinking or spare node substitution, due to less efficient communications (and in particular collective communications). A major difference with our work is that these studies focus on recovery overhead and do not address overall performance nor yield.

2.3. ABFT

ABFT stands for *Algorithm-Based Fault Tolerant* techniques. It is a widely used approach for linear algebra kernels. Since the pioneering paper of Huang and Abraham [18], ABFT protection has been successfully applied to dense LU [10], LU with partial pivoting [30], Cholesky [16] and QR [11] factorizations, and more recently to sparse kernels like SpMxV (matrix-vector product) and triangular solve [24].

In a nutshell, ABFT consists of adding a few checksum vectors as extra columns of each tile, which will be used to reconstruct data lost after a failure. The checksums are maintained by applying the kernel operations to the extra columns, just as if they were matrix elements. The beauty of ABFT is that these checksums can be used to recover from a failure, without any rollback nor re-execution, by reconstructing lost data and proceeding onward. In addition, the failure-free overhead induced by ABFT is usually small, which makes it a good candidate for the design of fault-tolerant linear algebra kernels. We refer to [5, 10] for recent surveys on the approach.

Altogether, we are not aware of any previous study aiming at determining the optimal number of spares as a function of the downtime and resilience parameters, for a general divisible-load application of either type (RIGID,, MOLDABLE or GRIDSHAPED).

3. Performance model

This section reviews the key parameters of the performance model. Some assumptions are made to simplify the computation of the yield. We discuss possible extensions in Section 6.

3.1. Application/platform framework

We consider perfectly parallel applications that execute on homogeneous parallel platforms. Without loss of generality, we assume that each processor has unit speed: we only need to know that the total amount of work done by p processors within T seconds requires $\frac{p}{q}T$ seconds with q processors.

3.2. Mean Time Between Failures (MTBF)

Each processor is subject to failures which are IID (independent and identically distributed) random variables[3] following an Exponential probability distribution of mean μ_{ind}, the individual processor MTBF. Then the MTBF of a section of the platform comprised of i processors is given by $\mu_i = \frac{\mu_{ind}}{i}$ [19].

[3]In datacenters, failures can actually be correlated in space or in time. But to the best of our knowledge, there is no currently available method to analyze their impact without the IID assumption.

3.3. Checkpoints

Processors checkpoint periodically, using the optimal Young/Daly period [31, 9]: for an application using i processors, this period is $\sqrt{2C_i\mu_i}$, where C_i is the time to checkpoint with i processors. We consider two cases to define C_i. In both cases, the overall application memory footprint is considered constant at Mem_{tot}, so the size of individual checkpoints is inversely linear with the number of participating/surviving processors. In the first case, the I/O bandwidth is the bottleneck (which is often the case in HPC platforms – it takes only a few processors to saturate the I/O bandwidth); then the checkpoint cost is constant and given by $C_i = \frac{Mem_{tot}}{\tau_{io}}$, where τ_{io} is the aggregated I/O bandwidth. In the second case, the processor network card is the bottleneck, and the checkpoint cost is inversely proportional to number of active processors: $C_i = \frac{Mem_{tot}}{\tau_{xnet} \times i}$, where τ_{xnet} is the available network card bandwidth, i.e. the bandwidth available for one and only one processor, and $\frac{Mem_{tot}}{i}$ the checkpoint size. In this second case, the cost of checkpointing (and recovery) increases when the number of processors decreases, since each processor has to read more and more application data from stable storage: this does not impact RIGID applications, but it may represent an important overhead for MOLDABLE and GRIDSHAPED applications.

We denote the recovery time with i processors as R_i. For all simulations we use $R_i = C_i$, assuming that the read and write bandwidths are identical.

3.4. Wait Time

Job schedulers allocate nodes to given applications for a given time. They aim at optimizing multiple criteria, depending on the center policy. These criteria include fairness (balancing the job requests between users or accounts), platform utilization (minimizing the number of resources that are idling), and job makespan (providing the answer as fast as possible). Combined with a high resource utilization (node idleness is usually in the single digit percentage for a typical HPC platform), a job has to wait a *Wait Time* (D) between its submission and the beginning of its execution.

Job schedulers implement the selection based on the list of submitted jobs, each job defining how many processors it needs and for how long. That definition is, in most cases, unchangeable: an application may use less resources than what it requested, but the account will be billed for the requested resources, and it will not be able to re-dimension the allocation during the execution.

Thus, if after some failures, an application has not enough resources left to efficiently complete, it will have to relinquish the allocation, and request a new one. During the wait time D, the application does not execute any computation to progress towards completion: its yield is zero during D seconds.

3.5. Objective.

We consider a long-lasting application that requests a resource allocation with N processors. We aim at deriving the optimal number of failures F that should be tolerated before paying the wait time and requesting a new allocation. We aim at maximizing the *yield* $\mathcal{Y}$ of the application, defined as the fraction of time during the allocation length and wait time where the N resources perform useful work. More precisely, the yield is defined by the following formula:

$$\mathcal{Y} = \frac{\text{total time spent computing for all processors}}{\text{number of processors } \times \text{ flow time since start of application}}.$$

. Of course a spare does not perform useful work when idle, processor do not compute when checkpointing or recovering, re-execution nodes not account for actual work, and no processor is active during wait time. All this explains that the yield will always be smaller than 1. We will derive the value of F that maximizes $\mathcal{Y}$ for the three application types using C/R (and both C/R and ABFT for GridShaped applications).

4. Expected yield

This section is the core of the paper. We compute the expected yield for each application type, Rigid (Section 4.1), Moldable (Section 4.2) and GridShaped (Section 4.3), using the C/R approach, and compare it with ABFT for GridShaped in Section 4.4.

4.1. Rigid *application*

We first consider a Rigid application that can be parallelized at compile-time to use any number of processors but cannot change this number until it reaches termination. There are N processors allocated to the application. We use $N - F$ for execution and keep F as spares. The execution is protected from failures by checkpoints of duration C_{N-F}. Each failure striking the application will incur an in-place restart of duration R_{N-F}, using a spare processor to replace the faulty one. However, when the $(F + 1)^{st}$ failure strikes, the job will have to stop and perform a full restart, waiting for a new allocation of N processors to be granted by the job scheduler.

We define $\mathcal{T}_R$ as the expected duration of an execution period until the $(F+1)^{st}$ failure strikes. The first failure is expected to strike after μ_N seconds, the second failure μ_{N-1} seconds after the first one, and so on. We relinquish the allocation after $F + 1$ failures and wait some time D. As faults can also happen during the checkpoint and the recovery, this means that:

$$\mathcal{T}_R = \sum_{i=N}^{N-F} \mu_i + D. \tag{1}$$

What is the total amount of work $\mathcal{W}_R$ computed during a period $\mathcal{T}_R$? During the sub-period of length μ_i, there are $\frac{\mu_i}{\sqrt{2C_{N-F}\mu_{N-F}}}$ checkpoints, each of length C_{N-F}. The failure hits one of live processors, either a working processor or a spare. In both cases, the number of live processors decreases. if the failure hits a spare, it has no immediate impact on the application, except that the number of available spares decreases. If the failure hits a working processor, which happens with probability $\frac{N-F}{i}$, some work is lost, and a restart is needed. During each sub-period, and weighting the cost by the probability of the failure hitting a working processor during that sub-period, the work lost by each processor by the end of the sub-period is in average $\frac{\sqrt{2C_{N-F}\mu_{N-F}}}{2} \cdot \frac{N-F}{i}$ (see [19] for further details). Each time there is a failure, the next sub-period is thus started by a restart R_{N-F} with probability $\frac{N-F}{i+1}$, except for the first sub-period which always starts by a restart (it corresponds to reading input data at the beginning of the allocation). All in all, during the sub-period of length μ_i with $i \neq N$, each

