Resilient N-Body Tree Computations with Algorithm-Based Focused Recovery: Model and Performance Analysis

Aurélien Cavelan[1,2], Aiman Fang[3], Andrew A. Chien[3,4], and Yves Robert[2,5]
aurelien.cavelan@unibas.ch, yves.robert@inria.fr, {aimanf, achien}@cs.uchicago.edu

[1] University of Basel, Switzerland
[2] Laboratoire LIP, ENS Lyon & Inria, France
[3] University of Chicago, USA
[4] Argonne National Laboratory, USA
[5] University of Tennessee Knoxville, USA

Abstract This paper presents a model and performance study for Algorithm-Based Focused Recovery (ABFR) applied to N-body computations, subject to latent errors. We make a detailed comparison with the classical Checkpoint/Restart (CR) approach. While the model applies to general frameworks, the performance study is limited to perfect binary trees, due to the inherent difficulty of the analysis. With ABFR, the crucial parameter is the detection interval, which bounds the error latency. We show that the detection interval has a dramatic impact on the overhead, and that optimally choosing its value leads to significant gains over the CR approach.

1 Introduction

Future large-scale systems are projected to have higher error rates, with MTBFs (Mean Time Between Failures) as low as 20 minutes [1]. We focus on latent errors, that are not detected immediately after their occurrence. Such errors escape simple system level detection and can only be exposed by sophisticated application checks [2,3]. We use the term "detection latency" to denote the time from error occurrence to detection. Such latency may be thousands (10^3) to billions (10^9) cycles, corrupting a range of computational data. Without support to detect and recover from latent errors, applications will suffer silent data corruption, producing invalid scientific results.

In previous work [4], we proposed a new approach, Application-Based Focused Recovery (ABFR), that exploits application data flow and intermediate states to focus recovery on an accurate estimate of potentially corrupted data. Our study on stencil computations demonstrated ABFR reduces recovery cost by up to 400x. This paper investigates the use of ABFR for N-body computations in the presence of latent errors. N-body computations are much more challenging, as information is exchanged along time-varying patterns that progress up and down the computation tree.

The first contribution of this paper is to propose a detailed model to enable the comparison of ABFR with the classical Checkpoint/Restart (CR) approach.

The model is valid for arbitrary N-body trees, which can be either binary trees, or quad-trees-, or oct-trees, and which are locally imbalanced to account for specific simulation requirements. The second and major contribution is to provide a comprehensive performance study for perfect binary trees. While the scenario of perfect binary trees is not the most general, it encompasses the intrinsic complexity of the whole model while being amenable to an exact analytical evaluation. In particular, setting the value of the detection interval, which bounds the error latency, is crucial to minimize the overhead incurred by ABFR to detect, and recover from, a latent error. We show how to compute the optimal value of this key parameter, and that the optimal value leads to significant savings over the CR approach. This result is an important step towards a full understanding of the potential impact of ABFR to N-body computations.

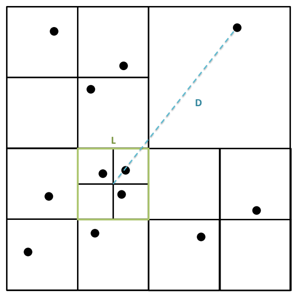
The rest of the paper is organized as follows. We start with background material in Section 2: in Section 2.1, we introduce Global View Resilience (GVR), the execution framework for resilient computing which is used as the support to deploy ABFR, and in Section 2.2, we briefly review N-body computations. Next, we outline the general principles of the ABFR approach in Section 3, and describe how to apply ABFR for N-body tree simulations. Then we provide a detailed formulation of the performance model in Section 4. Sections 5 and 6 are devoted to the performance study, and show how to compute the expected cost (Section 5) and expected overhead (Section 6) of the CR and ABFR approaches. Section 7 shows how to compute the optimal detection interval for ABFR. We report simulation results corresponding to a broad range of scenarios in Section 8. Section 9 presents related work. Finally, we give concluding remarks and hints for future directions in Section 10.

2 Background

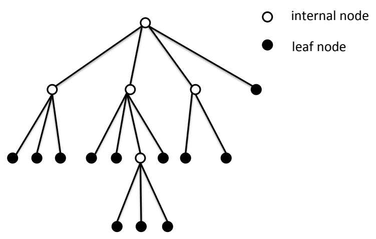
2.1 Global View Resilience (GVR)

We use the GVR library to preserve application data and enable flexible recovery. GVR provides a global view of array data, enabling an application to easily create, version and restore (partial or entire) arrays. In addition, GVR's convenient naming enables applications to flexibly compute across versions of single or multiple arrays. GVR users can control where (data structure) and when (timing and rate) array versioning is done, and tune the parameters according to the needs of the application. The ability to create multi-version array and partially materialize them, enables flexible recovery across versions. GVR has been used to demonstrate flexible multi-version rollback, forward error correction, and other creative recovery schemes [5,6]. Demonstrations include high-error rates, and results show modest runtime cost ($< 1\%$) and programming effort in full-scale molecular dynamics, Monte Carlo, adaptive mesh, and indirect linear solver applications [7,8].

GVR exploits both DRAM and high bandwidth and capacity burst buffers or other forms of non-volatile memory to enable low-cost, frequent versioning and retention of large numbers of versions. As needed, local disks and parallel file system can also be exploited for additional capacity. For example, NERSC Cori [9] supercomputer provides 1.8 PB SSDs in the burst buffer, with 1.7 TB/s aggregate bandwidth (6 GB/s per node). The JUQUEEN supercomputer at Jülich Supercomputing Center [10] is equipped with 2 TB flash memory, providing 2 GB/s bandwidth per node. Multi-versioning performance studies on JUQUEEN [10] showed GVR is able to create versions at full bandwidth, demonstrating low cost versioning is a reality [11]. In this paper, GVR's low-cost versioning enables flexible recovery for ABFR.



(a) 2D N-Body Spatial Domain



(b) Quad-Tree Representation

Figure 1: Barnes-Hut Quad-Tree Computation for 2D N-Body Simulation

2.2 N-body Computations

The N-body problem is the problem of predicting the motions of a dynamical system of objects, under the influence of physical forces, e.g. gravity. N-body simulations are a fundamental tool in the study of physical systems, from investigating three-body systems like the Earth-Moon-Sun to understanding the evolution of star clusters [12].

Over the past years, a number of methods have been introduced to solve N-body problem. The direct-summation method computes and integrates the pairwise forces on each particle with all others, in which the computation increases as $\mathcal{O}(N^2)$. Much effort [13,14,15,16] has been expended to reduce the complexity by approximating the contribution of many particles with a single interaction, resulting in complexity of $\mathcal{O}(N \log N)$. Among them, "tree codes" [15,17,18] are widely deployed, which use a tree structure to organize particles and group distant particles into one larger cell, allowing their gravity to be accounted for a single force. Barnes-Hut [17] is a commonly used tree algorithm, consisting of two major steps: first construct the tree and then compute the force of each particle by walking the tree.

