

Using Error Estimations for Detecting Silent Data Corruption in Numerical Integration Solvers

Mathematics and Computer Science Division

About Argonne National Laboratory

Argonne is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC under contract DE-AC02-06CH11357. The Laboratory's main facility is outside Chicago, at 9700 South Cass Avenue, Argonne, Illinois 60439. For information about Argonne and its pioneering science and technology programs, see www.anl.gov.

DOCUMENT AVAILABILITY

Online Access: U.S. Department of Energy (DOE) reports produced after 1991 and a growing number of pre-1991 documents are available free via DOE's SciTech Connect (<http://www.osti.gov/scitech/>)

Reports not in digital format may be purchased by the public from the National Technical Information Service (NTIS):

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Rd
Alexandria, VA 22312
www.ntis.gov
Phone: (800) 553-NTIS (6847) or (703) 605-6000
Fax: (703) 605-6900
Email: orders@ntis.gov

Reports not in digital format are available to DOE and DOE contractors from the Office of Scientific and Technical Information (OSTI):

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831-0062
www.osti.gov
Phone: (865) 576-8401
Fax: (865) 576-5728
Email: reports@osti.gov

Disclaimer

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor UChicago Argonne, LLC, nor any of their employees or officers, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of document authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof, Argonne National Laboratory, or UChicago Argonne, LLC.

Using Error Estimations for Detecting Silent Data Corruption in Numerical Integration Solvers

Prepared by

Dr. Franck Cappello
Pierre-Louis Guhur
Hisham Abou-Kandil
Dr. Tom Peterka

Mathematics and Computer Science Division, Argonne National Laboratory

September 30, 2016



ECOLE NORMALE SUPÉRIEURE DE CACHAN

ANNÉE DE RECHERCHE PRÉDOCTORALE À
L'ÉTRANGER

ARGONNE NATIONAL LABORATORY
MATHEMATICAL AND COMPUTER SCIENCES

Using Error Estimations for Detecting Silent Data Corruption in Numerical Integration Solvers

Author:
Pierre-Louis GUHUR

Supervisors:
Dr. Franck CAPPELLO
and Dr. Tom PETERKA

Director:
Dr. Hisham ABOU-KANDIL

September 2015 - July 2016

Contents

1	Introduction	6
1.1	Application Scenario	6
1.2	Contributions and organizations	6
2	Related Work	8
2.1	Resilience in High-Performance Computing	8
2.2	Numerical Integration Solver	8
2.2.1	Differential equation	8
2.2.2	Single-step and multi-step methods	9
2.2.3	Implicit and explicit methods	9
2.2.4	Function evaluations	9
2.2.5	Approximation error	10
2.2.6	Estimation of the approximation error	10
2.2.7	Adaptive solvers	12
2.3	Resilience to SDCs	14
2.3.1	Generic solutions	14
2.3.2	Algorithmic resilience	14
2.3.3	Fixed numerical integration solvers	14
2.3.4	Consequences of SDCs in Numerical Integration Solvers	15
2.4	SDC Injector	15
3	Model and Assumptions	17
3.1	Silent Data Corruption Model	17
3.2	Objectives	17
3.3	Workflow	17
3.4	Assumption on the solver	18
4	SDC Detection in Fixed Solver	19
4.1	The proposed <i>Hot Rod</i> method	19
4.2	First detector: <i>Hot Rod HR</i>	19
4.2.1	Threshold function	19
4.2.2	Detection of the significant SDC	20
4.2.3	A reliable training set	20
4.2.4	Adaptive control	21
4.3	Second detector: <i>Hot Rod LFP</i>	21
4.4	Algorithm	22
4.5	Experiments and results	22
4.5.1	Environment	22
4.5.2	Benchmark	23
4.5.3	Results	24
4.6	Conclusion	25
5	SDC Detection in Adaptive Solver	26
5.1	Simulations	26
5.2	Resilience of Adaptive Controllers	27
5.2.1	Inherent Resilience	27
5.2.2	Significant SDCs Not Detected	29

5.3	Resilience method for adaptive solvers	30
5.3.1	Double-checking based on Lagrange interpolating polynomials	31
5.3.2	Integration-based double-checking	31
5.4	Experiments	33
5.4.1	Cluster	34
5.4.2	Detection accuracy	35
5.4.3	Overheads	35
5.4.4	Scalability	36
5.5	Conclusion	36
6	Conclusion	38
7	References	39
8	Appendix	43

Abstract

Data corruption may arise from a wide variety of sources from aging hardware to ionizing radiation, and the risk of corruption increases with the computation scale. Corruptions may create failures, when execution crashes; or they may be silent, when the corruption remains undetected. I studied solutions to silent data corruptions for numerical integration solvers, which are particularly sensitive to corruptions. Numerical integration solvers are step-by-step methods that approximate the solution of a differential equation. Corruptions are not only propagated all along the resolution, but the solution could even diverge.

In numerical integration solvers, approximation error can be estimated at a low cost. I used these error estimates for detecting silent data corruptions in two high-performance applications in fault tolerance. On the one hand, I demonstrated a new lightweight detector for solvers with a fixed integration step size. I mathematically showed that all corruptions affecting the accuracy of a simulation are detected by our method. On the other hand, solvers with a variable integration size can naturally reject silent data corruptions during the selection of the next integration size. I showed that this mechanism alone can miss too many corruptions, and I developed a mechanism to improve it.

Acknowledgments

I wish to express my deep gratitude to my supervisors Franck Cappello and Tom Peterka without whose discussions, support and patience this work would not have been completed.

This study is done while I am associated with Argonne National Laboratory, LLC University of Chicago. The collaboration is done inside a pre-doctoral program from the Ecole Normale Supérieure de Cachan, France under Hisham Abou-Kandil's thoughtful guidance. I am also gratefully acknowledge the use of the services and facilities of the Decaf project at Argonne National Laboratory, supported by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357, program manager Lucy Nowell.

Abbreviations

AID Adaptive impact-driven which is an SDC detector developed by Di and Cappello

BBDC BDF-based double-checking which is an SDC detector

BDF Backward differentiation formula

BSS14 SDC detector developed by Benson et al.

EDBC Extrapolation-based double-checking which is an SDC detector

FLOPS Floating-point operations par second

FPR False positive rate

GTE Global truncation error

LTE Local truncation error

ODE Ordinary differential equation

PDE Partial differential equation

SDC Silent data corruption

SFNR False negative rate with steps injected by a significant SDC

TPR True positive rate

VAID A variant of AID with a variable step-size

WRF Weather research and forecasting model

1 Introduction

1.1 Application Scenario

The next step for high-performance computing is exascale computing. It attempts to compute at least 10^{18} floating-point operations per second (FLOPS). This technology has four main challenges: power consumption, memory storage, networks, and *resilience* [1]. Resilience can be defined as the ability of a system to cope with run-time errors. Sources of these errors are numerous, ranging from electromagnetic interference [2] to hardware aging. Their consequences are worrying: on the one hand, *fail-stop failures* conduct the execution to crash; on the other hand, *silent data corruptions* (SDCs) corrupt the results without any notification from the firmware or the operating system. Several reports [3, 4, 5] highlight that many scientific applications already suffer from corruptions. Snir et al. [6] showed that the situation is getting worse for exascale computers. The frequency of errors will increase possibly by a factor of 1000.

SDCs can be detected with replication. Replication duplicates an execution, and compares results between both executions. If results differ, an SDC is reported. But replication is limited by its overheads in memory and in computation, which are at least +100%.

Consequently, new solutions need to emerge to answer the challenge of resilience. In this report, I focus on numerical integration solvers. These solvers provide an approximation of the solution of a differential equation in time steps and discretized space. They are particularly sensitive to corruptions, because an SDC, that occurs at a certain step, is propagated in following steps, and because nonlinear problems with unstable dynamics tend to diverge in the presence of a corruption. Some solvers are qualified of *fixed*, because their time step size is fixed, whereas other solvers, qualified of *adaptive*, control their time step size based on an estimation of the approximation error and user-defined tolerances. Adaptive solvers are able to reject a step, when the approximation error appears to exceed the tolerances.

Not all SDCs need to be detected. Because of the inherent approximation error of a solver, some SDCs do not impact the accuracy of the results. At the opposite, SDCs that impact the user accuracy expectation are called *significant*.

Concerning fixed solvers, previous works [7, 8] attempt to compare a surrogate function to a threshold function at the end of a step. When the surrogate function exceeds the threshold function, an SDC is reported, and the step is recomputed. Because the surrogate function is based on curve-fitting methods, these SDC detectors are unlikely to detect SDCs that are higher than the approximation error of the solver, and thus unlikely to detect significant SDCs.

Chen et al. [9] showed that the rejection mechanism for adaptive solvers can reject some corruptions. But this mechanism is not reliable enough to reject all significant SDCs.

1.2 Contributions and organizations

My solution for SDC detection in fixed solvers is to compare two estimates of the approximation error of the solver. The two estimates are chosen such that they agree only in the absence of corruptions. I mathematically and experimentally showed that it allows to detect all significant SDCs. It improves the

trustworthiness of the results while avoiding wasting of resources to recover from insignificant SDCs. I performed experiments on a streamline integrator used for the visualization of the weather research and forecasting model (WRF) [10].

In the presence of an SDC, the error estimate is also corrupted. I showed that the error can be under-evaluated, and thus a significant SDC can be accepted by a solver. I suggest to double-check the acceptance of a step with a second error estimate. By using a second estimate with different terms, the probability that both estimates are corrupted is significantly reduced. In my experiments, the ratio of non-detected significant SDCs is reduced by a factor of 10.

Experiments were done in the context of high-performance computing by considering parallelized applications.

The remainder of this paper is organized as follows. In Section 2, I present the related works concerning resilience and numerical integration solvers. In Section 3, I present our model of SDCs and the assumptions of our study. In Section 4, I detail our SDC detector for fixed solver; in particular, I show that it detects all significant SDCs, and I compare with state-of-the-art detectors. In Section 5.2, I show that adaptive solvers are not able to reject all significant SDCs, and I present how a second estimate can be selected to double-check the acceptance of a step; then I test this mechanism on a cluster of 4096 cores.

2 Related Work

This part presents related work on high-performance computing, numerical solvers and fault tolerance.

2.1 Resilience in High-Performance Computing

The performance of a computer can be measured with the number of floating-point operations per second (FLOPS). In 1996, Intel's ASCI Red achieved the teraFLOPS (10^{12} FLOPS). The first computer to go petascale (10^{15} FLOPS) was IBM's Roadrunner, which sustained performance of 1.026 petaFLOPS according to the benchmark LINPACK [11].

High-performance computing works with multi-core processors: single processing units (cores) are combined on a same component. Multi-core processors can then be congregated into a node. Finally, nodes are put together in a rack. Communication costs are higher at each level. Usually, communications are done inside the message-passing interface standard (MPI) [12] which is implemented in particular in MPICH [13] or MVAPICH [14].

For example, Argonne National Laboratory has the fifth more performant computer in the world, MIRA. MIRA achieved a peak of 10 petaFLOPS with its 786,432 cores, 49,152 nodes and 48 racks. However, it consumes 3.9 MW. If the exascale computing were reached by adding more processors to current architectures, the power consumption might achieve 300 MW. New strategies to detect SDCs must limit computational overheads to avoid an upsurge of power consumption.