processor works during

$$\frac{1}{1+\frac{C_{N-F}}{\sqrt{2C_{N-F}\mu_{N-F}}}} \cdot \left(\mu_i - R_{N-F} \cdot \frac{N-F}{i+1} - \frac{\sqrt{2C_{N-F}\mu_{N-F}}}{2} \cdot \frac{N-F}{i} \right)$$

seconds. The first fraction corresponds to the proportion of the time that is used for useful computations and not for checkpointing. This fraction is actually: $\frac{\text{period time}}{\text{period time+checkpoint time}} = \frac{\sqrt{2C_{N-F}\mu_{N-F}}}{\sqrt{2C_{N-F}\mu_{N-F}}+C_{N-F}}$ which is equivalent to the former fraction after simplification.
Finally, each processor works during

$$\frac{1}{1+\frac{C_{N-F}}{\sqrt{2C_{N-F}\mu_{N-F}}}} \cdot \left(\mu_N - R_{N-F} - \frac{\sqrt{2C_{N-F}\mu_{N-F}}}{2} \cdot \frac{N-F}{i} \right)$$

seconds in the first sub-period of length μ_N as it always starts by reading the initial data.

There are $N-F$ processors at work, hence, re-arranging terms, we obtain that

$$\mathcal{W}_R = \frac{N-F}{1+\frac{C_{N-F}}{\sqrt{2C_{N-F}\mu_{N-F}}}} \cdot \sum_{i=N}^{N-F} \left(\mu_i - (R_{N-F} + \frac{\sqrt{2C_{N-F}\mu_{N-F}}}{2}) \cdot \frac{N-F}{i} \right) \tag{2}$$

Indeed, the factor for R_{N-F} is $\frac{N-F}{i+1}$ for all subperiods except the first one, i.e. $N-F \leq i \leq N-1$, which means it is equivalent to $\frac{N-F}{i}$ with $N-F+1 \leq i \leq N$. Moreover, $\frac{N-F}{N-F} = 1$ which is the corresponding factor for the first subperiod, so by summing all the terms we get to $\sum_{i=N}^{N-F} R_{N-F} \cdot \frac{N-F}{i}$.

During the whole duration $\mathcal{T}_R$ of the period, in the absence of failures and protection techniques, the application could have used all the N processors to compute continuously. Thus the effective yield with protection for the application during $\mathcal{T}_R$ is reduced to $\mathcal{Y}_R$:

$$\mathcal{Y}_R = \frac{\mathcal{W}_R}{N \cdot \mathcal{T}_R}$$

4.2. MOLDABLE *application*

We now consider a MOLDABLE application that can use a different number of processors after each restart. The application starts executing with N processors. After the first failure, the application recovers from the last checkpoint and is able to continue with only $N-1$ processors, after paying the restart cost R_{N-1}, albeit with a slowdown factor $\frac{N-1}{N}$ of the parallel work per time unit. After $F+1$ failures, the application stops, just as it was the case for a RIGID application.

We define $\mathcal{T}_M$ as the expected duration of an execution period until the $(F+1)^{st}$ failure strikes. The length of a period is

$$\mathcal{T}_M = \sum_{i=N}^{N-F} \mu_i + D, \tag{3}$$

the same as for RIGID applications.

However, for the total amount of work $\mathcal{W}_M$ during a period, things are slightly different. To compute the total amount of work $\mathcal{W}_M$ during a period $\mathcal{T}_M$, we proceed as before and consider each sub-period. During the sub-period of length μ_i, there are $\frac{\mu_i}{\sqrt{2C_i\mu_i}}$ checkpoints, each of length C_i. There is also a restart R_i at the beginning of each sub-period, and the average time lost is $\frac{\sqrt{2C_i\mu_i}}{2}$. The probability that the failure strikes a working processor is always 1, because all alive processors are working during each sub-period. Overall, there are i processors at work during the sub-period of length μ_i, and each of them actually works during

$$\frac{\mu_i - R_i \ - \frac{\sqrt{2C_i\mu_i}}{2}}{1 + \frac{C_i}{\sqrt{2C_i\mu_i}}}$$

seconds. Altogether, we derive that

$$\mathcal{W}_M = \sum_{i=N}^{N-F} i \times \frac{\mu_i - R_i \ - \frac{\sqrt{2C_i\mu_i}}{2}}{1 + \frac{C_i}{\sqrt{2C_i\mu_i}}} \tag{4}$$

The yield of the MOLDABLE application is then:

$$\mathcal{Y}_M = \frac{\mathcal{W}_M}{N \cdot \mathcal{T}_M}$$

4.3. GRIDSHAPED *application*

Next, we consider a GRIDSHAPED application, defined as a moldable execution which requires a rectangular processor grid. Here we mean a *logical* grid, i.e. a layout of processes whose communication pattern is organized as a 2D grid, not a *physical* processor grid where each processor has four neighbors directly connected to it. Indeed, there is little hope to use physical grids today. Some modern architectures have a multi-dimensional torus as physical interconnexion network, but the job scheduler never guarantees that allocated nodes are adjacent, let alone are organized along an actual torus. This means that the actual time to communicate with a logically adjacent processor is variable, depending upon the length of the path that connects them, and also upon the congestion of the links within that path (these links are likely to be shared by other paths). Other architecture communicate through a hierarchical interconnexion switch, hence a 2D processor grid is not meaningful for such architectures. Altogether, this explains that one targets logical process grids, not physical processor grids. Now why do the application needs a process grid? State-the-art linear algebra kernels such as matrix product, LU, QR and Cholesky factorizations, are most efficient when the data is partitioned across a logical grid of processes, preferably a square, or at least a balanced rectangle of processes [21]. This is because the algorithms are based upon outer-product matrix updates, which are most efficiently implemented on (almost) square grids. Say you start with 64 working processors, arranged as a 8×8 process grid. When one processor fails, the squarest grid would be $63 = 9 \times 7$, and then after a second failure we get $62 = 31 \times 2$ which is way too elongated to be efficient. After the first failure, it is more efficient to use a 8×7 grid and keep 7 spares; then we use spares for the next 7 failures, after which we shrink to a 7×7 grid (and keep 7 spares), and so on.

For the analysis, assume that the application starts with a square $p \times p$ grid of $N = p^2$ processors. After the first failure, execution continues on a $p \times (p-1)$ rectangular grid, keeping $p-1$ processors as spares for the next $p-1$ failures. After p failures, the grid is shrunk again to a $(p-1) \times (p-1)$ square grid, and the execution continues on this reduced-size square grid. After how many failures F should the application stop, in order to maximize the application yield?

The derivation of the expected length of a period and of the total work is more complicated for GridShaped than for Rigid and Moldable. To simplify the presentation, we outline the computation of the yield only for values of F of the form $F = 2pf - f^2$, hence $p^2 = F + (p-f)^2$, meaning that we stop shrinking and request a new allocation when reaching a square grid of size $(p-f) \times (p-f)$ for some value of $f < p$ to be determined. Obviously, we could stop after any number of faults F, and the publicly available software [25] shows how to compute the optimal value of F without any restriction.