Tree construction: A root node is used to encompass the full mass distribution. In 2D simulation, the space is repeatedly subdivided into four daughter nodes of half the side length each, until one ends up with single particles (see

Figure 1). After the topology of the tree has been constructed, the contents (mass, position) of each node are initialized by a post-order tree traversal.

Force computation: For each particle, forces are obtained by traversing the tree, i.e. starting at the root node, a decision is made whether or not to open a node (i.e. continue the tree walk) to provide an accurate enough partial force. Thus the error is controlled conveniently by the opening criterion, because higher accuracy is obtained by walking the tree to lower levels. The Barnes-Hut opening criterion determines if a node is sufficiently far away by computing l/D, where l is the length of the region represented by the node, and D the distance between the node's center-of-mass and the particle. If $l/D < \theta$ (i.e. opening criterion), then approximate the particles in the node by their center of mass. Otherwise, continue the tree walk. The typical value of θ ranges from 0.3 to 0.8.

The reconstruction of full tree at each step can lead to significant overhead. As a result, the dominant time of simulation is spent on tree construction rather than force computation. McMillan and Aarseth [19] first discussed that the geometric structure of tree evolves slowly in time, therefore it is sufficient to reconstruct the tree once in a while to take into account the slow changes in the tree hierarchy. Gadget [20] proposed a dynamic tree update scheme, in which the tree node is updated without reconstructing the full tree. The tree reconstruction frequency can be controlled to improve computation efficiency. In our study, we adopt the dynamic tree update scheme and allow tree nodes to be updated with tunable frequency.

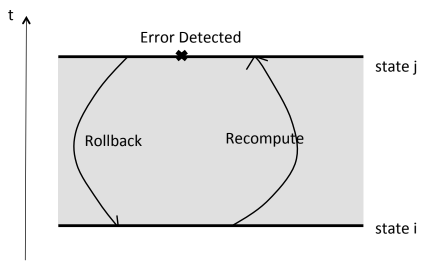
3 Algorithm-Based Focused Recovery (ABFR)

We propose to use the Algorithm-Based Focused Recovery (ABFR) approach [4] for N-body computations. ABFR exploits application semantics and versioned states to bound error impact and further localize recovery. ABFR exploits application algorithmics and data flow to identify potential root causes of a latent error and focus recovery effort on a small subset (see Figure 2b). ABFR allows recovery to be overlapped with computation, reducing recovery overhead and enabling tolerance of high error rates. In contrast, checkpoint-restart (CR) (Figure 2a) blindly rolls back the entire computation to the last verified checkpoint and recomputes everything.

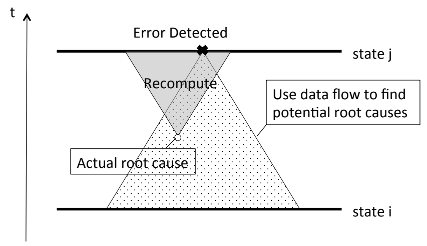
We assume that a latent error detector (or "error check") is available. Such detectors are application-specific and computationally expensive. In order to keep the model general, we make the following assumptions:

- The error detector has 100%[6] coverage, finding some manifestation whenever there is an error, but not precisely identifying all manifestations.
- The error check detects error manifestations in the data, namely, corrupted values and their locations.

[6] Errors that cannot be detected are beyond the ability of any error recovery system to consider.



(a) Blind CR system recovery



(b) ABFR recovery based on application knowledge

Figure 2: Checkpoint Restart (CR) vs. Algorithm-based Focused Recovery (ABFR).

– Because latent ("silent") errors are complex to identify, the detector is computationally expensive.[7]

The interval between two consecutive error detections bounds the error latency. Given the error location and timing, three steps are performed to correct the state of corrupted data.

1. **Inverse propagation:** application logic and dataflow is used to inverse error propagation, identifying all data points in past that could have contributed to this error manifestation. These data points are called potential root causes (**PRC**). For N-Body tree computations, the errors may reside in leave nodes (i.e. no propagation) or propagate to some up-level nodes depending on the latency. Nodes that have interacted with the detected erroneous node in the latency bound are considered as PRCs. Therefore the tree structure and the error latency bound are used to invert error propagation and identify PRCs.
2. **Diagnosis:** to bound error impact more precisely, PRCs can be tested (diagnosis), eliminating many of the initial PRCs. For N-Body tree computations, this can be accomplished by recomputing intermediate states from versions (courtesy of GVR) and comparing to previously saved results. If the values match, the PRC can be pruned.
3. **Recovery:** recovery is applied to the reduced set of PRCs and their downstream error propagation paths. For instance, recovery can be recomputing PRCs and particles that have interacted with PRCs in the latency bound.

4 Analytical Performance Model

In this section, we introduce the model. We start with the application framework before detailing all error-related and fault-tolerance parameters. Table 1 summarizes main notations.

[7] Assuming expensive checks means that any improvements in checking can be incorporated – cost is not a disqualifier.

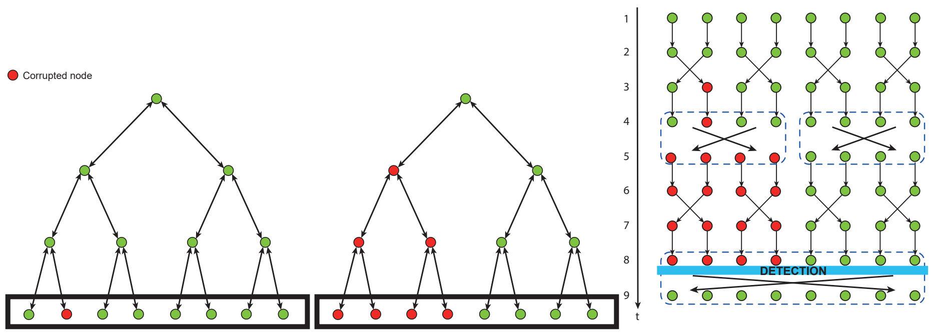
Definitions	
n	Height of tree
K	Number of iterations performed at level n (tree leaves)
Error Rate	
λ	Errors per second per leaf
Time	
c	Time to compute one leaf
d	Time to detect errors on one leaf
v	Time to version one leaf
r	Time to recover one leaf
Tree-wise	
T_c	Time to compute the tree without errors
T_d	Time for detection the tree without errors
T_v	Time for versioning the tree without errors
Frequency	
D	Detection interval of the form $2^x \cdot K$

Table 1: Summary of main notations.