2.2 Numerical Integration Solver

Numerical integration solvers are used by a broad family of scientific applications, including engineering, physics, biology and economy. These solvers approximate the integration of a differential equation. They are iterative, time-stepping methods.

2.2.1 Differential equation

Numerical integration solvers attempt to approximate the solution of a differential equation. If the differential equation contains one independent variable, it is called an ordinary differential equation (ODE), whereas with multiple independent variables it is called a partial differential equation (PDE). A PDE may be solved with the method of lines, where all but one variable is discretized. In this case, the solution of a PDE is approximated by solving several ODEs. We define an *ODE method* as the numerical method that solves an *initial value problem*, formulated as

$$x'(t) = f(t, x(t)), x(t_0) = x_0,$$

with $t_0 \in \mathbb{R}$, $x_0 \in \mathbb{R}$, $x : \mathbb{R} \rightarrow \mathbb{R}^m$, and $f : \mathbb{R} \times \mathbb{R}^m \rightarrow \mathbb{R}^m$; f is L -Lipschitz continuous.

ODE methods approximate the exact solution of the ODE $x(t_n)$ into x_n , with $n \in 1, \dots, N$, $t_n = t_0 + nh$, and $h \in \mathbb{R}_+^*$ is the step size.

2.2.2 Single-step and multi-step methods

Integration methods are often classified in two categories: single-step and multi-step methods. Single-step methods compute several function evaluations between the current step and the previous step. At contrary, multi-step methods depend on several previous steps, but they usually do not compute function evaluations between two steps.

Most single-step methods belong to the Runge-Kutta family of methods, and so are most of the methods employed in this study. Runge-Kutta methods can be represented into a Butcher table, as in Table 1.

c_1	a_{11}	a_{12}	\dots	a_{1s}
c_2	a_{21}	a_{22}	\dots	a_{2s}
\vdots	\vdots	\vdots	\ddots	\vdots
c_s	a_{s1}	a_{s2}	\dots	a_{ss}
	b_1	b_2	\dots	b_s

Table 1. Butcher table of a single-step method

with s the number of *stages* $(a_{ij}) \in \mathbb{R}^s \times \mathbb{R}^s$, $(c_i) \in \mathbb{R}^s$, and $(b_i) \in \mathbb{R}^s$. Given the solution x_n at tep n , the solution at the next step is approximated in:

$$x_{n+1} = x_n + h \sum_{i=1}^s b_i K_i,$$

$$\forall i \leq s, K_i = f \left(t_n + c_i h, x_n + h \sum_{j=1}^{i-1} a_{ij} f_{n,j} \right).$$

2.2.3 Implicit and explicit methods

ODE methods can be explicit or implicit. Explicit methods compute the step n from previous steps, whereas implicit methods also use the current step n . Implicit methods require solving a system of equations. This extracomputation is worthwhile when implicit methods can use larger step sizes than explicit methods. This is the case for stiff problems.

For example, the Butcher table of an explicit method has the following property: $a_{i,j} = 0$ if $i \geq j$. A classic implicit method is the backward Euler method: $x_{n+1} = x_n + h_{n+1} \cdot f(t_{n+1}, x_{n+1})$.

2.2.4 Function evaluations

ODE methods are composed of several terms that require a function evaluation. We denote those terms (K_i) . For example, in explicit Runge-Kutta methods, $K_i = f \left(t_n + c_i h, x_n + h \sum_{j=1}^{i-1} a_{ij} f_{n,j} \right)$.

Function evaluations are the most computationally expensive part of a resolution. Therefore, SDCs are more likely to affect them.

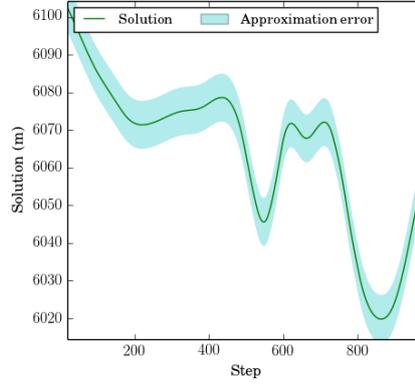


Figure 2. The local truncation error of the approximated solution is the blue interval.

2.2.5 Approximation error

Numerical solvers give nothing but an approximation of the solution of the differential equation. They have an inherent approximation error, as illustrated in Figure 2. The local truncation error (LTE) is the approximation error introduced at a step $n + 1$, whereas the global truncation error (GTE) is the absolute difference between the exact solution $x(t_n)$ and the approximated value x_n . Given the step size h , an ODE method is said to have an order p if $LTE_n = O(h^{p+1})$, and the global truncation error (GTE) is $GTE_N = O(h^p)$, where N is the last step. If $u(t, n)$ is the solution of the initial value problem: $u'(t, n) = f(t, u(t, n))$, $u(t_n, n) = x_n$, then the LTE at step n is $LTE_n = |x(t_n) - u(t_n, n - 1)|$.

2.2.6 Estimation of the approximation error

Since the 60s, a rich literature has been developed to provide an estimation of the approximation error.

For practical reasons, most of the solvers compute only an estimation of the LTE. However, methods [15, 16] exist to estimate the GTE. We present thereafter four methods for estimating the LTE. These methods are used in following sections. They rely on the same idea. First, two approximations x_n and \tilde{x}_n of the solution are computed at different order p and $q < p$. Hence \tilde{x}_n is supposed to be less accurate than x_n . Secondly, the difference between these two approximations is an estimation of less accurate approximation, \tilde{x}_n . The latter proposition is verified by:

$$\tilde{x}_n - x_n = \tilde{x}_n - u(t_n, n - 1) - (u(t_n, n - 1) - x_n), \quad (1)$$

$$= LTE[\tilde{x}]_n - LTE[x]_n, \quad (2)$$

$$= LTE[\tilde{x}]_n + O(h^p). \quad (3)$$

1. Embedded methods *Embedded methods* are the most widely used method for estimating the error. These methods are designed to compute the two approximated solution x_n and \tilde{x}_n from two ODE methods that share as many as possible function evaluations. The solution is propagated by one

of these results, while its $(K_i)_i$ defined in 2.2.1 are reused to compute the other result in order to achieve a low overhead.

2. Radau's quadrature

Another way for estimating LTE is suggested by Stoller and Morrison [17] and extended by Ceschino and Kuntzmann [18]. Relying on Radau's quadrature and Taylor's expansion, Ceschino and Kuntzmann give an expression of the LTE of a method given its order $p \leq 5$. The estimate \mathcal{R} , called here *Radau's estimate*, does not require the computation of any extra-stage, but it checkpoints previous stages and solutions. Therefore, it has a memory overhead, rather than a computational overhead like the embedded method. Since \mathcal{E} is a sixth order estimate, we use the following estimate \mathcal{R} presented by Butcher and Johnston [19]:

$$\begin{aligned}\mathcal{R} &= \frac{h}{10} [f(t_{n-3}, x_{n-3}) + 6f(t_{n-2}, x_{n-2}) + 3f(t_{n-1}, x_{n-1})] \\ &\quad + \frac{1}{30} [10x_{n-3} + 9x_{n-2} - 18x_{n-1} - x_n] \\ &= LTE_n^p + O(h^{p+2}).\end{aligned}$$

3. Richardson's extrapolation

Richardson's extrapolation can also be employed for estimating LTE. An ODE method at order p is applied firstly t_n to $t_n + h/2$ and then from $t_n + h/2$ to $t_n + h$, providing a solution \tilde{x}_n , and then from t_n to $t_n + h$ providing the solution x_n . As explained by Butcher [20], the estimation LTE is deduced from:

$$(1 - 2^{-p})^{-1} (\tilde{x} - \tilde{x}). \quad (4)$$

4. **Backward Differentiation Formula** To compute \tilde{x}_n , one can also employ a backward differentiation formula (BDF). BDF is a family of multistep implicit methods. They can also be used directly to compute the estimates by storing $(x_{n-k})_{k \geq 0}$. BDF methods achieve, at maximum, order 5. One could also use an Adam-Moulton method: it requires storing $f(t_{n-k}, x_{n-k})$ instead, which is often less practical.

Expressions of the backward differentiation formula can also be derived for an adaptive solver at several orders. The two first formula are given in [21]. For the first order:

$$\tilde{x}_n = x_{n-1} + hf(x_n).$$

For the second order:

$$\tilde{x}_n = (1 + \omega_n)^2 / (1 + 2\omega)x_{n-1} - \omega_n^2 / (1 + 2\omega)x_{n-2} + hf(x_n).$$

For the third order:

$$\begin{aligned}\tilde{x}_n &= \frac{(w_n + 1)^2 (w_{n-1} (w_n + 1) + 1)^2}{(w_{n-1} + 1) (2w_n + w_{n-1} (w_n + 1) (3w_n + 1) + 1)} x_{n-1} \\ &- \frac{w_n^2 (w_{n-1} (w_n + 1) + 1)^2}{2w_n + w_{n-1} (w_n + 1) (3w_n + 1) + 1} x_{n-2} \\ &+ \frac{w_n^2 (w_n + 1)^2 w_{n-1}^3}{(w_{n-1} + 1) (2w_n + w_{n-1} (w_n + 1) (3w_n + 1) + 1)} x_{n-3} \\ &+ h_n \frac{(w_n + 1) (w_n w_{n-1} + w_{n-1} + 1)}{3w_{n-1} w_n^2 + 4w_{n-1} w_n + 2w_n + w_{n-1} + 1} f(x_n),\end{aligned}$$

where $\omega_n = \frac{h_n}{h_{n-1}}$ and $\omega_{n-1} = \frac{h_{n-2}}{h_{n-1}}$.

The BDF method computes an approximated solution \tilde{x}_n . The LEE is obtained from the difference $x_n - \tilde{x}_n$. By employing previous solutions $(x_{n-k})_{k,0}$ and current solution x_n computing by the ODE method, BDF requires only the computation of $f(x_n)$. For most ODE methods, however, $f(x_n)$ is used for the next step. In this case, there is no extracomputation when the step is accepted. Certain ODE methods called first-same-at-last already compute $f(x_n)$ at step n .

2.2.7 Adaptive solvers

Users can control the approximation error by selecting a step size or an ODE method that achieves their expectation. The choice is a difficult trade-off, because it is also directed by the unstability of the problem and by the execution time expectations. Indeed, the step size cannot exceed a certain region of stability, which depends on the function f and the employed ODE method.

Adaptive solvers aim at choosing the step size at each step according to user-defined tolerances. In details, adaptive solvers estimate the LTE or the GTE at the end of a step. Then, the step size is reduced when the error estimate is close to the tolerance, and it is increased when the error estimate is far from the tolerance. If the error estimate exceeds the tolerance, the step size is rejected.

Figure 3 shows the relevance of adaptive solvers. A differential equation proposed by Kulikov [15] is solved by an adaptive solver and a fixed solver with the same accuracy. The adaptive solver used only 156 steps, whereas the fixed solver used 250 steps.

1. Design of Adaptive Solvers

The user typically provides a desired absolute error tolerance Tol_A or a relative error tolerance Tol_R . In practice, the error estimate is based on LTE, so for every step the algorithm verifies that the estimated local truncation error satisfies the tolerances provided by the user and suggests a new step size to be taken. For Runge-Kutta methods, the LTE is obtained from an approximation of the solution. The method to compute this approximation is chosen in order to save computation or memory.