We start by computing an auxiliary variable: on a $(p_1 - 1) \times p_2$ grid with $p_1 \geq p_2$, the expected time to move from $p_2 - 1$ spare nodes to no spare nodes will be denoted by $T_G(p_1, p_2)$. It means that the number of computing nodes never changes and is $(p_1-1)p_2$. It always starts with a restart $R_{(p_1-1)p_2}$, because having $p_2 - 1$ spare nodes means that a failure just occurred on one of the $p_1 p_2$ processors that were working just before that failure, and we had to remove a row from the process grid. As previously this time is the sum of all the intervals between each failure, namely:

$$T_G(p_1, p_2) = \sum_{i=p_1p_2-1}^{(p_1-1)p_2} \mu_i.$$

Going from p^2 processors down to $(p-f)^2$ processors thus require a time

$$\mathcal{T}_G = \mu_{p^2} + \sum_{g=0}^{f-1} \big(T_G(p-g, p-g) + T_G(p-g, p-g-1) \big) + D.$$

We simply add the time before the first failure and the wait time to the time needed to move from a grid of size p^2 to $(p-1)^2$, to $(p-2)^2$, ..., to $(p-f)^2$.

Similarly, we define the auxiliary variable $W_G(p_1, p_2)$ as the parallel work when moving from a $(p_1 - 1) \times p_2$ grid with $p_2 - 1$ spare nodes to a $(p_1 - 1) \times p_2$ grid with no spare node, where $p_1 \geq p_2$. There are $(p_1 - 1)p_2$ processors working during all sub-periods. Without restart and re-execution, this work is $(p_1 - 1)p_2 \cdot \sum_{i=p_1p_2-1}^{(p_1-1)p_2} \mu_i$. Any failure which hits one of the working processors calls for a restart $R_{(p_1-1)p_2}$ and incurs some lost work: $\frac{\sqrt{2C_{(p_1-1)p_2}}}{2}$ in average. The first sub-period starts with a restart $R_{(p_1-1)p_2}$,because the application (distributed on a grid of $p_1 \times p_2$ was previously hit by a failure, except if this is the beginning of a new allocation (which case will be dealt with later on). Then, for all other sub-periods, a restart is taken if one of $(p_1 - 1)p_2$ computing processors was hit by a failure. This means that the sub-period with i processors alive (of length μ_i) starts with a restart $R_{(p_1-1)p_2}$ with probability $\frac{(p_1-1)p_2}{i+1}$, for $i \leq p_1p_2 - 2$. Similarly, for all sub-periods with i processors alive, we lose the expected compute time $\frac{\sqrt{2C_{(p_1-1)p_2}\mu_{(p_1-1)p_2}}}{2}$ with probability $\frac{(p_1-1)p_2}{i}$. Finally, the checkpoint period evolves with the number of processors, just as for

MOLDABLE applications. We derive the following formula:

$$W_G(p_1,p_2) = \frac{(p_1-1)p_2}{1+\frac{C_{(p_1-1)p_2}}{\sqrt{2C_{(p_1-1)p_2}\mu_{(p_1-1)p_2}}}} \times \left(\mu_{p_1p_2-1} - R_{(p_1-1)p_2} - \frac{\sqrt{2C_{(p_1-1)p_2}\mu_{(p_1-1)p_2}}}{2} \cdot \frac{(p_1-1)p_2}{p_1p_2-1} + \sum_{i=p_1p_2-2}^{(p_1-1)p_2} \left(\mu_i - R_{(p_1-1)p_2} \cdot \frac{(p_1-1)p_2}{i+1} - \frac{\sqrt{2C_{(p_1-1)p_2}\mu_{(p_1-1)p_2}}}{2} \cdot \frac{(p_1-1)p_2}{i}\right)\right)$$

Going from p^2 processors down to $(p-f)^2$ processors thus corresponds to a total work

$$\mathcal{W}_G = \frac{p^2}{1+\frac{C_{p^2}}{\sqrt{2C_{p^2}\mu_{p^2}}}} \cdot \left(\mu_{p^2} - R_{p^2} - \frac{\sqrt{2C_{p^2}\mu_{p^2}}}{2}\right) + \sum_{g=0}^{f-1} \left(W_G(p-g,p-g) + W_G(p-g,p-g-1)\right)$$

We use the previously computed function just as we did with the time and we add the work done during the first sub-period on the initial grid of size p^2 (this is the special case for the beginning of an allocation that was mentioned above). Its computation is similar to that of other subperiods.

The yield of the GRIDSHAPED application is then:

$$\mathcal{Y}_G = \frac{\mathcal{W}_G}{N \cdot \mathcal{T}_G}$$

where $N = p^2$.

4.4. ABFT for GRIDSHAPED

Finally, in this section, we investigate the impact of using *Algorithm-Based Fault Tolerant* techniques, or ABFT, instead of Checkpoint:Restart (C/R). Just as before, we build a performance model that uses first-order approximations. In particular, we do not consider overlapping failures, thereby allowing for a failure-free reconstruction of lost data after a failure. This first-order approximation is accurate up to a few percent, whenever the failure rate is not too high, or more precisely, when the MTBF remains an order of magnitude higher than resilience parameters [19]. Note that this is the case for state-of-the-art platforms, but may prove otherwise whenever millions of nodes are assembled in the forthcoming years.

Consider a matrix factorization on a $p \times p$ grid. The matrix is of size $n \times n$ and is partitioned into tiles of size $b \times b$. These tiles are distributed in a 2D block-cyclic fashion across processors. Letting $n = pbr$, each processor initially holds r^2 tiles. Every set of p consecutive tiles in the matrix is checksummed into a new tile, which is duplicated for resilience. These two new tiles are added to the right border of the matrix and will be distributed in a 2D block-cyclic fashion across processors, just as the original matrix tiles. In other words, we add $2pr^2$ new tiles, extending each tile row of the matrix (there are pr such tile rows) with $2r$ new tiles. Fig. 2 illustrates this scheme: the white area represents the original user matrix of size $n \times n$, split in tiles of size $b \times b$, and distributed

over a $p \times p$ process grid. The number in each tile represents the rank that hosts a given tile of the matrix. There are two groups of tile-columns of checksums: the light grey ones checksum the right end of the matrix, and the dark grey ones checksum the left part of the matrix. For a bigger matrix, more groups would be added, each group accumulate the sum of p consecutive tile-columns of the matrix. In each group, there are two tile-columns: the checksum and its replica. These new tiles will be treated as regular matrix tiles by the ABFT algorithm, which corresponds to a ratio $\frac{2pr^2}{p^2r^2} = \frac{2}{p}$ of extra work, and to a failure-free slowdown factor $1 + \frac{2}{p}$ [5].

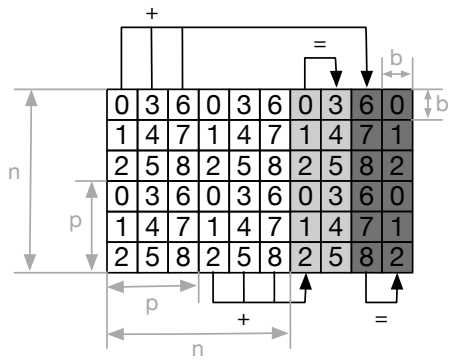


Figure 2: Example of data redundancy and checksuming in an ABFT factorization. Each white square represents a matrix tile; numbers in the square represent the rank on which this tile is hosted; grey tiles represent the checksums and their replica (symbolized by the = arrow). In this case, $p = 3$, $n = 6b$.