Level	Iterations							
	1	2	3	4	5	6	7	8
0								2^0
1				2^1				2^1
2		2^2		2^2		2^2		2^2
3	$K \cdot 2^3$	$K \cdot 2^3$	$K \cdot 2^3$	$K \cdot 2^3$	$K \cdot 2^3$	$K \cdot 2^3$	$K \cdot 2^3$	$K \cdot 2^3$

Table 2: Iterations and number of nodes updated each level for $\mathcal{T}_3$, with $n = 3$.

Application model. We consider a perfect binary tree $\mathcal{T}_n$ of depth n. Leaves at the bottom of the tree hold the original data and perform computations, while internal nodes operate by aggregating the data of their two children and keeping a *summary*. The root is at level 0, and leaves are at level n. Nodes at different levels are updated with different rates: these rates are decreasing from bottom to top, so that leaves are updated the most frequently, while the root is updated the least frequently. The execution proceeds through iterations with global period 2^n. Each iteration consists of K computing steps at leaf level n, plus some information propagation, first bottom-up and then top-down, to exchange summary data. The scope of the propagation across the tree varies as follows. Every odd iteration is limited to level n nodes (leaves), without any propagation. Iteration number $2j$ with j odd is a depth-1 propagation that goes up to level $n-1$ nodes and then back to the leaves. Iteration number $4j$ with j odd is a depth-2 propagation that goes up to level $n-2$ nodes and then back to the leaves. More generally, iteration number $2^i j$ with j odd is a depth-i propagation that goes up to level $n-i$ nodes and then back to the leaves. Hence the root is first updated at iteration 2^n. Note that the root is updated only once per global period of 2^n iterations, which represent $2^n K$ computing steps at leaf level. The value of parameter K is application-dependent. See an illustration with $n = 3$ on Table 2, which also shows how many nodes per iteration are updated at the different levels.



(a) Tree at iteration $iter = 3$ (b) Tree at iteration $iter = 5$ (c) Leaves at each iteration

Figure 3: Computation of a tree with $x = n = 3$. An error strikes a node at $t = 3$ (a). The error propagates to neighboring nodes following the communication pattern and four leaves are corrupted at $t = 5$ (b). Detection is done on the leaves at $t = 8$ after computation but before propagation (c).

During a global period of 2^n iterations, the 2^n leaves execute K computation steps every iteration, hence the total computing cost is $T_c = 4^n \cdot K \cdot c$, where c denote the time of a computing step at the bottom level.

Error model. An error can strike at any time during the computations of the leaves. When an error occurs, it produces a localized error on a leaf, as shown in Figure 3a. This error then spreads to other nodes every time data is exchanged, i.e., during iterations $1, 4, 6, 8$ in Table 2 and in Figure 3c). This results in several leaves being corrupted after a few iterations (as shown by Figure 3b). In particular, the whole tree will be corrupted after the root has been updated and the data has been sent back to all nodes, which happens every 2^n iterations (this corresponds to iteration 8 in Table 2). We assume that errors strike following an Exponential probability distribution. Let λ denote the error rate per leaf node, so that $1 - e^{-\lambda c}$ is the probability of having an error during the computation of one leaf, and $1 - e^{-\lambda T_c}$ the probability of having an error during the computation of the whole global period. We also assume that at most one error strikes during the execution of one period.

Versioning. Versioning a leaf consists in saving its current state. We assume that the cost of versioning is low in front of the detection cost. For simplicity, we version the state of the leaves every K steps. Therefore the total time needed for versioning 2^n leaves, for a period of 2^n iterations, is $T_v = 2^n \cdot 2^n \cdot v$, where v is the time to version a leaf.

Detection. Let D denote the detection interval, i.e. the number of steps between two consecutive error checks. D will be chosen as D has a multiple of K. Detection is performed at level x every D time-steps. Finding the optimal value of x is part of the optimization problem to be solved.

The detector is applied after computations at leaf nodes and before propagation to upper level nodes. D is of the form $D = 2^x \cdot K$, where x is an arbitrary integer between 0 and n. Therefore the detection is performed 2^{n-x} times during a global period. If $x = n$, detection occurs only once while if $x = 0$, detection occurs every K steps, just as versioning. The total time for detection is $T_d = 2^{n-x} \cdot 2^n \cdot d$, where d is the time to apply the detector at a leaf.

We assume that the detector is perfect: it always detects the manifestation of the error if one has struck. Finding how many leaves have been affected by the error (after its striking and until detection), and how many nodes must be recomputed, is performed through *diagnosis* and *recomputation*, respectively.

5 Performance Study: Expected Cost

In this section, we derive exact formulas for the expected total cost of the Checkpoint-Restart (CR) and Application-Based Focused Recovery (ABFR) approaches.

5.1 CR

Theorem 1. *The expected total cost for executing a global period with a binary tree of depth n using the CR approach is given by:*

$$\mathbb{E}(T_{CR}) = (2 - e^{-\lambda 4^n Kc}) \cdot 4^n \cdot K \cdot c + 2^n \cdot (d + v) \ . \tag{1}$$

Proof. Let $\mathbb{E}(T_{CR})$ denote the expected cost for executing the entire period before checkpointing, using the standard CR approach. We first need to account for the cost of computation T_c. Detection and checkpointing are done at the end of the period, with cost $2^n d$ and $2^n s$ respectively, where s denotes the time to save the state of az leaf onto global storage. If an error occurs, with probability $(1 - e^{-\lambda 2 T_c})$, all nodes need to be recomputed, with cost T_c again, from the last correct version. We can write:

$$\mathbb{E}(T_{CR}) = T_c + 2^n \cdot (d + v) + (1 - e^{-\lambda T_c}) T_c \ .$$

Then, setting $T_c = 4^n \cdot K \cdot c$ and simplifying, we retrieve Equation 1.

5.2 ABFR

Theorem 2. *The expected total cost for executing a global period with a binary tree of depth n using the ABFR approach is given by:*

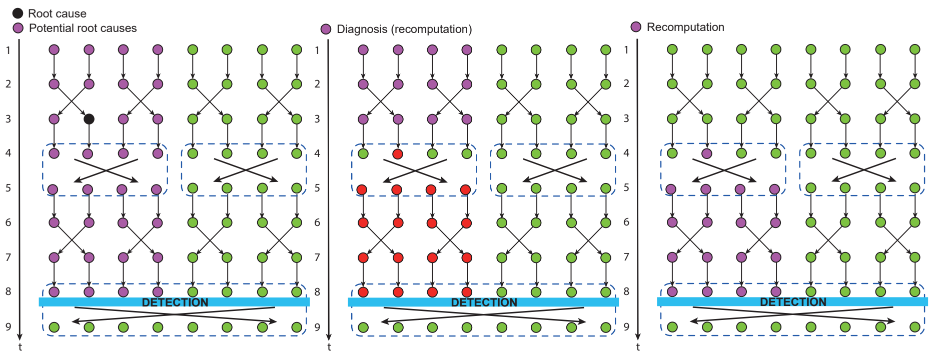
$$\begin{aligned}\mathbb{E}(T_{ABFR}) = {} & 4^n \cdot K \cdot c + 2^{n-x} \cdot 2^n \cdot d + 4^n \cdot v \\ & + (1 - e^{-\lambda 4^n Kc}) \left(\frac{1}{4}(4^x + 2^x)(Kc + r) + \frac{1}{6}(4^x - 1)(Kc + v) \right) \ . \end{aligned} \tag{2}$$