The adaptive controller at step n forms the acceptable error level and scaled level as

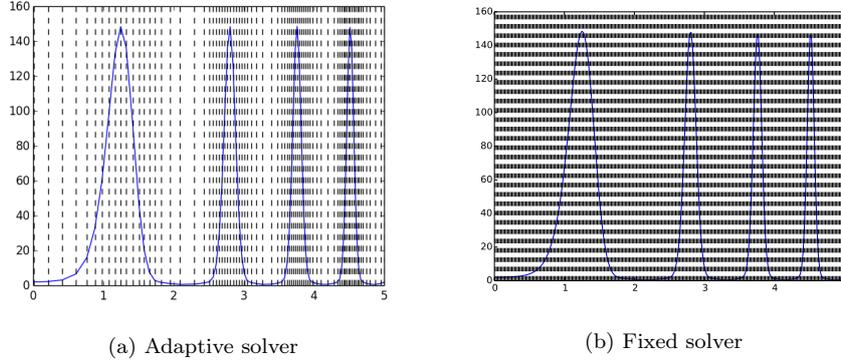


Figure 3. A differential equation is approximated with the same accuracy with an adaptive solver and a fixed solver.

$$Err_n = Tol_A + \|x_n\|Tol_R,$$

$$SErr_n = m^{\frac{1}{q}} \left\| \frac{|x_n - \tilde{x}_n|}{Err_n} \right\|_q,$$

where the errors are computed componentwise, m is the dimension of x , and q is typically 2 or ∞ (max norm). The error tolerances are satisfied when $SErr_k \leq 1.0$.

2. Estimating the local error truncation

Several LEEs can be employed for adaptive solvers. Usually, LEEs subtract x_n with an approximation of it, \tilde{x}_n , as explained in Equation (3).

Although estimates based on Richardson's extrapolation can be employed [22], the estimation is generally based on an embedded method. Embedded methods compute at each step two results at two different orders p and q : x_n^p and x_n^q (in general $|q - p| = 1$). The solution is propagated by one of these results, while the second result provides the approximation \tilde{x}_n that is used to compute an LEE at step n . If q is at a higher order than LTE^p , then the difference between x_n^p and x_n^q is an LEE of x_n^p :

$$x_n^p - x_n^q = LTE[x^p]_n - LTE[x^q]_n \quad (5)$$

$$= LTE[x^p]_n + O(h^{q+1}). \quad (6)$$

3. Control of error estimation

Based on this error estimate, in practice the step size that would satisfy the tolerances is

$$h_{\text{new}}(t_n) = h_{\text{old}}(t_n) \min(\alpha_{\text{max}}, \max(\alpha_{\text{min}}, A_{n+1})), \quad (7)$$

$$A_{n+1} = \alpha(1/SErr_{n+1})^{\frac{1}{p+1}},$$

where α_{\min} and α_{\max} keep the change in h to within a certain factor. $\alpha < 1$ is chosen so that there is some margin for which the tolerances are satisfied and so that the probability of rejection is decreased.

In this study we use the following settings: $\alpha = 0.9$, $\alpha_{\max} = 10$, $\alpha_{\min} = 0.1$ and $q = 2$. Therefore, the scaled error is $SErr = \sqrt{\frac{1}{n} \sum_n \frac{|x - \bar{x}|^2}{Err^2}}$, and the step size is adjusted as $h_{\text{new}} = h_{\text{old}} \min(10, \max(0.1, 0.9A_{n+1}))$.

4. Scheme of an adaptive controller

The adaptive controller works in the following way. After completing step n , if $SErr_n \leq 1.0$, then the step is accepted, and the next step is modified according to (7); otherwise the step is rejected and retaken with the step length computed in (7).

Numerical integration solvers have an inherent approximation error depending on the integration method and its order p : the GTE is $O(h^p)$. Because some low-order bits can be flipped without impacting a result, SDCs that affect those bits are called *insignificant*. Other SDCs affect higher-order bits: they increase the error and hence affect the user's accuracy requirement, or they may even cause the solver to diverge. Those are referred to as *significant*.

2.3 Resilience to SDCs

2.3.1 Generic solutions

The most generic solution for achieving the resilience to SDCs is replication [23]. It is implemented in RedMPI [24] (stands for redundant MPI), a recent variant of MPI. SDC detection is achieved by comparing results between an execution and its duplication. Correction can be obtained by using a variant called triple-modular redundancy [25]. In these cases, the overheads in memory and in computation are +200%.

At the hardware level, error-correcting code memory [26] consists in adding extra-memory bits and memory controller to verify and to correct SDCs. It makes memory, caches, and registers immune to SDCs.

2.3.2 Algorithmic resilience

At a higher level, resilience can be achieved by using algorithm properties. For example, Huang and Abraham [27] developed algorithm-based fault tolerance in the context of linear algebra. Several works have highlighted inherent resilient properties inside algorithms. For example, Pauli et al. [28] showed that even in presence of the nonrecoverable samples, Monte Carlo methods can still converge; and the authors provided recommendations to enhance resilience.

2.3.3 Fixed numerical integration solvers

In the context of fixed numerical integration solvers, several methods extract a surrogate function \mathcal{S} and compare \mathcal{S} to a threshold function \mathcal{T} . The step is validated when $|\mathcal{S}| < \mathcal{T}$.

The adaptive impact-driven detector (AID) [7] developed by Di and Cappello has been applied to numerical integration solvers, but it can be designed to any

iterative, time-stepping methods. \mathcal{S} is the difference between results at step n and a prediction of these results. If the results are too large to be stored, a sampling is done [29]. The prediction is obtained by an extrapolation method: the last value, a linear extrapolation, or a quadratic extrapolation. The method selected to the one that minimizes the error of extrapolation or the memory cost at a certain step, but the selection is often recomputed. \mathcal{S} is computed from the number of false positives, the maximum error of extrapolation, and a user-defined bound upon which an SDC is considered as unacceptable. Benson et al. developed an SDC detector [8] called BSS14. It computes an estimate of the approximation error based on an embedded method [30], Richardson estimate, linear extrapolation, or other specific estimates. \mathcal{S} is related to a relative difference to the last validated estimate and a ratio of the variance of previous estimates. \mathcal{S} is initialized by the user, but its values for each component are updated each time a step is accepted. Five parameters need to be set by the user. BSS14 and AID rely on extrapolation. Although extrapolation is easy to compute, it assumes a certain smoothness in the results. This is not always the case, especially for stiff problems.

For these SDC detectors, correction can be achieved with a rollback to the previous step. However, this requires to store this step.

2.3.4 Consequences of SDCs in Numerical Integration Solvers

Numerical integration solvers are particularly sensitive to SDCs: because of the iterative scheme, an SDC affects not only the corrupted step but also the following steps. We illustrate this sensitivity with two examples.

- In nonlinear ODEs, the stability region of the ODE method depends on the current step. An SDC can bring the solution outside the stability region. For example, in the equation $\frac{dx}{dt} = (x - 1)^2$, an initial point greater than 1 diverges to infinity, while an initial point less than 1 converges to 1.
- Even though the corruption is silent in the solver, it can produce corrupted results in the next stages of the application's workflow. For example, in image processing, feature extraction can be based on solving a PDE as shown by Zhou et al. [31]. If the PDE solution is incorrect, the iterative process of *level set evolution* may not converge.

2.4 SDC Injector

Recent papers on SDC detections propose different ways to inject SDCs.

In several papers [29, 32] injection were done by flipping randomly bits. In Guhur et al. [33], we compared several probability distributions to choose the position of the bit-flip. In the following, we refer to these kinds of injections by *singlebit* injections when one bit is flipped inside a data item, or *multibit* injections when several bits are flipped. The number of bit-flips in multibit injections is drawn from a uniform distribution.

A bit-flip on lowest-order positions may not have an impact on the results, while a bit-flip in highest-order positions may crash the application or be easy to detect. Consequently, Benson et al. [8] simulated SDC injections by multiplying a data item with a random factor. The factor is drawn from a normal

distribution with zero mean and unit variance. We refer to this method as *scaled injections*.

But some SDC may still have no impact on the results. One can also inject only significant singlebit corruptions. In the context of numerical integration solver, an SDC is considered as significant, when the difference between the corrupted result x_n^c and the uncorrupted result x_n^o is higher than a tenth of the approximation error: $|x_n^c - x_n^o| < LTE_n/10$

3 Model and Assumptions

3.1 Silent Data Corruption Model

A corruption is more likely to occur in data than in instructions, because instructions occupy less memory than data do. Moreover, corrupted instructions typically result in crashes and not silent corruptions. Other mechanisms besides SDC detection, such as checkpointings, may be employed for protecting an execution against instruction corruptions. We assume here that corruptions affect only data.

An SDC is called *nonsystematic* when it affects a program randomly. Such SDCs typically are triggered when radiation or aging hardware flips a bit. The probability of such SDCs is low and is unlikely to occur two times consecutively in the same step on the same data and the same bits. Therefore, recomputing a corrupted step to recover from a nonsystematic SDC is appropriate. On the contrary, a *systematic* corruption is triggered by a repeatable pattern such as a bug. In this study, we consider only nonsystematic SDCs.

We model an SDC as a random variable ϵ_i added to K_i . If K_i^o and K_i^c are respectively the noncorrupted and corrupted value of K_i , then $K_i^o = K_i$, and $K_i^c = K_i^o + \epsilon_i$.

3.2 Objectives

Replication is a generic solution for detecting all nonsystematic SDCs, but its overheads in memory and in compilation can not comply with exascale challenges in storage and in power consumption. New SDC detectors must have lower memory and/or computational overhead than does replication. For a numerical integration solver, SDC detection can be interpreted as a function of $(x_{n-k})_{k \geq 0}$ and $(f_{n-k,i})_{k \geq 0,i}$. Minimizing the computational overhead means computing as few operations as possible than those required by the ODE method. Minimizing the memory overhead is equivalent to storing as little extra data as possible.

3.3 Workflow

The numerical integration solvers represent one step in a scientific application. Figure 4 shows an overview of a typical high-performance computing workflow composed of a resilient numerical integration solver. The SDC detection is done at each step. When a step is found to be corrupted, it is recomputed in order to allow the solver to continue.

With the assumption of nonsystematic SDCs, the step, that is recomputed after being struck by an SDC, can not be affected by the same SDC. Consequently, the solutions before and after the recomputation differ. Even after the recomputation a corrupted step, the step might still be corrupted in the unlikely case of two nonsystematic SDCs in a row.