Now, if a processor crashes, we finish the current step of the factorization on all surviving processors, and then reconstruct the tiles of the crashed processor as follows:

- For each tile lost, there are $p - 1$ other tiles (the ones involved in the same checksum as the lost tile) and at least one checksum tile (maybe two if the crashed processor did not hold any of the two checksum tiles for that lost tile). This is what is needed to enable the reconstruction. We solve a linear system of size b and reconstruct the missing tile for a time proportional to $b^3 + pb^2$.
- We do this for the r^2 tiles of the crashed processor, for a total time of $O(r^2(b^3 + pb^2)\tau_a)$, where τ_a is the time to perform a floating-point operation.

Doing so, we reconstruct the same tile as if we had completed the factorization step without failure.

Then, there are two cases: the ABFT algorithm relies on a process grid, so the application behaves similarly to a GridShaped application. If spare nodes are available, one of them is selected and inserted within the process grid at the place of the crashed processor, at a cost of communicating $O(r^2b^2)$ matrix coefficients (the amount of data held by the faulty processor). If, on the other hand, there are no spare nodes, we have to start the redistribution

of the matrix onto a $p \times (p-1)$ grid. The distribution is operated in parallel across the p grid rows. Within a processor row, most of the tiles will have to change owner to keep enforcing a 2D block-cyclic distribution, which implies $O(\frac{n^2}{p}) = O(r^2pb^2)$ communications as we redistribute everything on every row. Since all rows operate in parallel, the time for the redistribution is $O(r^2pb^2\tau_c)$, where τ_c is the time to communicate a floating point number. Altogether, the total cost to recover from a failure is $O(r^2b^2(b\tau_a + p(\tau_a + \tau_c)))$.

In the following, we compute the expected yield of a linear algebra kernel protected by ABFT. Again, we consider numbers of failures of the form $F = 2pf - f^2$ so that $p^2 = F + (p-f)^2$. Again, we could stop after any number of faults F, and the publicly available software [25] shows how to do so.

We first compute the expected time between two full restarts:

$$\mathcal{T}_{ABFT} = \sum_{i=p^2}^{(p-f)^2} \mu_i + D.$$

As previously, we tolerate failures up to reaching a processor grid size of $(p-f) \times (p-f)$, each inter-arrival time being the MTBF of the platform with the corresponding number of alive processors.

Now, we define the auxiliary variable $W_{ABFT}(p_1, p_2)$ as the parallel work when moving from a $(p_1-1) \times p_2$ grid with $p_2 - 1$ spare nodes to a $(p_1-1) \times p_2$ grid with no spare node. There are $(p_1-1)p_2$ processors working during all sub-periods, just as it was the case for GridShaped applications. A failure-free execution would imply a parallel work of $\frac{1}{1+\frac{2}{p}} \sum_{i=p_1p_2-1}^{(p_1-1)p_2} \mu_i$ which is the total time of computation divided by overhead added with the checksum tiles. However, for each failure we need to reconstruct the lost tiles and either pay a redistribution cost (during the first sub-period where we just reduced the size of the grid because we had no spare) or a communication cost to send data to one of the $p_2 - 1$ spare nodes (all other sub-periods where we select a spare). This happens if and only if the failure stroke a working processor just as in the GridShaped case, i.e. with probability $\frac{(p_1-1)p_2}{i+1}$. In the end, since there are $(p_1-1)p_2$ processors at work, we get the following formula:

$$W_{ABFT}(p_1,p_2) = \frac{(p_1-1)p_2}{1+\frac{2}{p}} \left(\mu_{p_1p_2-1} - RD_{p_1} + \sum_{i=p_1p_2-2}^{(p_1-1)p_2} (\mu_i - RP \cdot \frac{(p_1-1)p_2}{i+1}) \right),$$

where RP is the cost to replace a faulty processor by a spare, namely

$$RP = r^2(b^3 + pb^2)\tau_a + r^2b^2\tau_c,$$

and RD_i is the cost to reistribute data, namely

$$RD_i = r^2(b^3 + pb^2)\tau_a + \frac{n^2}{i}\tau_c.$$

To compute these values, we proceed as follows:

- The reconstruction of the lost tiles always takes $r^2(b^3+pb^2)$ floating-point operations, and the enrollment of the spare requires that it receives r^2b^2 floating-point values, which directly leads to the value of RP, which is independent of the number of processors;

- However, when we redistribute the tiles to shrink the grid, the time needed increases as the number of processors decreases because each of them has to gather more data: it requires $O(\frac{n^2}{i})$ communications when redistributing from a $i \times j$ grid of processors to a $(i-1) \times j$ grid, or similarly from a $j \times i$ grid to a $j \times (i-1)$ grid, where $j = i$ or $j = i-1$.

Overall, going from a $p \times p$ grid to a $(p-f) \times (p-f)$ grid corresponds to a total work of $\mathcal{W}_{ABFT}$. At the beginning of the allocation, we need to read the input data, then we wait for the first failure to happen (giving a work of $\mu_{p^2} - R_{p^2}$ during the first sub-period that we divide by the overhead added by the checksum tiles) and then we use our auxiliary variables to decrease the size of the grid step by step. This leads to the following:

$$\mathcal{W}_{ABFT} = \frac{p^2(\mu_{p^2} - R_{p^2})}{1 + \frac{2}{p}} + \sum_{i=0}^{f-1} \left(W_{ABFT}(p-i, p-i) + W_{ABFT}(p-i, p-1-i) \right).$$

The yield of the application protected with ABFT is then:

$$\mathcal{Y}_{ABFT} = \frac{\mathcal{W}_{ABFT}}{N \cdot \mathcal{T}_{ABFT}}$$

where $N = p^2$.

5. Applicative scenarios

We consider several applicative scenarios in this section. We start with a platform inspired from existing ones in Section 5.1, then we study the impact of several key parameters in Section 5.2. Finally, we compare ABFT and C/R for a GridShaped application in Section 5.3.

5.1. Main scenario

As a main applicative scenario using C/R, we consider a platform with 22,250 nodes (150^2), with a node MTBF of 20 years, and an application that would take 2 minutes to checkpoint (at 22,250 nodes). In other words, we let $N = 22,500$, $\mu_{ind} = 20y$ and $C_i = C = 120s$. These values are inspired from existing platforms: the Titan supercomputer at OLCF [15], for example, holds 18,688 nodes, and experiences a few node failures per day, implying a node MTBF between 18 and 25 years. The filesystem has a bandwidth of 1.4TB/s, and nodes altogether aggregate 100TB of memory, thus a checkpoint that would save 30% of that system should take in the order of 2 minutes to complete. In other words, $C_i = C = 120$ seconds for all $i \leq 18,688$.

Figure 3 shows the yield that can be expected if doing a full restart after an optimal number of failures, as a function of the wait time, for the three kind of applications considered (Rigid, Moldable and GridShaped). We also plot the expected yield when the application experiences a full restart after each failure (NoSpare). First, one sees that the three approaches that avoid paying the cost of a wait time after every failure experience a comparable yield, while the performance of the NoSpare approach quickly degrades to a small efficiency (30% when the wait time is around 14h).