Proof. Let $\mathbb{E}(T_{ABFR})$ denote the expected cost for executing the entire period using the ABFR approach. We first need to account for the cost of computation T_c, the cost of detection T_d and the cost of versioning at every step T_v. Then, we need to account for the cost of diagnosis and recomputation in case of error. Let T_{diag} and T_{recomp} denote the time for diagnosis and recomputation, respectively. By definition, the probability that an error strikes during the period is given by $(1 - e^{-\lambda T_c})$, therefore we can write:

$$\mathbb{E}(T_{ABFR}) = T_c + T_d + T_v + (1 - e^{-\lambda T_c})(T_{diag} + T_{recomp}) \ .$$

Note that diagnosis and recomputation are random variables, because they depend upon when the error strikes. We take expectations and write:

$$\mathbb{E}(T_{ABFR}) = T_c + T_d + T_v + (1 - e^{-\lambda T_c}) \left(\mathbb{E}(T_{diag}) + \mathbb{E}(T_{recomp})\right) \tag{3}$$



(a) Inverse propagation (b) Diagnosis (c) Recomputation

Figure 4: Computation of the leaves with $x = n = 3$. (a) After an error has been detected, we use inverse propagation to identify the set of potential root causes; (b) Then we use diagnosis, i.e. we recompute potential root causes from the last correct version and check again old versions, to locate the root cause; (c) Finally, we recompute all corrupted nodes.

Inverse Propagation. When an error is detected, we can use inverse error-propagation to identify the set of potential root causes. The number of potential root causes depends on the detection interval $D = 2^x K$. Indeed, the error can only be located in the 2^{x-1} nodes connected to the manifestation of the error, as shown in Figure 4a.

With $x = n = 3$, this means that there is exactly one detection during the execution of the entire tree. Remember that detection is done after computations, but before propagation. Therefore in this example the number of potential root causes can be restricted to the 4 leaves (out of 8 leaves) that are directly linked to the manifestation of the error. Similarly, setting $x = 2$ and $n = 3$ means two detections during the execution of the tree and at most 2 potential root causes (out of 8 leaves).

Diagnosis. There are 2^x versions in one detection interval. Knowing that an error has occurred, the probability of having an error in each version is uniformly distributed. Thus the probability of having an error in version i (resp. iteration i) is $\frac{1}{2^x}$. Diagnosis is done by recomputing all potential root causes and comparing the result with previous versions (as shown in Figure 4b. We need to recompute (and reload) 2^{x-1} nodes i times in order to locate the root cause of the error. Thus the expected time for diagnosis is given by:

$$\begin{aligned}\mathbb{E}(T_{diag}) &= \sum_{i=1}^{2^x} \frac{1}{2^x} \cdot i \cdot 2^{x-1}(Kc + r) \\ &= 2^{x-2}(2^x + 1)(Kc + r) = \frac{1}{4}(4^x + 2^x)(Kc + r) \ . \end{aligned} \tag{4}$$

Recomputation. Recomputation follows diagnosis. The root cause of the error has been localized and we know which node must be recomputed, as shown in Figure 4c. As seen in diagnosis, when an error is detected, we only need to consider the 2^{x-1} nodes that are directly linked to the manifestation of the error. Now, depending on the actual location of the root cause.

The probability that an error strikes a node is uniformly distributed in space and time. Therefore, we start with two cases:

1. With probability $\frac{1}{2}$, the error has struck during the first 2^{x-1} iterations. This means that the error has propagated to all of the 2^{x-1} nodes in the last 2^{x-1} iterations, and that we must recompute *at least* 2^{x-1} nodes 2^{x-1} times.
2. With probability $\frac{1}{2}$ the error has struck in the other 2^{x-1} nodes and we don't need to recompute any of the first 2^{x-1} nodes.

We can write

$$\mathbb{E}(T_{recomp}) = \frac{1}{2}2^{x-1} \cdot 2^{x-1} \cdot (Kc + v) + \dots$$

Then, we have four cases:

1. With probability $\frac{1}{4}$, the error has struck in the *first* 2^{x-2} nodes of the *first* 2^{x-1} iterations.
2. With probability $\frac{1}{4}$, the error has struck in the *last* 2^{x-2} nodes of the *first* 2^{x-1} iterations.
3. With probability $\frac{1}{4}$, the error has struck in the *first* 2^{x-2} nodes of the *last* 2^{x-1} iterations.
4. With probability $\frac{1}{4}$, the error has struck in the *last* 2^{x-2} nodes of the *last* 2^{x-1} iterations.

In cases 1 and 3, we need to recompute *at least* 2^{x-2} nodes 2^{x-2} times. However, in cases 3 and 4, we do not need to recompute these nodes. We can write:

$$\mathbb{E}(T_{recomp}) = \frac{1}{2}2^{x-1} \cdot 2^{x-1} \cdot (Kc+v) + \frac{2}{4}(2^{x-2} \cdot 2^{x-2} \cdot (Kc+v) + \dots$$

This approach can be used recursively to compute the probability and cost of all possible scenarios (i.e. for all possible error locations). See Figure 5 for an example with $x = n = 3$. .We derive that:

$$\begin{aligned} \mathbb{E}(T_{recomp}) &= \sum_{i=1}^{x} \frac{2^{i-1}}{2^i} 2^{x-i} \cdot 2^{x-i} \cdot (Kc+v) \\ &= \sum_{i=1}^{x} \frac{1}{2} 4^{x-i} \cdot (Kc+v) = \frac{1}{6}(4^x - 1)(Kc+v) \ . \end{aligned} \tag{5}$$

$$\mathbb{E}(T_{recomp}) = \frac{1}{2}\left\{\begin{array}{l} 2^{n-1}2^{n-1}(K \cdot c + v) + \frac{1}{2}\left\{\begin{array}{l} 2^{n-2}2^{n-2}(K \cdot c + v) + \frac{1}{2}\left\{\begin{array}{l}(K \cdot c + v) \\ 0\end{array}\right. \\ \frac{1}{2}\left\{\begin{array}{l}(K \cdot c + v) \\ 0\end{array}\right. \end{array}\right. \\ \frac{1}{2}\left\{\begin{array}{l} 2^{n-2}2^{n-2}(K \cdot c + v) + \frac{1}{2}\left\{\begin{array}{l}(K \cdot c + v) \\ 0\end{array}\right. \\ \frac{1}{2}\left\{\begin{array}{l}(K \cdot c + v) \\ 0\end{array}\right. \end{array}\right. \end{array}\right.$$

Figure 5: Computation of the expected recomputation cost for $x = n = 3$ considering all 8 possible scenarios. The error can hit any one of the 8 iterations with uniform probability.