After a recomputation, if the step is identical to the previous one, then the step was not corrupted. It means that the detector made a false positive: it detected an SDC in a noncorrupted step. In order to avoid an infinite loop, the step is automatically accepted when the solution is the same before and after

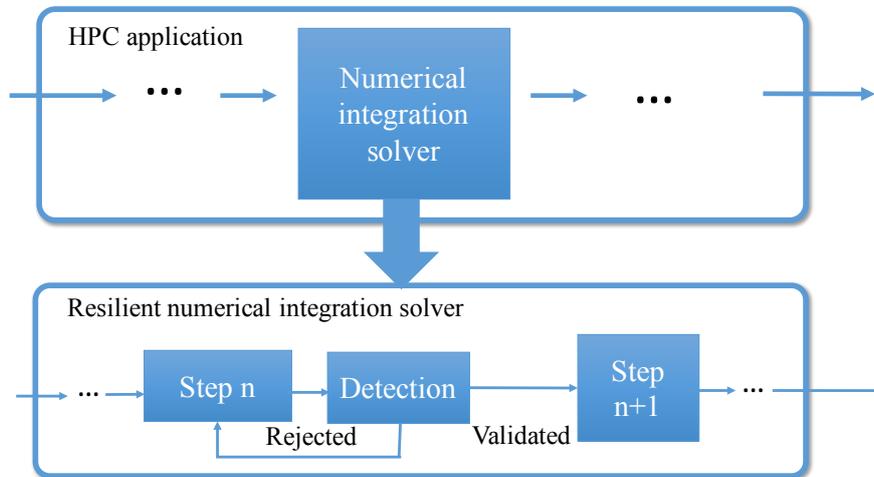


Figure 4. SDC detector for an HPC application with a numerical integration solver. At the end of each step, the SDC detector decides whether to validate or reject the step.

the recomputation. In practice, only the norm of the solution at a rejected step is stored to limit memory overheads.

The *false positive rate* is defined as the ratio between the number of false positives (a noncorrupted step that is reported corrupted) and the number of noncorrupted step, whereas the *true positive rate* is the ratio between the number of true positives (a corrupted step that is reported) and the number of corrupted steps. Because replication computes two times each step, we can consider that replication has a false positive rate of 100.0%.

3.4 Assumption on the solver

In the absence of SDCs, we assume that the solver works well. This means that it converges in a limited number of steps and achieves the user's accuracy expectations.

4 SDC Detection in Fixed Solver

An SDC detector was developed for fixed solver. It follows the state-of-the-art principle of comparing a surrogate function to a threshold function. The step is rejected as soon as the surrogate function exceeds the threshold function.

State-of-the-art detectors have two limitations that affect the detection performance. Firstly, a detector relying on extrapolation can not detect significant SDCs with an impact lower than the approximation error of the extrapolation. Secondly, their surrogate function is expected to be different from zero, but the threshold function only provide an upper bound of its value. A significant SDC that shifts the surrogate function below its expected value is not detected.

The proposed surrogate function is based on the difference between two estimates of the error. It is not affected by previous limitations, because it does not rely on the extrapolation, and its expected value is zero.

4.1 The proposed *Hot Rod* method

More specifically, the surrogate function Δ_n is defined by $\Delta_n = \mathcal{A}_n - \mathcal{B}_n$ with \mathcal{A} and \mathcal{B} two estimates of the error. For Cash-Karp's method, a single-step integration method, \mathcal{A} is the embedded estimate \mathcal{E} , and \mathcal{B} is Radau's estimate \mathcal{R} . In the absence of SDC, the surrogate function becomes $O(h^{p+2})$:

$$\begin{aligned}\Delta_n &= (\text{LTE}_n + O(h^{p+2})) - (\text{LTE}_n + O(h^{p+2})), \\ &= O(h^{p+2}).\end{aligned}$$

Two threshold functions were designed: *Hot Rod HR* (for High Recall) and *Hot Rod LFP* (for low false positives). In *Hot Rod HR*, the surrogate function is compared with a certain confidence interval centered over zero. When the surrogate function is outside the confidence interval, an SDC is reported. However, *Hot Rod HR* may have a false positive rate of a few percents. In *Hot Rod LFP*, a larger confidence interval is considered, in order to keep its false positive rate below one percent.

4.2 First detector: *Hot Rod HR*

In regular cases, the surrogate function is one order higher than the LTE. In presence of an SDC, Δ_n exceeds the threshold function. Hence, SDCs whose introduced errors are even smaller than the LTE are expected to be detected. I show that all significant SDCs are detected by *Hot Rod HR*.

4.2.1 Threshold function

Because $\Delta_n = O(h^{p+2})$, one can assume that Δ_n acts as a random variable, with a zero-mean in the absence of SDC. Obtaining an *a priori* expression of Δ_n is complex. Instead, a statistical evaluation is computed from a training set T composed of N_s samples. The samples are composed of the first computed Δ_n . More specifically, the standard deviation of Δ_n can be estimated from T with the unbiased sample standard deviation:

$$\sigma = \sqrt{\frac{1}{N-1} \sum_{n=1}^N \Delta_n^2}. \quad (8)$$

Assuming that $(\Delta_n)_n$ follows a normal distribution, the “three sigma rule” [34] suggests choosing $\mathcal{C}_n = 3\sigma$. Thus, we expect that 99.7% of uncorrupted $(\Delta_n)_n$ fall within the confidence interval, or in other words a false positive rate of 0.3%. The normal distribution is a natural choice for modeling the repartition of training samples.

4.2.2 Detection of the significant SDC

The SDC is detected when $|\Delta_n^c| \geq \mathcal{C}_n$ with \mathcal{C}_n the half-length of the threshold function at step n . It is all the more difficult to detect when $\Delta_n^o = 0$. I show that the threshold function is tight enough to detect all significant SDCs. This is done by showing that the minimum injected error ϵ_{min} that can be detected is of the same order than the approximation error.

As explained in Section 3.3, the corruption affects a function evaluation K_i^o such that $K_i^c = \epsilon - K_i^o$, where c (respectively o) denotes corrupted (respectively uncorrupted) data.

$$\begin{aligned} \Delta_n^c - \Delta_n^o &= \mathcal{E}_n^c - \mathcal{E}_n^o - (\mathcal{R}_n^c - \mathcal{R}_n^o), \\ &= h\epsilon \left[\hat{b}_i + b_i \left(\frac{31}{30} - \frac{3\delta_{i,1}}{20} \right) \right], \end{aligned}$$

where δ_{ij} is the Kronecker’s symbol defined by $\delta_{ij} = 1$ if $i = j$; otherwise $\delta_{ij} = 0$, (b_i) (respectively (\hat{b}_i)) are the coefficients of the order 4 (respectively 5) in Cash-Karp’s method.

The minimum error ϵ_{min} that we can detect corresponds to the case $|\Delta_n^c - \Delta_n^o| = \mathcal{C}_n - 0$. We note that $B = \hat{b}_i + b_i \left(\frac{1}{30} - \frac{3\delta_{i,1}}{10} \right)$. This leads to

$$\epsilon_{min} = \frac{\mathcal{C}_n}{hB} = O\left(\frac{\mathcal{C}_n}{h}\right).$$

When x_n is corrupted instead of a stage, one can derive that $\epsilon_{min} = O(\mathcal{C}_n)$. If \mathcal{C}_n has the same order of Δ_n , then 1. $\epsilon_{min} = O(h^{p+1})$ when an error is injected inside a stage and 2. $\epsilon_{min} = O(h^{p+2})$ when an error is injected inside a result. In other words, the threshold of detection has the same order as (or better than) the LTE of Cash-Karp’s method. This guarantees that all significant SDCs are detected.

4.2.3 A reliable training set

Because items from T are not labeled as trusted or untrusted samples, the evaluation of σ might be corrupted. It thus would jeopardize the confidence interval and thus the SDC detector. To improve reliability, we weighted each Δ_n with its own value. Equation (8) becomes

$$\begin{aligned} \Sigma &= \sum_{n=1}^N \exp(-\Delta_n^2), \\ \sigma &= \sqrt{\frac{1}{(N-1)\Sigma} \sum_{n=1}^N \exp(-\Delta_n^2) \Delta_n^2}. \end{aligned}$$

4.2.4 Adaptive control

The hypothesis of a normal distribution may be invalidated. We develop thus a correction of the threshold function based on false positives.

In Section 3.3, we showed that a false positive is reported, when a solution remains the same after its recomputation. Because of the “three sigma rule,” the FPR is expected to be 0.3%. If the FPR is an order of magnitude higher, at 3%, for k times, an online learning allows us to increase the threshold function. The latter is increased with a certain coefficient $1 + \alpha$ (α is a learning rate). \mathcal{C}_n becomes $\mathcal{C}_n = (1 + \alpha)^k \times 3\sigma$, where α fixes the rate of the adaptive control. Because $(1 + \alpha)^k = 1 + \alpha k + O(\alpha^2)$, α is taken as $1/(\max(FPR) \times N_{steps})$, where N_{steps} is the number of steps in the application and $\max(FPR)$ is the maximum acceptable false positive rate. Because a false positive requires the recomputation of a noncorrupted step, we suggest setting $\max(FPR)$ at 5% to limit the computational overhead. In our experiments, we have $N_{steps} = 1000$; thus $\alpha = 0.02$.

Thanks to the adaptive control, the training set requires a few step. In experiments, I have found that $N_s = 5$ samples are sufficient to initialize the threshold function.

4.3 Second detector: *Hot Rod LFP*

If the cost of a false positive is too high, *Hot Rod HR* is not suitable. Hence, we designed a second detector with a larger threshold function. Nonetheless, all significant SDCs must still be detected.

This new confidence interval is defined by

$$\mathfrak{C}_n = 10C_{99}(|\Delta| \in T).$$

C_{99} denotes the 99th percentile of the training set. The threshold function can be interpreted as a bound that is an order of magnitude bigger than the surrogate functions in the training set. Considering the 99th percentile instead of the maximum increases the reliability of the training set: a corrupted sample with a large value that would burst the threshold function, is rejected. Because this threshold is larger than the previous one, this detection performance is lower. Because the estimates are at order $p = 4$ for Cash-Karp’s method, the LTE at step n can be expressed as $LTE_n = Ch^{p+1} + O(h^{p+2})$. We show that the GTE at the last step N is still an order p as it used to be without corruption. We assume the probability that an SDC occurs and is accepted is small enough to guarantee that at worst only one SDC is accepted. The worst case is when this SDC is accepted at the first step, $n = 1$, and when $\mathfrak{C}_n = \Delta_n$. Hence, the introduced error is $LTE_1 = 10Ch^{p+1} + O(h^{p+2})$. Because $GTE_1 = LTE_1$, $GTE_1 = 10Ch^{p+1} + O(h^{p+2})$.

With $\tilde{x}(t, x_n)$ the notation in Section 2.2.1, $x(t) = \tilde{x}(t, x_0)$, and one can write that the GTE at a step $0 < n < N_{steps}$ is

$$\begin{aligned} |GTE_{n+1}| &= |x(t_{n+1}) - \tilde{x}(t_{n+1}, x_n) + \tilde{x}(t_{n+1}, x_n) - x_{n+1}|, \\ &\leq |x(t_{n+1}) - \tilde{x}(t_{n+1})| + |x_{n+1} - \tilde{x}(t_{n+1}, x_n)|. \end{aligned}$$

Because f is L -Lipschitz continuous, the Gronwall’s inequality [35] simplifies

the first term to

$$\begin{aligned} |x(t_{n+1}) - \tilde{x}(t, x_{n+1})| &\leq |\tilde{x}(t_n, x_0) - \tilde{x}(t, x_n)|e^{Lh}, \\ &= |GTE_n|e^{Lh}. \end{aligned}$$

The second term, $|x_{n+1} - \tilde{x}(t, x_{n+1})|$, is the LTE at step $n+1$ and so is evaluated at $Ch^{p+1} + O(h^{p+2})$. Denoting $\gamma = e^{Lh}$, we obtain

$$\begin{aligned} \frac{|GTE_{n+1}|}{\gamma^n} &\leq \frac{|GTE_n|}{\gamma^{n-1}} + \frac{Ch^{p+1}}{\gamma^n}, \\ &\leq \dots, \\ &\leq |GTE_1| + Ch^{p+1} \sum_{i=1}^n \frac{1}{\gamma^i}. \end{aligned}$$

Because $\sum_{i=1}^N 1/\gamma^i = (\gamma^N - 1)/\gamma^N(\gamma - 1)$ and $\gamma - 1 \geq Lh$, noting $\tau = Nh$, we obtain

$$|GTE_{n+1}| \leq 10Ch^{p+1} + \frac{Ch^p}{L} (e^{L\tau} - 1) + O(h^{p+2}).$$

At the last step, $GTE_N = O(h^p)$ is verified. The order of GTE is unchanged: the SDC is insignificant.