The zoom box to differentiate the Rigid, Moldable and GridShaped yield shows that the Moldable approach has a slightly higher yield than the

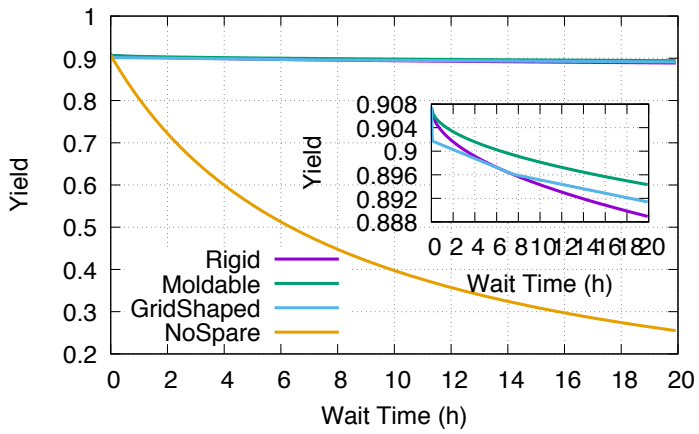


Figure 3: Optimal yield as function of the wait time, for the different types of applications.

other ones, but only for a minimal fraction of the yield. This is expected, as the MOLDABLE approach takes advantage of all living processors, while the GRIDSHAPED and RIGID approaches sacrifice the computing power of the spare nodes waiting for the next failure. However, the size of the gain is small to the point of being negligible. The GRIDSHAPED approach experiences a yield whose behavior changes in steps: it starts with a constant slope, under the RIGID yield, until the wait time reaches 8h at which point both RIGID and GRIDSHAPED yields are the same. The slope of GRIDSHAPED then becomes smaller, exhibiting a better yield than RIGID and slowly reaching the yield of MOLDABLE. If we extend the wait time, or change the configuration to experience more phase changes (as is done in Section 5.2 below), the yield of GRIDSHAPED would reach the same value as the yield of MOLDABLE, at which point the slope of GRIDSHAPED would change again and become higher. This phenomenon is explained by the next figures.

Figure 4 shows the number of failures after which the application should do a full restart, to obtain an optimal yield, as a function of the wait time, for the three kind of applications considered. We observe that this optimal is quickly reached: even with long wait times (e.g. 10h), 170 to 250 failures (depending on the method) should be tolerated within the allocation before relinquishing it. This is small compared to the number of nodes: less than 1% of the resource should be dedicated as spares for the RIGID approach, and after losing 1% of the resource, the MOLDABLE approach should request a new allocation.

This is remarkable, taking into account the poor yield obtained by the approach that does not tolerate failures within the allocation. Even with a small wait time (assuming the platform would be capable of re-scheduling applications that experience failures in less than 2h), Figure 3 shows that the yield of the NOSPARE approach would decrease to 70%. This represents a waste of 30%, which is much higher than the recommended waste of 10% for resilience in the current HPC platforms recommendations [8, 6]. Comparatively, keeping only 1% of additional resources (within the allocation) would allow to maintain a yield at 90%, for every approach considered.

The GRIDSHAPED approach experiences steps that correspond to using all

the spares created when redeploying the application over a smaller grid before relinquishing the allocation. As illustrated in Figure 3, the yield evolves in steps, changing the slope of a linear approximation radically when redeploying over a smaller grid. This has for consequence that the maximal yield is always at a slope change point, thus at the frontier of a new grid size. It is still remarkable that even with very small wait times, it is more beneficial to use spares (and thus to lose a full row of processors) than to redeploy immediately.

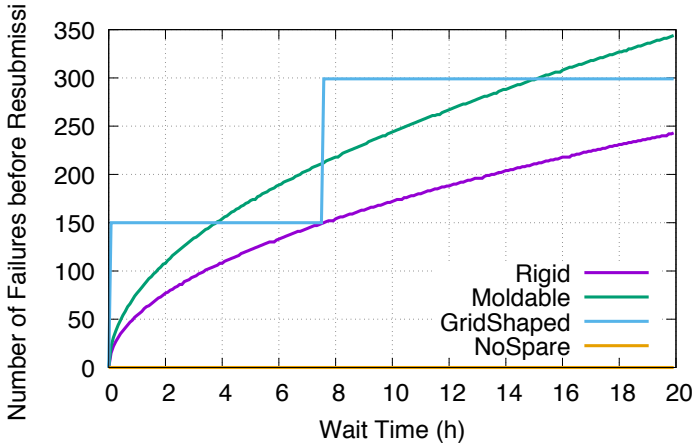


Figure 4: Optimal number of failures tolerated between two full restarts, as function of the wait time, for the different types of applications.

Figure 5 shows the maximal length of an allocation: after such duration, the job will have to fully restart in order to maintain the optimal yield. This figure illustrates the real difference between the RIGID and MOLDABLE approaches: although both approaches are capable of extracting the same yield, the MOLDABLE approach can do so with significantly longer periods between full restarts. This is important when considering real life applications, because this means that the applications using a MOLDABLE approach have a higher chance to complete before the first full restart, and overall will always complete in a lower number of allocations than the RIGID approach.

Finally, Figure 6 shows an upper limit of the duration of the wait time in order to guarantee a given yield for the three applications. In particular, we see that to reach a yield of 90%, an application which would restart its job at each fault would need that restart to be done in less than 6 minutes whereas the RIGID and GRIDSHAPED approaches need a full restart in less than 3 hours approximately. This bound goes up to 7 hours for the MOLDABLE approach. In comparison, with a wait time of 1 hour, the yield obtained using NOSPARE is only 80%. This shows that, using these parameters, it seems impossible to guarantee the recommended waste of 10% without tolerating (a small) number of failures before rescheduling the job.

5.2. Varying key parameters

We performed a full-factorial 4 level design simulation to assess the impact of key parameters. We tried all combinations of MTBF (5 years, 10 years, 20 years, 50 years), checkpointing cost (2 minutes, 10 minutes, 30 minutes, 60 minutes)

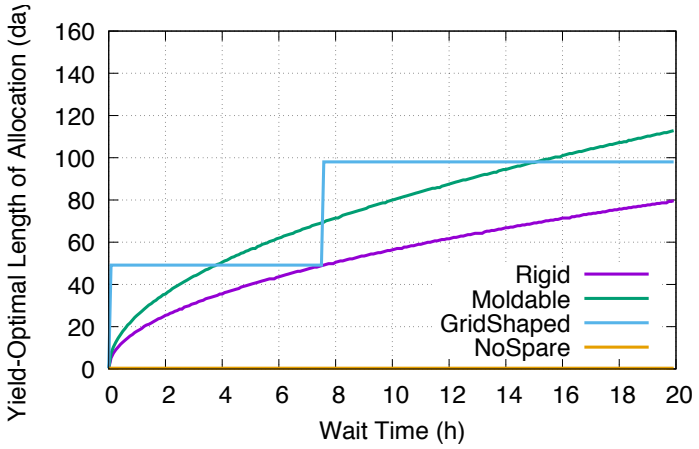


Figure 5: Optimal length of allocations, for the different types of applications.

and application size ($50 \times 50 = 2500$, $150 \times 150 = 22500$, $250 \times 250 = 62500$, $350 \times 350 = 122500$). Not all results are presented for conciseness, but they all give very similar results compared to the main scenario of Section 5.1.

Figure 7 shows the yield and the corresponding allocation length for different values of the MTBF, when using the largest application size $N = 350 \times 350$. The top subfigure is for $\mu_{ind} = 5$ years while the bottom subfigure is for $\mu_{ind} = 50$ years. The checkpoint cost is $C_i = C = 10$ minutes. As expected, the yield increases when the MTBF increases. However, the variation of the allocation length is a bit different. At first, it decreases with the MTBF (for example, with a wait time of 10 hours, it decreases from around 150 days to around 100 days when μ_{ind} decreases from 50 years to 20 years). This is because the optimal number of faults allowed is not much higher when $\mu_{ind} = 20$ years, thus it decreases the overall allocation length. However, when we reach limit behaviours with short node MTBF, the number of failures to tolerate explodes and increases the allocation length. We can also see that the allocation length for GridShaped applications tends to follow that of a Moldable application when μ_{ind} decreases.