Expected cost. Altogether, putting expressions for diagnosis (see Equation 4) and recomputation (see Equation 5) back into Equation 3, we retrieve Equation 2.

6 Performance Analysis: Expected Overhead

In this section, we derive exact formulas for the overhead incurred by using the CR or ABFR approach. For either method, the expected overhead is defined as $\mathbb{E}(H_X) = \frac{\mathbb{E}(T_X)}{T_c} - 1$, where T_X denotes the cost for method X. Recall that T_c is the baseline cost, so that the overhead measures extra the fraction of work spent to mitigate the impact of errors.

6.1 CR

Let $\mathbb{E}(H_{CR})$ denote the expected overhead for CR. We can write:

$$\mathbb{E}(H_{CR}) = \frac{\mathbb{E}(T_{CR})}{T_c} - 1 \ .$$

Taking Equation 1 for $\mathbb{E}(T_{CR})$ and setting T_c to $4^n \cdot K \cdot c$, we obtain:

$$\mathbb{E}(H_{CR}) = 1 - e^{-\lambda 4^n Kc} + \frac{d+v}{2^n \cdot K \cdot c} \ . \tag{6}$$

6.2 ABFR

Let $\mathbb{E}(H_{ABFR})$ denote the expected overhead for ABFR. We can write:

$$\mathbb{E}(H_{ABFR}) = \frac{\mathbb{E}(T_{ABFR})}{T_c} - 1 \ .$$

Taking Equation 2 for $\mathbb{E}(T_{ABFR})$ and setting $T_c = 4^n \cdot K \cdot c$, we obtain:

$$\begin{aligned}\mathbb{E}(H_{ABFR}) = {} & \frac{4^n \cdot K \cdot c + 2^{n-x} \cdot 2^n \cdot d + 4^n \cdot v}{4^n Kc} \\ & + \frac{(1 - e^{-\lambda 4^n Kc})}{4^n Kc} \left(\frac{1}{4}(4^x + 2^x)Kc + \frac{1}{6}(4^x - 1)Kc) \right) \\ = {} & \frac{2^{-x}d + v}{Kc} + \frac{(1 - e^{-\lambda 4^n Kc})}{4^n Kc} \left(\frac{1}{4}(4^x + 2^x)(Kc + r) + \frac{1}{6}(4^x - 1)(Kc + v)) \right) \ .\end{aligned} \tag{7}$$

7 Optimal Detection Interval for ABFR

We now show how to derive the optimal detection interval for ABFR. Recall that the detection interval is of the form $D = 2^x \cdot K$. Our goal is to find the optimal value for x, denoted by x^*.

First, we use Taylor series to approximate $1 - e^{-\lambda 4^n Kc}$ to $\lambda 4^n Kc + O(\lambda^2)$, and derive that:

$$\mathbb{E}(H_{ABFR}) = \frac{2^{-x}d + v}{Kc} + \lambda \Big(\frac{4^x + 2^x}{4}(Kc + r) + \frac{4^x - 1}{6}(Kc + v) \Big) \ .$$

Then, in order to get the optimal value for x, denoted by x^*, we need to solve the following equation:

$$\frac{\partial \mathbb{E}(H_{ABFR})}{\partial x} = 0 \ , \tag{8}$$

Note that letting $y = 2^x$ in Equation (8) leads to solving a third-degree equation in y, so it is possible to obtain a closed-form expression for the optimal value y^*, and hence for x^*. In the following, we simply solve Equation (8) numerically, obtain the optimal solution as a real variable, and using nearest rounding to retrieve the optimal integer value. Finally, plugging x^* back into Equation 7, we obtain $\mathbb{E}(H^{opt}_{ABFR})$.

7.1 Limits of the Analysis

For the sake of simplicity, we have made the assumption that only one error can strike during the computation of a tree, meaning that (1) re-execution after an error always succeeds, and (2) diagnosis only needs to find one root cause. While this makes for a good approximation with large MTBE, the error rate can only get so small in the analysis. In particular, we must ensure that MTBE $>> T_c$ in order to keep the probability of having more than one error as low as possible.

Note that this is a common assumption when dealing with CR models. However there are several possible ways the model could be extended to handle multiple errors. First, multiple errors within a detection interval could trigger multiple ABFR responses. Alternatively, diagnosis and recovery could be extended to deal with multiple errors concurrently. These are promising directions for future work.

8 Simulations

In this section, we run a set of simulations whose goal is twofold: (1) show the accuracy of the theoretical analysis; and (2) assess the performance of the proposed ABFR approach against the standard CR approach. We describe the settings of the simulations in Section 8.1 and we present the results in Section 8.2.

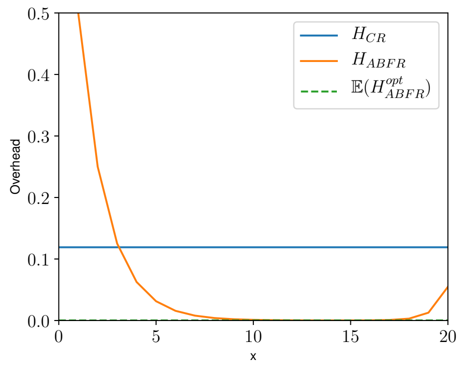
8.1 Settings

We target large platforms subject to silent errors. Such platforms can handle large simulations with millions of nodes, and we set $n = \lceil log_2(10^6) \rceil = 20$. The time needed to compute one node in N-Body computation is typically measured at around $c = 10^{-5}$s and we set the the number of iterations at the bottom level to $K = 100$. In addition, we assume that ABFR can take advantage of high bandwidth, high capacity burst buffers or other form of non-volatile memory to perform low-cost, frequent versioning, and we set cost to version and recover a node to $r = v = \frac{c}{100}$. Detection, on the opposite, is assumed to be expensive and we set the detection cost for one node to $d = 100 \cdot c$. Finally, we set the error rate to $\lambda = 1.15 \cdot 10^{-10}$, which corresponds to a MTBE of 275 years for a single processor (or one day on a platform with 100000 of such processors).

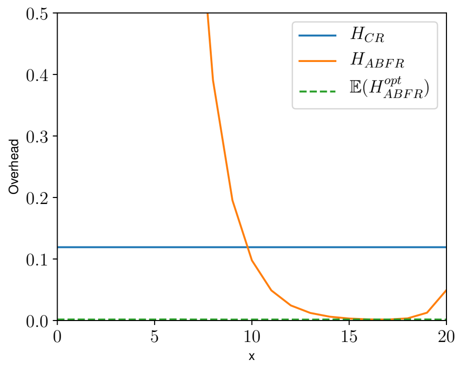
Simulations are based on the model and we instantiate the model using the above values by default. Errors are injected into the computation following the error rate λ. Note that at most one error is injected into the computation of a tree and that errors can strike any node with uniform probability. When an error strikes a node, diagnosis and recomputation are computed according to the exact number of nodes that need to be recomputed for diagnosis and recomputation, with respect to the error location. The overhead of the simulation is obtained by averaging the results of 1000 runs.