4.4 Algorithm

Two efficient detectors were presented. They differ in their tradeoffs: *Hot Rod HR* has a higher true positive rate and *Hot Rod LFP* has a lower false positive rate. We saw that undetected SDCs have no impact on the accuracy of the ODE method. They require fixing the learning rate α , but simple indications are given. One can thus derive two scenarios. If an SDC is likely to happen (it could be the case when the processor is not protected from SDC by ECC memory or other protection system), then *Hot Rod HR* is employed. Otherwise, employing *Hot Rod HR* allows us to detect all significant SDCs with fewer false positives. The scheme is illustrated in Algorithm 1 for a given detector.

4.5 Experiments and results

We showed theoretically that all significant SDCs are detected with Hot Rod. In this section, we evaluate the SDC detectors with a meteorology application.

4.5.1 Environment

Experiments were computed on a machine with four Intel Xeon E5620 CPUs (each with 4 cores and 8 threads), 12 GB RAM, and one NVIDIA Kepler K40 GPU with 12 GB memory. It was programmed in C++11 using CUDA. The application is particle tracing for streamline flow visualization [10],[36], [37]. Input data are velocity field of the weather provided by WRF. The center of Earth is the origin of the axis. The solver integrates the velocity field to compute the streamlines. It stops when the streamline goes outside the velocity field. Uncorrected streamlines can thus be shorter than they were supposed to be. The result can be seen in Figure 5.

```

while learning do
  step  $\leftarrow$  simulation(prev. step) ;
   $\Delta \leftarrow |\mathcal{A}(step, prev.steps) - \mathcal{B}(step, prev.steps)|$  ;
  TrainingSet.push( $\Delta$ ) ;
end
while new step do
  step  $\leftarrow$  simulation(prev. step) ;
   $\Delta \leftarrow |\mathcal{A}(step, prev.steps) - \mathcal{B}(step, prev.steps)|$  ;
  if (Detector == Hot Rod HR and  $\Delta \leq C_n$ ) or (Detector == Hot
  Rod LFP and  $\Delta \leq C_n$ ) then
    report("no error") ;
    accept step ;
  end
  else
    step  $\leftarrow$  simulation(prev. step) ;
     $\Delta' \leftarrow |\mathcal{A}(step, prev.steps) - \mathcal{B}(step, prev.steps)|$  ;
    if  $\Delta' = \Delta$  then
      report("false positive") ;
      if FPR > 3% then
        k++ ;
      end
    end
    accept step ;
  end
end

```

Algorithm 1: Pseudocode for the execution of our detectors

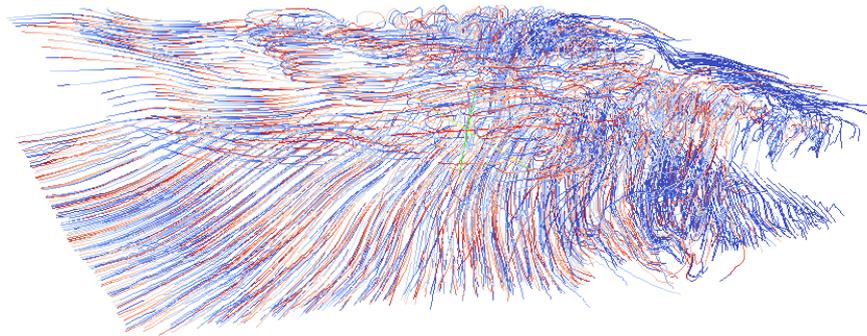


Figure 5. Streamlines computed by the application. The color gradient starts in red at seeds; 1,408 streamlines are computed.

4.5.2 Benchmark

I compared these SDC detectors with other detectors presented in Section 2: replication, AID and BSS14 detectors. The two latter detectors need to be parametrized. I selected the parameters that provide the best results in the

application. Using the same notation as in [7], I configured AID with $\theta r = 1$. Results were improved if the threshold function is taken as $(1 + \alpha)^k(\epsilon + \theta r)$ with $\alpha = 0.2$ and k defined in Section 4.2.4. Concerning BSS14, five parameters should be set, but no indication is detailed in [8] about two of them. With the notation of [8], the considered values are $\tau_j = 1e^{-5}$, $\tau_v = 0.02$, $\Gamma = 1.4$, $\gamma = 0.95$, and $p = 10$.

4.5.3 Results

Table 2. Benchmark of the detectors Hot Rod (H.R.) LFP and HR, replication, AID and BSS14. Values in the column “IRE 95%” are the injected relative errors (IRE) that were detected 95% of the time. Sign. = significant. Comp. = computational.

Detector	TPR (%)			FPR (%)	IRE 95%	Overheads (%)	
	Singlebit	Multibit	Sign.			Comp.	Memory
Replication	100.0	100.0	100.0	100.0	0.0	+100	+100
AID	14.3	43.2	86.7	1.6	$7e^{-6}$	+4.6	+50
BSS14	18.8	49.5	91.2	0.6	$4e^{-6}$	+3.7	+13
H.R. LFP	23.1	64.6	99.9	0.01	$7e^{-8}$	+3.8	+50
H.R. HR	28.6	69.6	99.9	1.2	$5e^{-9}$	+4.4	+50

Table 2 presents results from the benchmark. I did not compare each detector with a solver with no detector. I compared each detector with a perfect detector that returns the ground truth. For computational overhead, I divided the execution time of each detector with that of the perfect detector. My detectors have a computational overhead lower than 5%, as do the BSS14 and AID detectors. It is 20 times less computationally expensive than replication. But unlike the AID detector, my detectors have to employ an embedded integration method that computes more stages than does another Runge-Kutta method of the same order.

My detectors have a higher memory cost than does the BSS14 detector, but a smaller memory cost than does replication. For estimating memory overheads, I counted the number of stored vectors, such as solutions $(x_n)_n$, stage slopes $(k_i)_i$ and estimates. Cash-Karp’s method requires computing and storing two additional stage slopes than does Runge-Kutta 4, but the same number as the other embedded fourth-order methods. Cash-Karp’s method requires storing 6 $(k_i)_i$ (among them $f(x_{n-1})$), and x_n ; x_{n-1} is stored to allow a rollback in case of SDC detection; when $f(x_{n-1})$ is employed in the Radau estimation, $f(x_n)$ can be computed at the position (the result is employed at the next step if the step is accepted). Thus in total, 8 data elements are stored by the perfect detector, whereas \mathcal{E} (\mathcal{R} can use the same storage as \mathcal{E}), $f(x_{n-2})$, x_{n-2} and x_{n-3} are stored for our detectors; AID stores x_{n-2} , x_{n-3} , x_{n-4} , and the extrapolated solution; and BSS14 stores \mathcal{E} .

The true positive rate (TPR) shows that our detectors detect perfectly (at 99.9%) significant SDCs. Replication does as well, but the BSS14 and AID detectors have a TPR of 91.2% and 86.7% of significant SDCs, respectively. For BSS14 and AID, some SDCs can thus be undetected while affecting the accuracy of the solvers. Moreover, the “IRE 95%” measures the smallest injected relative error that is detected at least 95% of the times. Its value can be interpreted as the smallest detectable value. It is smaller for our detectors than the mean

local error estimate ($1.5e^{-6}$) by a factor of 100. Because all significant SDCs are detected, SDCs undetected by Hot Rod are sure to have no impact. The undetected 76.9% of SDCs by Hot Rod LFP are thus insignificant and do not need to be corrected: correcting these insignificant SDCs would not improve results and would demand extra computation. Figure 6 shows the *LTE* of the solver in the confidence interval in the absence of SDC. It represents the approximation error. As defined in Section 3, significant SDCs inject errors that are higher than this error. Because the streamlines of the AID and BSS14 detectors are pushed outside the confidence interval at SDC injections, they do not detect those SDCs. On the other hand, Hot Rod HR and LFP's streamlines are not affected by SDCs: these detectors protected the solver. This result is consistent with the fact that the IRE 95% of Hot Rod is two orders of magnitudes less than the approximation error.

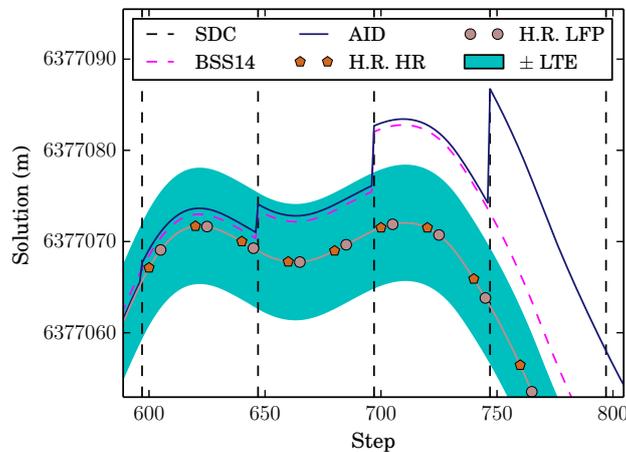


Figure 6. One streamline computed by the different detectors. Singlebit injection is made every 50 steps. In the window, the position of the bit-flip varies from 31 to 35 in IEEE754 doubleprecision. The interval “ $\pm LTE$ ” represents the approximation error. Significant SDCs shift the solution outside this interval. In the application, the origin is the center of the Earth.

4.6 Conclusion

This section presented two SDC detectors Hot Rod for fixed solvers. Both experimental and theoretical results show that all significant SDCs are detected. Except for replication, no other tested SDC detectors achieve these results. More specifically, compared with the state-of-the-art SDC detectors, the true positive rate is improved by 52% for singlebit corruptions; whereas compared with replication, the computational overhead is reduced by 20 times. Moreover, users need only to fix the learning rate α , as explained in Section 4.1.

My detectors were employed for one of the ODE integration methods. Other embedded Runge-Kutta methods can be directly employed. Radau’s estimates have a general expression in the case of adaptive step size; see the work of Butcher and Johnston [19]. For implicit methods or linear multisteps, Richardson’s estimates 3 can also be used. In future work, we plan to investigate detection in partial differential equation solvers.