Figure 8 shows the optimal number of faults to tolerate for the four different application sizes (with $\mu_{ind} = 20$ years and $C_i = C = 10$ minutes). We can see from this experiment that the number of tolerated failures stays within a small percentage of the total number of processors. In particular, the optimal number of failures allowed for every type of application stays below or equals 2% of the total application size in all the four cases.

Figure 9 aims at showing the impact of the checkpointing cost on the allocation length. The trend is that it does not depend on the checkpointing cost. This can be explained by the fact that the allocation length does not take into account the checkpoint/restart strategy into its computation, only the MTBF and the number of failures allowed. Overall, the impact of the checkpointing cost stays minimal compared to the impact of the wait time or the MTBF.

Finally, Figure 10 describes the yield obtained when using different models for the checkpointing cost: either the checkpoint is constant (independent of the number of processors: left figure) or it is inversely proportional to the number

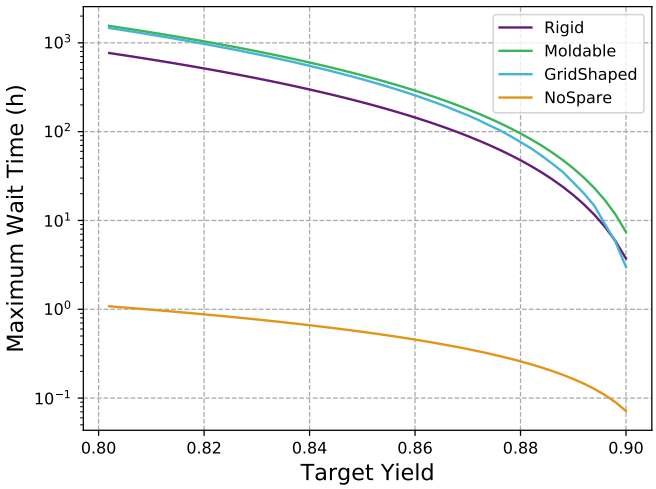


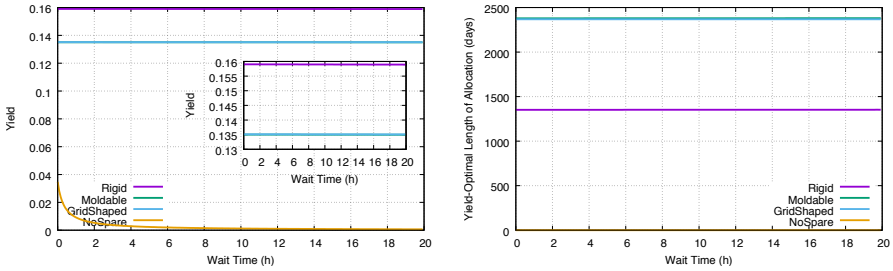
Figure 6: Maximum wait time allowed to reach a target yield.

of processors (right figure). As these plots show, the difference between the two models does not have a noticeable impact on the yield of the applications. This can be explained as follows: as Figure 8 showed, only a small number of faults is allowed before resubmission, in comparison to the application size. Changing the number of active processors by a few percentage does not really make a difference for the checkpoint cost, which remains almost the same in both models.

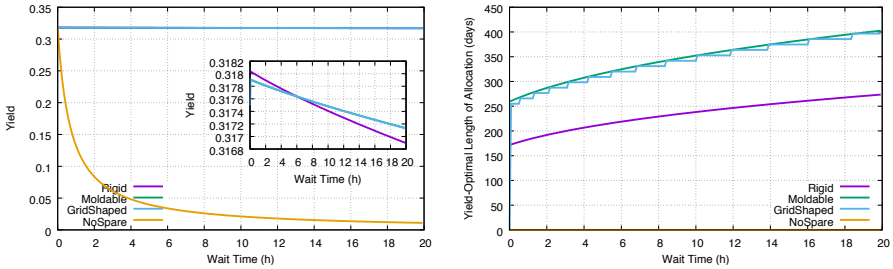
5.3. Comparison between C/R and ABFT

In this subsection, we present the results of ABFT and C/R strategies, for a GridShaped application. In order to compare both strategies, we introduce ABFT parameters, and use data from the Titan platform [2]:

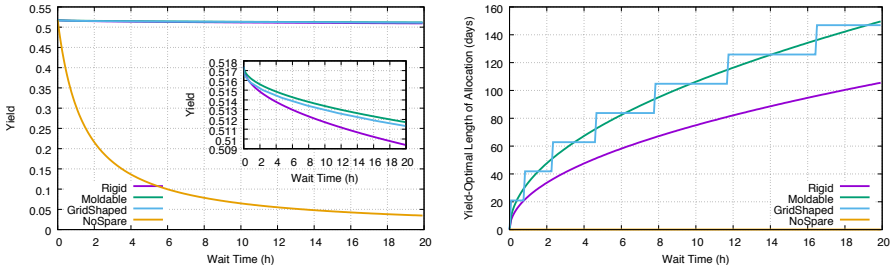
- We use tiles of size 180×180, i.e. we let $b = 180$. We set $r = 325$; so that a node holds $325^2 = 105625$ tiles.
- These values give a total of almost 25.5 GB used by each node, which corresponds to 80% of the memory of a node in Titan.
- Overall, the total memory of the application is $8p^2r^2b^2$ bytes, so we set the checkpointing cost to be $\frac{8p^2r^2b^2}{1.4\times1024^4}$, using 1.4 TB/s for the I/O bandwidth of the Titan platform. With r and b set as mentioned, we get $C_i = C \approx \frac{25.5N}{1.4\times1024} \approx \frac{N}{56.3}$.
- Titan has 18,688 cores for a peak performance of 17.59 PFlop/s. We derive a performance per core of 987 GFlop/s, i.e. $\tau_a = \frac{1}{987\times1024^3}$.
- Using the same reasoning, we derive that $\tau_c = \frac{1}{87.2\times1024^3}$.