8.2 Results

In this section, we present the results of the simulations for different scenarios.



(a)



(b)

Figure 6: Overhead of CR and ABFR approaches for different values of x and with detection cost $d = 100 \cdot c$ (a) and $d = 10000 \cdot c$ (b).

Impact of Detection Interval. Figure 6 shows the expected overhead obtained using the CR and ABFR approach, denoted by H_{CR} and H_{ABFR}, respectively, for all possible values of x between 0 and n. We show the results for the default detection cost $d = 100 \cdot c$ (a) and for a much larger detection cost $d = 10000 \cdot c$ (b). In addition, we plot the theoretical optimal expected overhead for ABFR, denoted by $\mathbb{E}(H_{ABFR}^{opt})$, which is obtained with Equation 7 for the optimal value of x. By solving Equation 8 numerically, we find that $x^* = 14$ for $d = 100 \cdot c$ and $x^* = 17$ for $d = 10000 \cdot c$.

First, we observe that, in both cases, the optimal overhead obtained with the simulations closely matches the optimal theoretical overhead, which confirms the accuracy of the analysis. Then, we can see that both figures show a dramatic increase of the overhead for small values of x. This is because the detection interval D is of the form $2^x \cdot K$. This means that decreasing x (and therefore the detection interval D) causes an exponential increase in the number of detections, which in turns increases the overhead. In addition, we note a slight increase of the overhead for large values of x. Indeed, when the detection interval is too large, (e.g. only one detection at the end of the computation when $x = n = 20$), errors have more time to propagate and more nodes need to be recomputed as a result, which increases the recovery cost, and therefore the overhead.

While the optimal overhead is very sensitive to the detection interval, we observe (by comparing both scenarios) that it does not vary much as a function of the detection cost. This is because the detection cost only represents a small part of the total computation. Overall, we show that ABFR is able to improve the overhead by several orders of magnitude compared to the standard CR approach, and is up to 120 times more efficient with this setting.

Impact of Detection and Version Cost The detection cost d has almost no effect on the optimal overhead (as shown in the previous scenario). It does

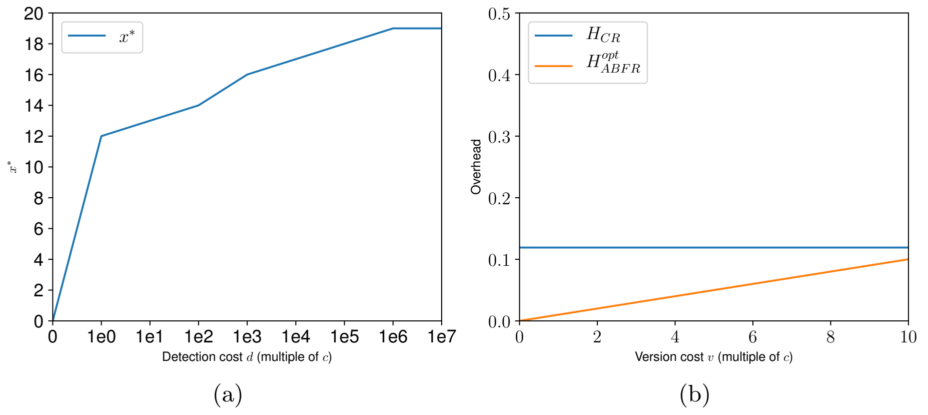


Figure 7: Impact of the detection cost on the optimal x (a) and impact of the versioning cost on the optimal overhead (b).

however have an impact on the optimal value of x. Figure 7 (a) shows the optimal x^* obtained for different detection costs. We can see that unless the detection cost is extremely small, the optimal x^* must be a trade-off between the detection cost and the recovery cost in case of error.

As opposed to the detection cost, the versioning cost v has no effect on the optimal x^*, but its value can have a significant impact on the overhead. Because all nodes are versioned, the overhead increases linearly with the version cost v, as shown in Figure 7 (b), and we must ensure that this cost remains cheap for ABFR to perform better than CR.

Impact of MTBE. Figure 8 (a) shows the overhead obtained with CR and ABFR for different MTBE ranging from 100 years to 1000. Figure 8 (b) shows the corresponding optimal detection interval for ABFR. We can see that ABFR scales much better than CR with high error rates ($MTBE < 400$ years). Where CR needs to recompute the entire tree in case of error, ABFR has the ability to detect the error earlier in the computation, and to recompute only a fraction of all the nodes. Indeed, the number of nodes to recompute in case of error depends on the size of the detection interval D and is at most $O(4^x)$ for ABFR, while it is exactly $\Theta(4^n)$ for CR. Note that for the same reason, ABFR remains better than CR even with low error rates ($MTBE > 1000$ years).

Impact of Error Latency. Figure 9 (a) shows the recovery cost of ABFR normalized with respect to the recovery cost of CR, while Figure 9 (b) shows the cost of diagnosis and recomputation normalized with respect to the recovery cost of ABFR. For the sake of simplicity, we simulated the execution of small trees with $x = n = 10$. Here there are $2^{10} \cdot K = 1024 \cdot K$ iterations in total. The x-axis denotes the number of iterations already done before the error occurred

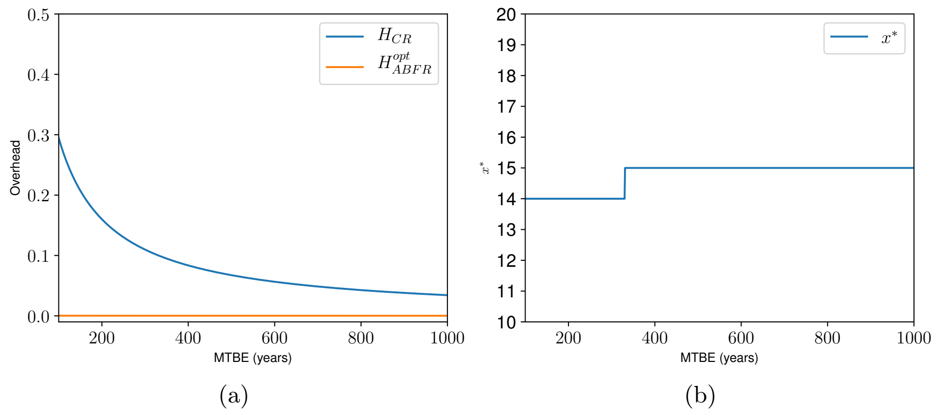


Figure 8: Overhead of CR and ABFR for different MTBE (a) and corresponding optimal detection interval x^* (b).

from 1 (first iteration) to $1024 \cdot K$ (last iteration). Because the detection interval x is set to $x = n = 10$, both CR and ABFR detect faults only at the end of the computation.