5 SDC Detection in Adaptive Solver

In adaptive solvers, users define their tolerances in the approximation error. The tolerances are the maximum absolute and relative approximation errors that users accept. An SDC can thus be qualified of significant, when it shifts the solution to the point that the approximation error exceeds the tolerances. A typical workflow of an adaptive solver is illustrated in Figure 7. The adaptive

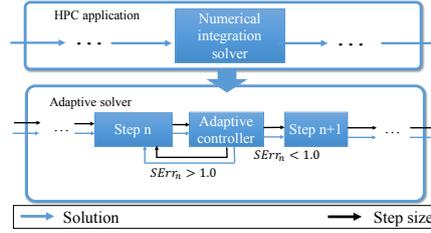


Figure 7. Scheme of the adaptive controller without our method (left) and with our method (right).

controller can reject a step. In particular, an SDC can increase the approximation error, and the adaptive controller is supposed to reject SDCs that exceed the tolerances. However, the approximation error is only estimated, and the estimate is also corrupted when an SDC strikes the solution. As a consequence, the corrupted estimate might underevaluate the error, and a significant SDC might be accepted.

Contrary to fixed solvers, finding two estimates that agree in the absence of SDCs is a challenge. Instead of using Hot Rod, I suggest a solution that consists in double-checking the acceptance of a step with a second error estimate. Because error estimates are difficult to compute in adaptive solvers, an algorithm based on the number of false positives select the estimation method.

5.1 Simulations

Simulations were done in a use case that solves the problem of a rising warm bubble in the atmosphere. The governing equations are the three-dimensional nonhydrostatic unified model of the atmosphere [38], expressed as

$$\begin{aligned} \frac{\partial \rho'}{\partial t} + \nabla \cdot (\rho \mathbf{u}) &= 0, \\ \frac{\partial \rho \mathbf{u}}{\partial t} + \nabla \cdot (\rho \mathbf{u} \otimes \mathbf{u}) &= -\nabla P' - \rho' \mathbf{g}, \\ \frac{\partial \rho \theta'}{\partial t} + \nabla \cdot (\rho \mathbf{u} \theta) &= 0, \end{aligned} \quad (9)$$

where ρ and P are density and pressure, respectively; \mathbf{u} is the flow velocity; \mathbf{g} is the gravitational force vector per unit mass; θ is the potential temperature; and $(\cdot)'$ denotes the perturbation to that quantity with respect to the hydrostatic mean value. The initial solution comprises a stationary atmosphere with $P = 10^5 \text{ N/m}^2$ and $\theta = 300 \text{ K}$, with a warm bubble defined as a potential temperature

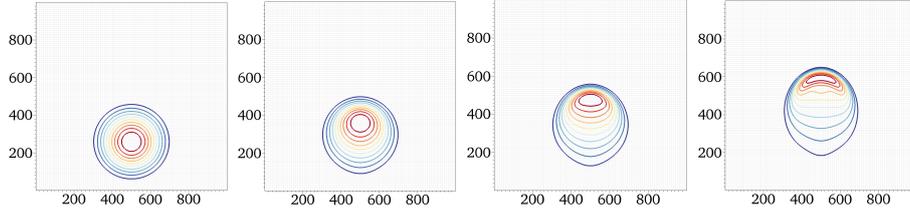


Figure 8. Rising thermal bubble: Density perturbation (ρ') contours at 0s (initial), 100s, 150s, and 200s (final). Ten contours are plotted between -0.0034 (red) and -0.0004 (blue). The cross-sectional profile is shown at $y = 500$ m.

perturbation [38],

$$\Delta\theta = \begin{cases} 0 & r > r_c \\ \frac{1}{2} \left[1 + \cos\left(\frac{\pi r}{r_c}\right) \right] & r \leq r_c \end{cases}, \quad (10)$$

where $r = \|\mathbf{x} - \mathbf{x}_c\|_2$, $r_c = 250$ m is the radius of the bubble, and $x_c = [500 \text{ m}, 500 \text{ m}, 260 \text{ m}]$ is the center of the bubble. The domain is a cube of side 1000 m, and no-flux boundary conditions are applied at all boundaries. The gravitational force \mathbf{g} is 9.8 m/s^2 along the z -axis.

The use case is solved with HyPar. The domain is discretized on equi-spaced Cartesian grids, and the 5th-order WENO [39] and CRWENO [40] schemes are used to compute the spatial derivatives. This computation results in an ODE in time that is solved by using the time integration methods implemented in PETSc. Figure 8 shows the density perturbation (ρ') contours for the rising thermal bubble case at 0s, 100s, 150s, and 200s, solved on a grid with 64^3 points. The bubble rises as a result buoyancy and deforms as a result of temperature and velocity gradients.

5.2 Resilience of Adaptive Controllers

Resilience to SDCs is one of the challenge to achieve the exascale computing. In Section 2, we saw that replication mandates an overhead in computation and in memory higher than +100.0%. Chen et al. [9] remarked that some solvers have an inherent resilience. In the following, we extend this point to all adaptive solvers. Experimentally, we observe that the adaptive controller rejects some steps where the error estimate exceeds a certain threshold due to an SDC.

However, this assumes that the adaptive controller is not corrupted in the presence of an SDC. This is not verified, because the error estimation used by the adaptive controller is computed from corrupted results. In Section 5.2.2, we observe that the error estimate can be shifted under the threshold of the adaptive controller.

5.2.1 Inherent Resilience

- Not all SDCs have an impact on the results

Numerical integration solvers have an inherent approximation error depending on the integration method and its order p : the GTE is $O(h^p)$.

When the lowest-order bit is flipped, the impact is insignificant with respect to the approximation error, and this SDC does not affect the accuracy of the results. Basically, we called an insignificant SDC any SDC that does not affect the user's expectation in accuracy. At the opposite, other SDCs affect higher-order bits, and then they drastically increase the error, or they may even cause the solver to diverge. These SDCs are referred to as *significant*.

It is difficult to distinguish a significant and insignificant SDCs in the general case. In our previous work on fixed solvers [33], the user does not give an explicit expectation in accuracy, and we considered that any SDC higher than a tenth of the LTE was significant. In the case of an adaptive solver, the user explicit the maximum acceptable approximation error with the tolerances Tol_A and Tol_R . Each time a step was corrupted, we measured thus the LTE with and without the corruption. When the error scaled by the tolerances was drifted above 1.0, the corruption was considered significant.

- Rejection of corrupted steps

In Section 2, we saw that an error estimate exceeding the tolerances Tol_A and Tol_R is rejected, because the approximation error is considered unacceptable for the user.

When an SDC occurs, and the approximation error is shifted outside the tolerance because of the SDC, the step is naturally rejected. In this case, the step size is reduced according to equation (6); then the next noncorrupted step observes that the error estimate is too small and increases the step size. Overall, the computation time is just increased during one step, while the accuracy is preserved.

The corrupted step can be unrejected, if the SDC shifts the approximation error below the tolerance, or if the SDC is small enough to avoid the approximation error to exceed the tolerance. Accepting such steps seems dangerous. One could object that the approximation error can be higher than it would have been without the SDC; even if the current step is below the tolerance, it might affect next steps. However, an adaptive solver is designed in a way that if all steps are below the tolerances, then the accuracy's expectation is achieved. Accepting such corrupted steps might increase the approximation error on next steps, but the accuracy's expectation will be achieved. Detecting and correcting such SDCs would thus be a waste of resources.

One caveat must be added. The approximation error is only estimated. In the presence of an SDC, the estimate is also corrupted, and its value might differ from the real value of the approximation error. This case is considered in Section 5.2.2.

We injected SDCs in the use case introduced in Section 2. In Table 3, we disclose the detection performances of the adaptive controller. Detection performances are based on the false positive rate and the true positive rate. The false positive rate is defined as the ratio between the number of non-corrupted steps that are rejected, and the number of non-corrupted steps. Similarly, the true positive rate is the ratio between the number of corrupted and rejected steps, and the number of corrupted steps.

The false positive rate remains below 0.1% for all considered ODE methods, thanks to α , α_{\max} , α_{\min} . At the same time, the true positive rate is usually below 50%. Singlebit SDCs are the hardest SDCs to detect (9.3%), whereas the multibit SDCs are the easier (55.1%). This results come from the fact that singlebit SDCs have usually a lower impact on the results. The true positive rate is decreasing the number N_k of the function evaluations $(K_i)_i$ of the ODE methods: for the Dormand-Prince fifth-order method, $N_k = 7$, $N_k = 4$ for the Bogacki-Shampine third-order method, and $N_k = 2$ for the Heun-Euler method. No explanation for now can explain this phenomenon.

The true positive rate can seem low, but only significant SDCs need to be rejected. Further experiments must thus distinguish significant to insignificant SDCs to know if the inherent resilience of adaptive solvers is reliable enough.

Rate	Injector	Heun-Euler	Bogacki-Shampine	Dormand-Prince
FP	All	0.0	0.0	0.0
TP	Scaled	31.1	23.3	20.1
TP	Multibit	55.1	46.8	35.3
TP	Singlebit	13.2	11.8	9.3

Table 3. Detection accuracy for several ODE methods and several SDC injectors. FP: false positive. TP: true positive. Results are given in percentage.

5.2.2 Significant SDCs Not Detected

The approximation error is not precisely known but is only estimated. In presence of a corruption, the estimate is also corrupted. In particular, it may be shifted below the tolerances of the adaptive controller; in such case, the step would be accepted. We can give several examples where it can happen.

- In the extreme case, the registers of $(K_i)_{i \geq 0}$ could be erased. In this case, the corrupted error estimate is equal to zero; consequently the step is accepted, and the step size is increased by α_{max} . The solution would be the same than during last step: $x_n = x_{n-1}$. The approximation error may then be unacceptable with respect to user's requirements.
- Because any K_i depends on other $(K_j)_{j \neq i}$, the corruption of a certain K_l corrupts the other $(L_j)_{j \neq l}$. Such cascading patterns increase the possibility of underestimating the approximation error.
- The SDCs can affect only the estimate. In this case, the estimate can be completely decorrelated from the approximation error.

Consequences of accepting a corrupted step can be disastrous. Not only the corrupted step exceeds the user's accuracy expectation, but the next steps will be initialized with a corrupted result. Moreover, the step size might be increased after the corrupted step, and it might even exceed its stability region; in such case, the solution may not converge at all.

In our usecase, we observe that this phenomenon can occur with a random corruption. In Table 4, we disclose the false negative rate of the classic adaptive

controller. The false negative rate is the ratio between the number accepted but corrupted steps, and the number of corrupted steps. Because not all SDCs have an impact on the accuracy of the results, we distinguish the case where a step is corrupted by any kind of SDC, and steps corrupted by at least one significant SDCs. In the latter case, the false negative rate is qualified of significant. The false negative rate with all steps is higher than the significant false negative rate, because insignificant SDCs can have a too low impact on the results to be detectable.

While the significant false negative rate with significant steps achieves 13.3% for Heun-Euler’s method with scaled SDCs, the rate bursts to 50.4% for Dormand-Prince’s method. This can be explained by the fact the number N_k of function evaluations $(K_i)_i$ is higher for Dormand-Prince’s method. In this case, more pattern of SDCs can lead to an underevaluation of the error estimation, and the probability of a non-detection is thus higher. While the false negative rate with all steps is higher with singlebit SDCs than with scaled SDCs, the significant false negative rate is lower with singlebit SDCs than with scaled SDCs. This is due to the fact that a singlebit SDC becomes significant when one of the highest-order bit is flipped, and this is easily detectable, whereas a scaled SDC can be significant while being difficult to detect.