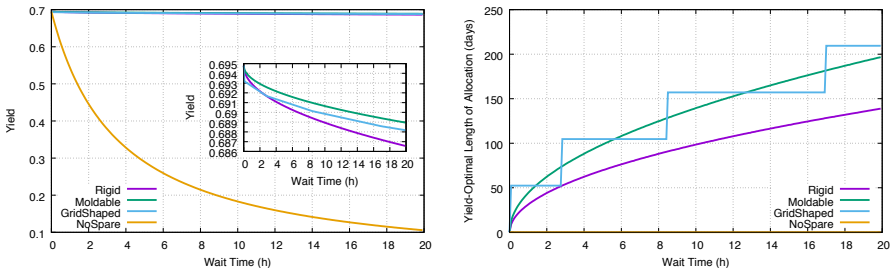
(a) $\mu_{ind} = 5$ years



(b) $\mu_{ind} = 10$ years

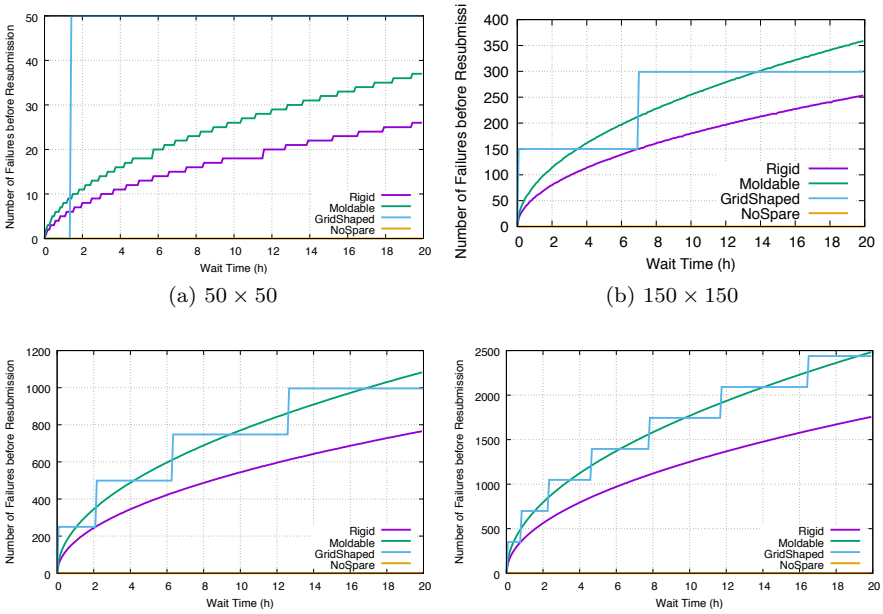


(c) $\mu_{ind} = 20$ years



(d) $\mu_{ind} = 50$ years

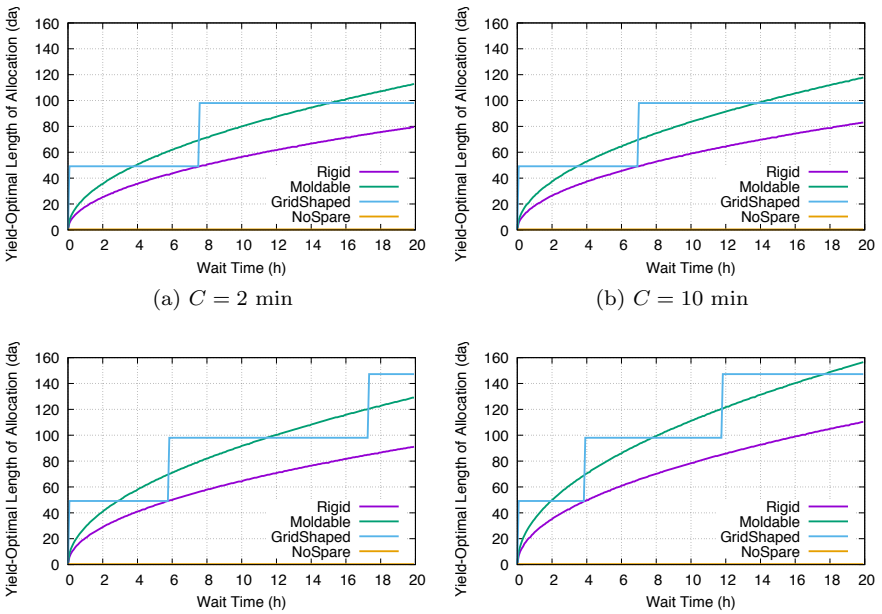
Figure 7: Yield and optimal allocation length of as a function of the wait time with $N = 350 \times 350$, and $C = 10$ minutes.



(a) 50×50 (b) 150×150

(c) 250×250 (d) 350×350

Figure 8: Optimal number of faults before rescheduling the application for different application sizes.



(a) $C = 2$ min (b) $C = 10$ min

(c) $C = 30$ min (d) $C = 60$ min

Figure 9: Optimal number of faults before rescheduling the application for different checkpointing costs.

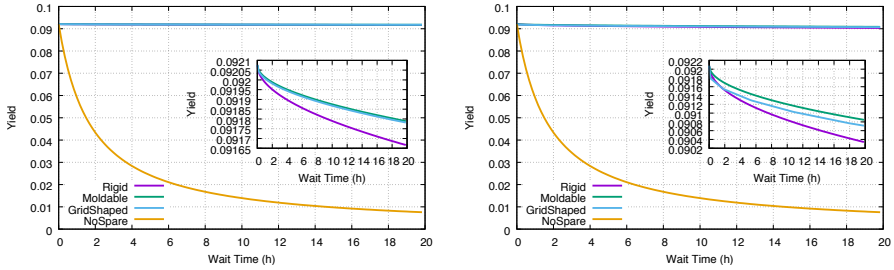


Figure 10: Constant checkpoint cost ($C_i = 60$ min) on the left, and increasing checkpoint cost ($C_i = \frac{N}{i} \times 60$ min) on the right, with $\mu_{ind} = 5$ years and $N = 350 \times 350$.

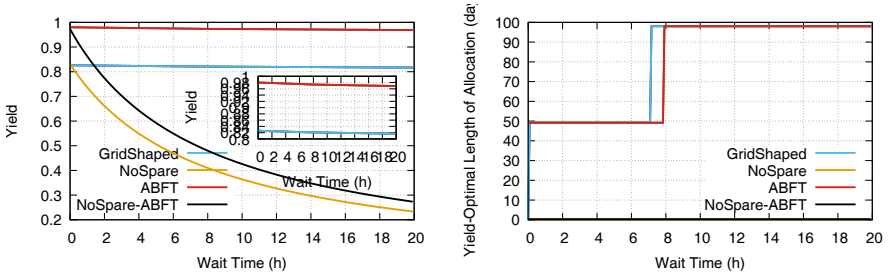


Figure 11: Comparison of ABFT and C/R strategies for a GridShaped application, $N = 150 \times 150$ and $\mu_{ind} = 20$ years.

Figure 11 presents the yield obtained by both strategies with either no spare processors (NoSpare and NoSpare-ABFT) or with the optimal number of spare processors (GridShaped for the C/R strategy and ABFT for the ABFT strategy). In Figure 11 , we use $N = 150 \times 150$ and $\mu_{ind} = 20$ years. Unsurprisingly, the ABFT strategy grants a better yield than the C/R strategy with a yield very close to 1, compared to ≈ 0.8 for C/R. This is largely due to the fact that the overhead added by the ABFT is $\frac{2}{p}$ and so is negligible compared to the checkpoint overhead. Moreover, the reconstruction of the tiles is done in parallel so it does not induce any significant overhead. This can also be seen when we do not use any spare: C/R and ABFT follow the same trend but ABFT is always more efficient than C/R, which exactly shows that the checkpoint overhead is larger than the ABFT overhead, since it is the only source (along with the wait time) of wasted time if $F = 0$. For a wait time of 10 hours the C/R strategy gives a yield of 0.820 while ABFT grants a better yield of 0.973 (0.364 and 0.426 respectively with no spare processors). We can see on the right figure that the allocation lengths are similar for both strategies. However, for some values, ABFT will have a shorter allocation length, mostly due to the fact that its overhead is small and does not depend on the number of alive processors; hence loosing a few nodes implies a greater slowdown than for the C/R strategy where the checkpointing period is adapted regularly.

The conclusion of this comparative study is that, for a GridShaped application, ABFT uses a very small percentage of spare resources and grants a better yield than classical C/R.

6. Conclusion

In this paper, we have compared the performance of RIGID, MOLDABLE and GRIDSHAPED applications when executed on large-scale failure-prone platforms. We have mainly focused on the C/R approach, because it is the most widely used approach for resilience. For each application type, we have computed the optimal number of faults that should be tolerated before requesting a new allocation, as a function of the wait time. Through realistic applicative scenarios inspired by state-of-the-art platforms, we have shown that the three application types experience an optimal yield when requesting a new allocation after experiencing a number of failures that represents a small percentage of the initial number of resources (hence a small percentage of spares for RIGID applications), and this even for large values of the wait time. On the contrary, the NOSPARE strategy, where a new allocation is requested after each failure, sees its yield dramatically decrease when the wait time increases. We also observed that MOLDABLE applications enjoy much longer execution periods in between two re-allocations, thereby decreasing the total execution time as compared to RIGID applications (and GRIDSHAPED applications lying in between).