First, we note that the cost of diagnosis is linear, while the cost of recomputation is not. Indeed, diagnosis is done by recomputing all iterations from the last correct version until we find the error, while recomputation will skip part of the nodes depending on the location of the error. In particular, the spike that we can observe at iteration 512 corresponds to the biggest propagation step. If the error strikes right before this step, then exactly half of the remaining nodes must be recomputed. On the contrary, if the error strikes right after this step, it is not possible for the error to propagate further, and only one fourth of the remaining nodes needs to be recomputed. Note that, as shown in Section 5.2, we never need to recompute more than half of all the nodes, hence the recovery cost of ABFR is at worst 48% better than CR, and at best it is 65% better than CR.

9 Related Work

Latent errors, also known as silent errors or silent data corruption, represent a major threat to scientific applications executing on large scale platforms [21,22,23]. There are several causes of silent errors, such as cosmic radiation, packaging pollution, among others. Silent errors can strike the cache and memory (bit flips) as well as CPU operations; in the latter case they resemble floating-point errors due to improper rounding, but have a dramatically larger impact because any bit of the result, not only low-order mantissa bits, can be corrupted. In contrast to a fail-stop error whose detection is immediate, a latent error is identified only when the corrupted data leads to an unusual application behavior. This detection latency renders periodic checkpointing insufficient: if the error struck before

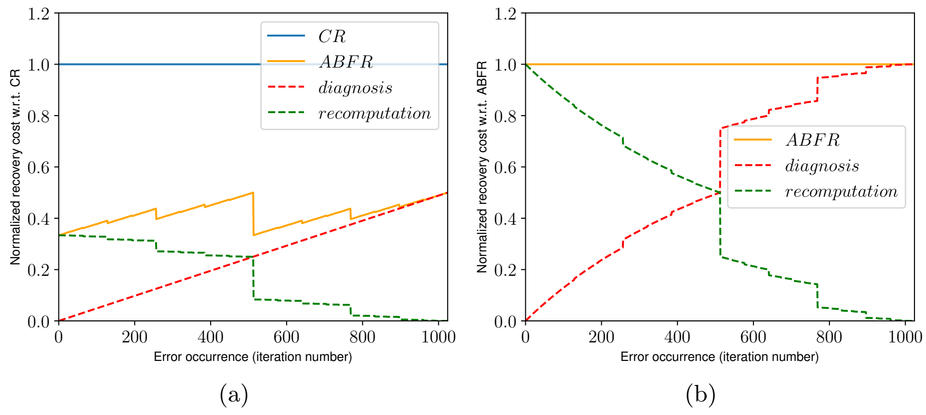


Figure 9: Recovery cost normalized w.r.t. CR (a) and w.r.t. ABFR (b) with detection interval set to $x = n = 10$.

the last checkpoint, and is detected after that checkpoint, then the checkpoint is corrupted and cannot be used for rollback. This is why checkpointing must be coupled with some verification mechanism, in order to detect any latent error before taking a new checkpoint.

Replication remains the most transparent and least intrusive technique and can be used at different levels (duplication, triplication or even more) . Combined with checkpointing, replication comes with two flavors: *process replication* [24,25] and *group replication* [26]. Process replication applies to message-passing applications with communicating processes. Each process is replicated, and the platform is composed of process pairs, or triplets. Group replication applies to black-box applications, whose parallel execution is replicated several times. The platform is partitioned into two halves (or three thirds). In both scenarios, results are compared before each checkpoint, which is taken only when both results (duplication) or two out of three results (triplication) coincide. If not, one or more silent errors have been detected, and the application rolls back to the last checkpoint. Note that duplication enables to detect but not to correct a latent error, while triplication enables both. Replication is not a new technique. Triple Modular Redundancy, or TMR [27], is the standard fault-tolerance approach for critical systems, such as embedded or aeronautical devices [28]. However, triplication has a high cost, since two-thirds of the processors are executing redundant work, and HPC scientists are not ready to pay such a price.

To address the problem of latent errors in HPC, many application-specific detectors have been proposed. Indeed, application-specific information enables ad-hoc solutions, which dramatically decrease the cost of error detection. Algorithm-based fault tolerance (ABFT) [29,30,31] is a well-known technique, which uses checksums to detect up to a certain number of errors in linear algebra kernels. Unfortunately, ABFT can only protect datasets in linear algebra kernels, and it must be implemented for each different kernel, which incurs a large amount of work for large HPC applications. Other techniques have also been advocated. Benson, Schmit and Schreiber [32] compare the result of a higher-order scheme with that of a lower-order one to detect errors in the numerical analysis of ODEs and PDEs. Sao and Vuduc [33] investigate self-stabilizing corrections after error detection in the conjugate gradient method. Bridges et al. [34] propose linear solvers to tolerant soft faults using selective reliability. Elliot et al. [35] design a fault-tolerant GMRES capable of converging despite latent errors. Bronevetsky and de Supinski [36] provide a comparative study of detection costs for iterative methods. Recently, several silent error detectors based on data analytics have been proposed, showing promising results. These detectors use several interpolation techniques such as time series prediction [37] and spatial multivariate interpolation [38,39,40]. Such techniques offer large detection coverage for a negligible overhead. However, these detectors do not guarantee full coverage; they can detect only a certain percentage of corruptions (i.e., partial verification with an imperfect recall). Nonetheless, the accuracy-to-cost ratios of these detectors are high, which makes them interesting alternatives at large scale. Similar detectors have also been designed to detect silent errors in the temperature data of the Orbital Thermal Imaging Spectrometer (OTIS) [41].

The ABFR approach presented in this paper is similar to ABFT approaches, exploiting application knowledge for error detection, but adding the use of application knowledge to diagnose what state is potentially corrupted, and using that knowledge to limit recomputation, and thereby achieve efficient recovery from latent errors. Recently, we have successfully applied ABFR to stencil computations [4], which are perfectly suited to ABFR due to their regular and neighbor-based communication pattern. The tree-based propagation pattern of N-Body computations is much more challenging for ABFR.

10 Conclusion

We have applied ABFR for N-Body tree computations to efficiently recover from latent errors. By exploiting application data flow and intermediate states, ABFR focuses recovery on an accurate estimate of potentially corrupted data, reducing recovery cost significantly. To explore the performance of ABFR, we build an analytical model parameterized by error rate and detection interval for a perfect binary tree. Simulation results show that ABFR reduces 50% of recovery overhead compared to checkpoint-restart approach. While the model is built for binary trees, it can be generalized to higher dimensions of simulations. Future directions include applying ABFR to production N-Body tree codes and demonstrating an application-agnostic ABFR runtime that supports portable and scalable performance.