Injection	Heun-Euler		Bogacki-Shampine		Dormand-Prince	
	All	Sign.	All	Sign.	All	Sign.
Singlebit	86.8	5.4	88.2	10.1	90.7	15.0
Multibit	44.9	3.9	53.2	4.5	64.7	7.9
Scaled	68.9	13.3	26.7	36.1	79.9	50.4

Table 4. False negative rate for several ODE methods and several SDC injectors. Sign. = significant (only steps, that corrupted with at least one significant SDC, are considered). Results are given in percentage.

5.3 Resilience method for adaptive solvers

We saw that the adaptive solvers are using an estimate to reject or accept a step. In the presence of SDCs, the adaptive solver can underestimate the approximation error because the estimator is using corrupted data; in this case, the adaptive solvers may not reject all significant SDCs. To solve this issue, I suggest to increase the redundancy of the rejection mechanism, by adding a second acceptance step. When the adaptive controller accepts a step, I apply a second rejection mechanism to validate the decision. The new workflow is illustrated in Figure 9. This idea can easily be interpreted, when I remark that the rejection mechanism could be underevaluated following its own pattern of corruptions. By selecting two rejection mechanisms with different patterns, the risk of the nondetection of a significant SDCs is reduced. I call *double-checking* this method.

I explore here two different approaches for computing the double-checking. Both of them compute first an estimate of the approximation error, and then compare the estimate to a threshold function. The first approach is inspired from AID and is presented in Section 5.3.1. The second approach consider an estimate based on another ODE method, and it is explained in Section 5.3.2.

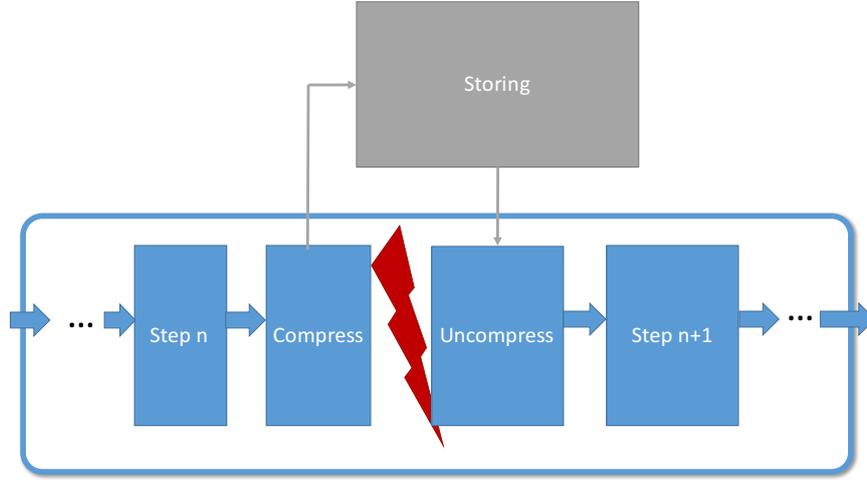


Figure 9. SDC detector for an HPC application with a numerical integration solver. At the end of each step, the SDC detector decides whether to validate or reject the step.

5.3.1 Double-checking based on Lagrange interpolating polynomials

The adaptive impact-driven detector (AID) [7] developed by Di and Cappello is designed to detect SDCs in an iterative, time-stepping methods with a fixed step size. In particular, it can be employed in the context of numerical integration solvers. Details on the method are given in Section 2.

First at all, I extended AID for variable step size. This was achieved by replacing the extrapolation methods to Lagrange interpolating polynomials (LIP). I provide formulations for order 0, 1, and 2:

$$\begin{aligned}\tilde{x}_n^0 &= x_{n-1}, \\ \tilde{x}_n^1 &= x_{n-1} \frac{h_n + h_{n-1}}{h_{n-1}} - x_{n-2} \frac{h_n}{h_{n-1}}, \\ \tilde{x}_n^2 &= x_{n-1} \frac{(h_n + h_{n-1})(h_n + h_{n-1} + h_{n-2})}{h_{n-2}(h_{n-2} + h_{n-1})} \\ &\quad - x_{n-2} \frac{h_n(h_n + h_{n-1} + h_{n-2})}{h_{n-2}h_{n-1}} \\ &\quad + x_{n-3} \frac{h_n(h_n + h_{n-1})}{h_{n-2}(h_{n-1} + h_{n-2})}.\end{aligned}$$

Secondly, I replaced the threshold function to adapt it for an adaptive solver, in which the user does not give the error bound θ , but the absolute and relative tolerances Tol_A and Tol_B .

The obtained double-checking is called *LIP-based double-checking*.

5.3.2 Integration-based double-checking

My second approach consists in computing another rejection mechanism based on a second error estimate. The second error estimate is computing from a

different ODE method than the one used in the solver. It must not require extra-computations, in order to reach a low computational overhead. It must also have a larger stability area than the ODE method used by the method. Because implicit methods have usually a larger stability area than explicit methods, the latter condition can be followed by employing an implicit method for the double-checking and an explicit method for the solver. The second method computes an approximated solution \tilde{x}_n . The error estimation is obtained from the difference $x_n - \tilde{x}_n$. It computes actually a local truncation error. I called this method an *integration-based double-checking*. The step is rejected when the norm of $x_n - \tilde{x}_n$ is higher than 1.0.

I suggest to employ a backward differentiation formula (BDF) for the double-checking, because it uses previously computations, and because it has a large stability area. I compute the estimates by storing $(x_{n-k})_{k \geq 0}$. One could also use an Adam-Moulton method: it requires storing $f(t_{n-k}, x_{n-k})$ instead, though it often appears less practical. BDF are multistep and implicit methods. In the literature, several expressions for a variable step size are given. The following expressions of \tilde{x}_n^1 , \tilde{x}_n^2 , and \tilde{x}_n^3 for the orders one, two and three are employed:

$$\begin{aligned}\tilde{x}_n^1 &= x_{n-1} + hf(x_n), \\ \tilde{x}_n^2 &= \frac{(1 + \omega_n)^2}{1 + 2\omega} x_{n-1} - \frac{\omega_n^2}{1 + 2\omega} x_{n-2} + hf(x_n), \\ \tilde{x}_n^3 &= h_n \frac{(w_n + 1)(w_n w_{n-1} + w_{n-1} + 1)}{3w_{n-1} w_n^2 + 4w_{n-1} w_n + 2w_n + w_{n-1} + 1} f(x_n) \\ &\quad + \frac{(w_n + 1)^2 (w_{n-1} (w_n + 1) + 1)^2}{(w_{n-1} + 1)(2w_n + w_{n-1}(w_n + 1)(3w_n + 1) + 1)} x_{n-1} \\ &\quad - \frac{w_n^2 (w_{n-1} (w_n + 1) + 1)^2}{2w_n + w_{n-1} (w_n + 1)(3w_n + 1) + 1} x_{n-2} \\ &\quad + \frac{w_n^2 (w_n + 1)^2 w_{n-1}^3}{(w_{n-1} + 1)(2w_n + w_{n-1}(w_n + 1)(3w_n + 1) + 1)} x_{n-3}\end{aligned}$$

where $\omega_n = \frac{h_n}{h_{n-1}}$ and $\omega_{n-1} = \frac{h_{n-2}}{h_{n-1}}$.

BDF methods have expressions until the order 6, but the stability area is decreasing with the order. At the same time, ODE methods with a small q requires less computation and less storage of previous solutions $(x_{n-k})_{k \geq 0}$. In this study, we restrict to the orders 1, 2, and 3, in order to prevent our system from stability issues and to mitigate the overheads.

By employing previous solutions $(x_{n-k})_{k \geq 0}$ and current solution x_n computing by the ODE method, BDF requires only the computation of $f(x_n)$. For most ODE methods, however, $f(x_n)$ is used for the next step. In this case, there is no extracomputation when the step is accepted. Certain ODE methods called first-same-at-last already compute $f(x_n)$ at step n , such as Dormand-Prince's method.

- Choice of the order

The estimation of the approximation error is using solutions computed at the order p . Thus, the error estimate does not exceed an accuracy higher than $O(h^{p+1})$, even if the second ODE method is expressed at an higher order $q > p$. However, \tilde{x}_n tends to be more similar to x_n with

an higher value of q . Consequently, the higher q is, the smaller the error estimate tends to be. It makes the detection less sensitive: the second error estimate is less likely higher than 1.0, and less steps tend to be rejected. This also means that the number of false positives decrease: less non-corrupted steps are rejected.

Because we want to improve the detection while maintaining a low false positive rate, we propose to adapt the order of the ODE method. When the false positive rate is higher than γ for an order q , a formula with one higher order $q' \leq q_{max}$ is considered. On the contrary, when the FPR is lower than Γ , the order of the ODE method is decreased to $q' = q - 1 \geq 1$. Γ can be chosen as the maximum false positive rate we can accept. γ should be lower but in the same of order of magnitude than Γ . In our experiments, we took $\gamma = 0.05$, $\Gamma = 0.1$, and $q_{max} = 3$. This procedure is explained in Section 2. The selection of the order is every $c_{max} = 10$ times or when the detector made a false positive.

- Implementation

The implementation was directly done inside the adaptive controller. This allows to reuse some allocation in memory to compute the second estimate. Thereafter, we refer the adaptive controller without double-check mechanism as *classic adaptive controller*. Because x_{n-1} is already stored by the classic adaptive controller, the double-checking only requires the storage of x_{n-2} and x_{n-3} . The implementation is illustrated in Section 10.

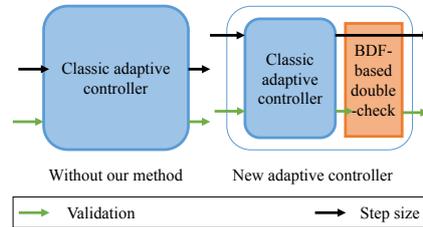


Figure 10. Scheme of the adaptive controller without our method (left) and with our method (right). The SDC detector can reject a step but does not change the step size.

5.4 Experiments

In Section 5.2, we saw that the rejection mechanism of an adaptive solver is able to correct only a part of the SDCs. Some SDCs, though significant, remained in the solution, because the rejection mechanism was corrupted and did not detect any outlier. Then, I proposed a method that enhance the rejection mechanism by double-checking the acceptance of a step in Section 4.1.

In this section, we experimentally validate our method with the use case introduced in Section 5.1. First, we show that our method reduces significantly the risk of accepting a significant SDC. Secondly, we measure the overheads and the scalability of our double-checkings, in order to compare them with replication and to suggest some improvements.

Numerical experiments use HyPar [41], a high-order, conservative finite-difference solver for hyperbolic-parabolic PDEs. It also uses the time integra-

```

Data:  $(x_{n-k})_{k \geq 0}, f(x_n), q, N_{steps}$ 
Result: Rejection or validation of step  $n$ 
rejected = True;
SErr1 = Estimating1(( $x_{n-k}$ ) $k \geq 0$ ,  $f(x_n)$ );
// Eq. (3)
if  $c++ == c_{max}$  then
    /* Update ODE method's order  $q$  */
     $c_{max} = 0$ ;
    if  $FP_q/N_{steps} < \gamma$  then
        |  $q = \max(1, q - 1)$ 
    else if  $FP_q/N_{steps} > \Gamma$  then
        |  $q = \min(q_{max}, q + 1)$ 
    end
if SErr1 == lastSErr then
    /* Case of a false positive */
    validation = True;
     $FP_q++$ ;
     $c = c_{max}$ 
else
    bool validation = SErr < 1.0;
    if validation then
        |  $SErr_2 = Estimating_2((x_{n-k})_{k \geq 0}, f(x_n), q)$ ;
        | validation = SErr2 < 1.0;
        | lastSErr = SErr1;
    end
end
if validation then
    |  $n++$ ;
    | rejected = False;
    |  $h = NewStepSize(Serr_1, h)$ ;
    | // Eq. (7)
end

```

Algorithm 2: Adaptive-controller with integration-based double-checking

tors (ODE solvers) implemented in PETSc [42, 43, 44], a portable and scalable toolkit for scientific applications. HyPar and PETSc are written in C and use the MPICH library on distributed computing platforms.