GRIDSHAPED applications may also be protected using ABFT, and we have compared the efficiency of C/R and ABFT for a typical dense matrix factorization problem. As expected, using ABFT leads to even better yields than C/R for a wide variety of scenarios, in particular for larger problem sizes for which ABFT scales remarkably well.

Future work will be devoted to exploring more applicative scenarios, and running actual experiments using ULFM [4]. We also intend to extend the model in several directions. On the application side, we aim at dealing with non-perfectly parallel applications but instead with applications whose speedup profile obeys Amdahl's law [1]. On the platform side, we aim at adapting the model to heterogeneous platforms and at doing more experiments with different values for the recovery and checkpoint costs as bandwidths are different when reading or writing data. We will also introduce a more refined speedup profile for GRIDSHAPED applications, with an execution speed that depends on the grid shape (a square being usually faster than an elongated rectangle). On the resilience side, we will explore the case with different costs for checkpoint and recovery. More importantly, we will address the combination of ABFT and C/R (instead of dealing with either method individually). Such a combination would allow to tolerate for several failures striking within the same computational step: the idea would be to use ABFT to recover from a single failure and to rollback to the last checkpoint only in the case of multiple failures. Such a combination would enable us to go beyond first-order approximations and single-failure scenarios. Finally, we would like to investigate the case for correlated failures. Even if a theoretical analysis seems out of reach, we could generate failures with models accounting for correlation but optimize for the current model, and check how the correlations affect the results.

Acknowledgement

This research is partially supported by the NSF (award #1564133). We would like to thank the reviewers for their comments and suggestions.

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] APEX. APEX Workflows. Research report SAND2016-2371 and LA-UR-15-29113, LANL, NERSC, SNL, 2016.

[3] R. A. Ashraf, S. Hukerikar, and C. Engelmann. Shrink or substitute: Handling process failures in HPC systems using in-situ recovery. *CoRR*, abs/1801.04523, 2018.

[4] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Post-failure recovery of MPI communication capability: Design and rationale. *International Journal of High Performance Computing Applications*, 27(3):244–254, 2013.

[5] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou. Algorithm-based fault tolerance applied to high performance computing. *J. Parallel Distrib. Comput.*, 69(4):410–416, 2009.

[6] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1), 2014.

[7] W. Cirne and F. Berman. Using moldability to improve the performance of supercomputer jobs. *J. Parallel Distrib. Comput.*, 62(10):1571–1601, 2002.

[8] CORAL: Collaboration of Oak Ridge, Argonne and Livermore National Laboratorie. Draft CORAL-2 build statement of work. Technical Report LLNL-TM-7390608, Lawrence Livermore National Laboratory, March, 30 2018.

[9] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Comp. Syst.*, 22(3):303–312, 2006.

[10] P. Du, A. Bouteiller, et al. Algorithm-based fault tolerance for dense matrix factorizations. In *PPoPP*, pages 225–234. ACM, 2012.

[11] P. Du, P. Luszczek, S. Tomov, and J. Dongarra. Soft error resilient QR factorization for hybrid system with GPGPU. *Journal of Computational Science*, 4(6):457 – 464, 2013. Scalable Algorithms for Large-Scale Systems Workshop (ScalA2011), Supercomputing 2011.

[12] P. Dutot, G. Mounié, and D. Trystram. Scheduling parallel tasks approximation algorithms. In J. Y. Leung, editor, *Handbook of Scheduling - Algorithms, Models, and Performance Analysis*. CRC Press, 2004.

[13] A. Fang, H. Fujita, and A. A. Chien. Towards understanding post-recovery efficiency for shrinking and non-shrinking recovery. In *Euro-Par 2015: Parallel Processing Workshops*, pages 656–668. Springer, 2015.

[14] Y. Guo, W. Bland, P. Balaji, and X. Zhou. Fault tolerant MapReduce-MPI for HPC clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, pages 34:1–34:12, 2015.

[15] S. Gupta, T. Patel, C. Engelmann, and D. Tiwari. Failures in large scale systems: Long-term measurement, analysis, and implications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 44:1–44:12, New York, NY, USA, 2017.

[16] D. Hakkarinen, P. Wu, and Z. Chen. Fail-stop failure algorithm-based fault tolerance for cholesky decomposition. *Parallel and Distributed Systems, IEEE Transactions on*, 26(5):1323–1335, May 2015.

[17] A. Hori, K. Yoshinaga, T. Herault, A. Bouteiller, G. Bosilca, and Y. Ishikawa. Sliding substitution of failed nodes. In *Proceedings of the 22Nd European MPI Users' Group Meeting*, EuroMPI '15, pages 14:1–14:10, New York, NY, USA, 2015. ACM.

[18] K.-H. Huang and J. A. Abraham. Algorithm-Based Fault Tolerance for Matrix Operations. *Computers, IEEE Transactions on*, C–33(6):518–528, 1984.

[19] T. Hérault and Y. Robert, editors. *Fault-Tolerance Techniques for High-Performance Computing*. Springer Verlag, 2015.

[20] J. E. Moreira and V. K. Naik. Dynamic resource management on distributed systems using reconfigurable applications. *IBM Journal of Research and Development*, 41(3):303–330, 1997.

[21] A. Petitet, H. Casanova, J. Dongarra, Y. Robert, and R. C. Whaley. Parallel and distributed scientific computing: A numerical linear algebra problem solving environment designer's perspective. In J. Blazewicz, K. Ecker, B. Plateau, and D. Trystram, editors, *Handbook on Parallel and Distributed Processing*. Springer Verlag, 1999. Available as LAPACK Working Note 139.

[22] S. Prabhakaran, M. Neumann, and F. Wolf. Efficient fault tolerance through dynamic node replacement. In *18th Int. Symp. on Cluster, Cloud and Grid Computing CCGRID*, pages 163–172. IEEE Computer Society, 2018.

[23] S. Prabhakaranw. *Dynamic Resource Management and Job Scheduling for High Performance Computing*. PhD thesis, Technische Universität Darmstadt, 2016.

[24] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In *ICS*. ACM, 2012.

[25] Simulation Software. Computing the yield. `https://zenodo.org/record/2159761#.XA51mhCnfCJ`, 2018.

[26] R. Sudarsan and C. J. Ribbens. Design and performance of a scheduling framework for resizable parallel applications. *Parallel Computing*, 36(1):48–64, 2010.

[27] R. Sudarsan, C. J. Ribbens, and D. Farkas. Dynamic resizing of parallel scientific simulations: A case study using LAMMPS. In *Int . Conf . Computational Science ICCS*, pages 175–184. Procedia, 2009.

[28] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Proactive process-level live migration in hpc environments. In *SC '08: Proc.ACM/IEEE Conference on Supercomputing*. ACM Press, 2008.

[29] K. Yamamoto, A. Uno, H. Murai, T. Tsukamoto, F. Shoji, S. Matsui, R. Sekizawa, F. Sueyasu, H. Uchiyama, M. Okamoto, N. Ohgushi, K. Takashina, D. Wakabayashi, Y. Taguchi, and M. Yokokawa. The K computer Operations: Experiences and Statistics. *Procedia Computer Science (ICCS)*, 29:576–585, 2014.

[30] E. Yao, J. Zhang, M. Chen, G. Tan, and N. Sun. Detection of soft errors in LU decomposition with partial pivoting using algorithm-based fault tolerance. *International Journal of High Performance Computing Applications*, 29(4):422–436, 2015.

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