References

1. Snir, M., Wisniewski, R.W., Abraham, J.A., Adve, S.V., Bagchi, S., Balaji, P., Belak, J., Bose, P., Cappello, F., Carlson, B., et al.: Addressing failures in exascale

computing. The International Journal of High Performance Computing Applications **28**(2) (2014) 129–173
2. Huang, K.H., Abraham, J.: Algorithm-based fault tolerance for matrix operations. IEEE Trans. Computers (1984)
3. Chen, Z.: Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In: PPoPP. (2013) 167–176
4. Fang, A., Cavelan, A., Robert, Y., Chien, A.A.: Resilience for stencil computations with latent errors. In: ICPP'2017, the 46th Int. Conf. on Parallel Processing, IEEE Computer Society Press (2017)
5. Dun, N., et al.: Data decomposition in monte carlo neutron transport simulations using global view arrays. Int. J. High Performance Computing Applications (2015)
6. Fang, A., Chien, A.A.: Applying GVR to Molecular Dynamics: Enabling Resilience for Scientific Computations. Technical Report TR-2014-04, University of Chicago (2014)
7. Chien, A., , et al.: Versioned distributed arrays for resilience in scientific applications: Global view resilience. Procedia Computer Science (2015)
8. Chien, A., et al.: Exploring versioned distributed arrays for resilience in scientific applications: global view resilience. Int. J. High Performance Computing Applications (2016)
9. Platform: NERSC CORI. https://www.nersc.gov/users/computational-systems/cori/
10. Platform: JUQUEEN. http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/JUQUEEN_node.html
11. Dun, N., et al.: Multi-versioning performance opportunities in bgas system for resilience. In: Int. Conf. High Performance Computing, Springer (2016)
12. Blelloch, G., Narlikar, G.: A practical comparison of n-body algorithms. In: Parallel Algorithms. Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society (1997)
13. Eastwood, J., Hockney, R.: Computer simulation using particles. New York: Mc GrawHill (1981)
14. Van Albada, G., Van Leer, B., Roberts Jr, W.: A comparative study of computational methods in cosmic gas dynamics. Astronomy and Astrophysics **108** (1982) 76–84
15. Appel, A.W.: An efficient program for many-body simulation. SIAM Journal on Scientific and Statistical Computing **6**(1) (1985) 85–103
16. Greengard, L., Rokhlin, V.: A fast algorithm for particle simulations. Journal of computational physics **73**(2) (1987) 325–348
17. Barnes, J., Hut, P.: A hierarchical o (n log n) force-calculation algorithm. nature **324**(6096) (1986) 446–449
18. Hernquist, L.: Performance characteristics of tree codes. The Astrophysical Journal Supplement Series **64** (1987) 715–734
19. McMillan, S.L., Aarseth, S.J.: An o (n log n) integration scheme for collisional stellar systems. The Astrophysical Journal **414** (1993) 200–212
20. Springel, V., Yoshida, N., White, S.D.: Gadget: a code for collisionless and gasdynamical cosmological simulations. New Astronomy **6**(2) (2001) 79–117
21. O'Gorman, T.: The effect of cosmic rays on the soft error rate of a DRAM at ground level. IEEE Trans. Electron Devices **41**(4) (1994) 553–557
22. Ziegler, J.F., Curtis, H.W., Muhlfeld, H.P., Montrose, C.J., Chin, B.: IBM experiments in soft fails in computer electronics. IBM J. Res. Dev. **40**(1) (1996) 3–18

23. Moody, A., Bronevetsky, G., Mohror, K., Supinski, B.R.d.: Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: SC, ACM (2010)
24. Ferreira, K., Stearley, J., Laros, J.H.I., Oldfield, R., Pedretti, K., Brightwell, R., Riesen, R., Bridges, P.G., Arnold, D.: Evaluating the Viability of Process Replication Reliability for Exascale Systems. In: SC'11, ACM (2011)
25. Fiala, D., Mueller, F., Engelmann, C., Riesen, R., Ferreira, K., Brightwell, R.: Detection and correction of silent data corruption for large-scale high-performance computing. In: SC, ACM (2012)
26. Casanova, H., Bougeret, M., Robert, Y., Vivien, F., Zaidouni, D.: Using group replication for resilience on exascale systems. Int. Journal of High Performance Computing Applications **28**(2) (2014) 210–224
27. Lyons, R.E., Vanderkulk, W.: The use of triple-modular redundancy to improve computer reliability. IBM J. Res. Dev. **6**(2) (1962) 200–209
28. Avizienis, A., Laprie, J., Randell, B., Landwehr, C.E.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Sec. Comput. **1**(1) (2004) 11–33
29. Huang, K.H., Abraham, J.A.: Algorithm-based fault tolerance for matrix operations. IEEE Trans. Comput. **33**(6) (1984) 518–528
30. Bosilca, G., Delmas, R., Dongarra, J., Langou, J.: Algorithm-based fault tolerance applied to high performance computing. J. Parallel Distrib. Comput. **69**(4) (2009) 410–416
31. Shantharam, M., Srinivasmurthy, S., Raghavan, P.: Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In: ICS, ACM (2012)
32. Benson, A.R., Schmit, S., Schreiber, R.: Silent error detection in numerical time-stepping schemes. Int. J. High Performance Computing Applications (2014)
33. Sao, P., Vuduc, R.: Self-stabilizing iterative solvers. In: ScalA '13. (2013)
34. Heroux, M., Hoemmen, M.: Fault-tolerant iterative methods via selective reliability. Research report SAND2011-3915 C, Sandia Nat. Lab. (2011)
35. Elliott, J., Hoemmen, M., Mueller, F.: Evaluating the impact of SDC on the GMRES iterative solver. In: IPDPS, IEEE (2014)
36. Bronevetsky, G., de Supinski, B.: Soft error vulnerability of iterative linear algebra methods. In: ICS, ACM (2008)
37. Berrocal, E., Bautista-Gomez, L., Di, S., Lan, Z., Cappello, F.: Lightweight silent data corruption detection based on runtime data analysis for HPC applications. In: HPDC, ACM (2015)
38. Bautista Gomez, L., Cappello, F.: Detecting silent data corruption through data dynamic monitoring for scientific applications. In: PPoPP, ACM (2014)
39. Bautista Gomez, L., Cappello, F.: Detecting and correcting data corruption in stencil applications through multivariate interpolation. In: FTS, IEEE (2015)
40. Bautista Gomez, L., Cappello, F.: Exploiting Spatial Smoothness in HPC Applications to Detect Silent Data Corruption. In: HPCC, IEEE (2015)
41. Ciocca, E., Koren, I., Koren, Z., Krishna, C.M., Katz, D.S.: Application-level fault tolerance in the orbital thermal imaging spectrometer. In: PRDC, IEEE (2004)