5.4.1 Cluster

The first case was computed on the cluster Blues at Argonne National Laboratory. The cluster is composed of 310 compute nodes, 64 GB of memory on each node, 16 cores per compute node with the microarchitecture Intel Sandy Bridge and a theoretical peak performance of 107.8 TFlops. PETSc was configured with MVAPICH2-1.9.5, shared libraries, 64 bit ints, and O3 flag.

	FPR	TPR	Significant FNR
Classic	0.0	31.1	13.3
LBDC	2.3	33.1	4.1
IBDC	4.2	41.9	1.1
Replication	100.0	100.0	0.0

Table 5. Our double-checking based on Lagrange interpolation polynomials (LBDC) and on a numerical integration method (IBDC) are compared with the expensive state-of-the-art replication, and the classic adaptive controller without our enhancement (Classic). FPR = false positive rate. TPR = true positive rate. FNR = false negative rate.

5.4.2 Detection accuracy

We applied the integration-based double-checking and the LIP-based double-checking to the Heun-Euler method. Table 5 compares their detection performances with replication, and the classic adaptive controller. Details on rates are given in Section 5.2. We consider that replication has a false positive rate of 100%, because all steps are recomputed, at least once.

LIP-based double-checking reduces the rate of significant false negatives with a factor of 3, whereas integration-based double-checking decreases the rate with a factor of 10. This difference of accuracy results from the fact that the error estimate used by integration-based double-checking is more precise than the one used by the LIP-based double-checking. One might suggest to tighten the threshold function of the LIP-based double-checking in order to improve the detection accuracy. This sounds reasonable, because tightening the threshold function increases the false positive rate, and the false positive rate of the LIP-based double-checking is lower than the false positive rate of the integration-based double-checking. Furthermore, the threshold function can initially be tightened with the parameter θ . However, results would hardly change. Indeed, at the beginning of the simulation, the LIP-based double-checking does many false positives, until η , the number of false positives enlarges enough the threshold function.

5.4.3 Overheads

Overheads in	memory (%)	computation (%)
Classic	+0.0	+0.0
LBDC	+57.6	+2.4
IBDC	+42.7	+4.5
Replication	+100	+100

Table 6. Benchmark of overheads between our double-checking based on Lagrange interpolation polynomials (LBDC) and on a numerical integration method (IBDC), replication, and the classic adaptive controller (Classic).

In Table 6, we compare the overheads in memory and in computation of the classic adaptive controller, our methods, and replication. When a corrupted step is detected by replication, the step is recomputed. The computational overhead of replication is exactly +100% plus the rate of corrected steps, but the rate of corrected steps is below 1%, and thus the overhead is equal to +100.

The computational overhead is partly due to the cost of the double-checking and to false positives, as false positives require to recompute a noncorrupted step. For the integration-based double-checking, the false positive rate is 4.2%, while the computational overhead is +4.5%. Therefore, the computational cost of our method is mainly due to the cost of recomputing a false positive. To a certain extent, the computational overhead can be reduced by decreasing the parameters γ and Γ , though this would also decrease the detection accuracy. The memory overhead can appear important, but is in average two times lower than the memory overhead of replication. It decreases with the complexity of the ODE method of the solver: in general, the solver requires $N_k + 2$ vectors of data with N_k the number of function evaluations, whereas the double-checking requires a fix number of vectors.

A variant of AID has been proposed by Subasi et al. [45] using support vector. It allows to reduce the memory overhead of AID. Such method could be employed also in the double-checkings to mitigate their memory overheads.

5.4.4 Scalability

Cores	512			4096		
	Class.	LBDC	IBDC	Class.	LBDC	IBDC
Double-check	-	$3.8e^2$	$3.9e^2$	-	$1.5e^1$	$1.6e^1$
Step	$1.2e^3$	$1.3e^3$	$1.3e^3$	$4.6e^2$	$4.8e^2$	$4.8e^2$

Table 7. Details of the mean execution time computation for the classic adaptive controller (Class.), LIP-based double-checking (LBDC), and integration-based double-checking (IBDC). Results are given in seconds.

Table 7 discloses the mean execution time computation for the double-checkings and the classic adaptive controller. The computational overheads remain below 5%. If the double-checkings suffer from a limited efficiency, it has the same efficiency than for the step. This is mainly due to the collective operation for computing the norms. Moreover, the table shows also that the double-checking is almost a pure additional cost to the classic adaptive controller. A better implementation must instead integrate better the double-checking inside the adaptive controller. This could be done by computing the norm of the error estimates used by the classic adaptive controller and the double-checkings at the same time. However, this requires also to allocate an additional vector, and then to increase the memory overheads.

Figure 11 show the relative performance in time (yellow) and memory (green) compared to the classical adaptive controller of the LIP-based double-checking (square) and the integration-based double-checking (circle) until 4096 cores. The overheads tend to decrease with the number of cores, because the SDC detectors provide a better scalability than the rest of HyPar. Indeed, with the number of cores decreasing, parts of HyPar that can not be parallelized becomes more and more important with respect to the cost of the double-checkings.

5.5 Conclusion

The section showed that cascading patterns are likely to corrupt an SDC detector. Consequences are worrying: a significant SDC can be accepted and the step

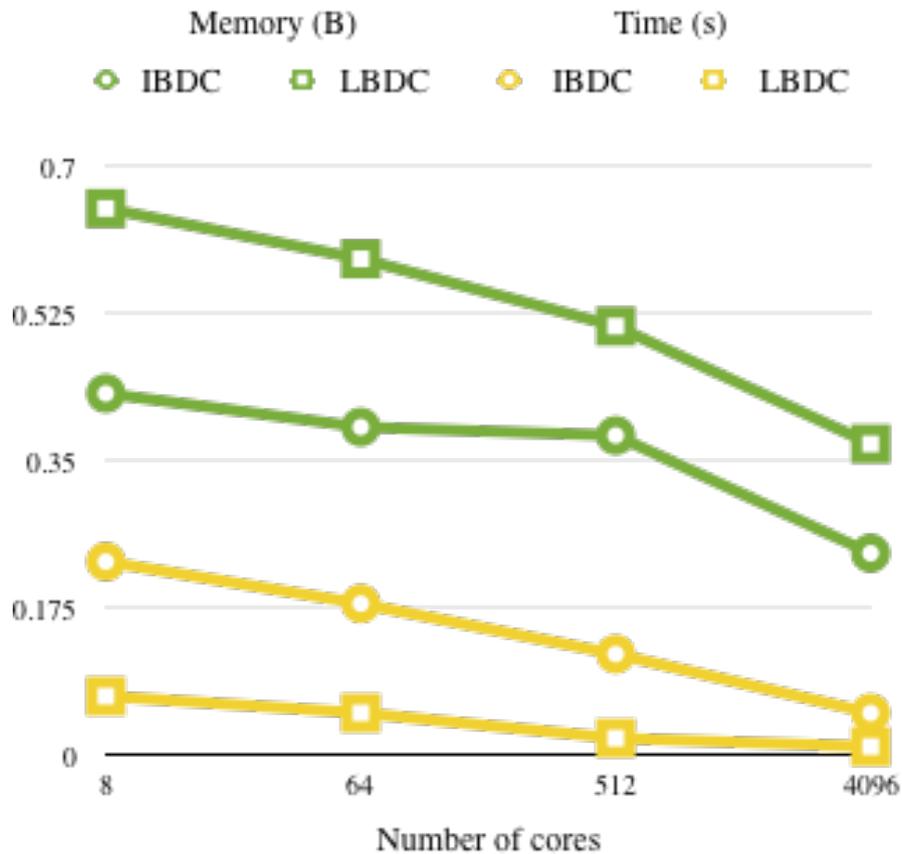


Figure 11. Relative performance in time and in memory of LIP-based double-checking (LPDC) and integration-based double-checking (IBDC) compared to the classic adaptive controller until 4096 cores.

size might be increased where it should be decreased. Proposed solution is based on redundancy: because the SDC detector is lightweight, a double-checking is achievable at a low computational cost.

We showed that the double-checking estimate and the estimate from the classic adaptive controller should agree. But dynamics of adaptive solvers are complex: the double-checking estimate must keep pace with the dynamics controlled by the first estimate. It appeared that extrapolation is not fitted for SDC detection on adaptive solvers: its evaluation is over-evaluated, and it does many false positives (44.8%). Using backward differentiation formula appears to be efficient when the order of the estimate is controlled to provide a trade-off between the number of false positives and the number of true positives.

6 Conclusion

In this report, we saw that numerical integration solvers are sensitive to corruptions. Improving their resilience is a requirement for the exascale computing and next generation of supercomputers. We made a distinction between solvers with a fixed integration step size and those with a variable integration step size. Replication is an efficient solution, but its memory and computational overheads can be prohibitive. Consequently, solutions with a lower cost but similar detection performance were presented.

For fixed solvers, users' accuracy expectation is implicit. Error estimations can approximate the expectation. Furthermore, checking that two different error estimates agree can detect all significant SDCs. We provided mathematical proofs and we performed experiments in a high-performance computing application.

For adaptive solvers, the expectation is given in the tolerances of the adaptive controller. If this controller can reject some SDCs, it is not reliable enough to reject all significant SDCs. We suggest to combine it with a double-check mechanism based on a second estimate. Experiments were performed with PETSc, a scalable toolkit for differential equations. It shows that the ratio of non-detected significant SDCs is reduced by a factor of 10.

In both cases, I compared the methods with state-of-the-art SDC detectors. We showed that solutions based on extrapolation can not detect significant SDCs, because the accuracy of extrapolation is lower than the accuracy of the solver. My methods have performed similarly than replication, but with a computation cost around 20 times lower and a memory cost 2 times lower.

This work considered only the case of nonsystematic SDCs. Recomputing a step was enough for correcting an SDC. Future works will consider the case of systematic SDC. It can be achieved by correcting an SDC with another ODE method. In this case, the systematic SDC might not be triggered during the correction.

Moreover, proposed SDC detectors have a large memory overhead, although it is still lower than replication. Improvements should consider solutions to decrease it. Because of inherent approximation error of a solver, storing the entire solutions are not required. Instead, compression methods or dimensionality reduction methods may be employed.

7 References

References

- [1] S. Ashby, P. Beckman, J. Chen, P. Colella, B. Collins, D. Crawford, J. Dongarra, D. Kothe, R. Lusk, P. Messina, *et al.*, “The opportunities and challenges of exascale computing,” *Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee*, pp. 1–77, 2010.
- [2] S. E. Lapinsky and A. C. Easty, “Electromagnetic interference in critical care,” *Journal of critical care*, vol. 21, no. 3, pp. 267–270, 2006.
- [3] B. Panzer-Steindel, “Data integrity,” 2007.
- [4] “A conversation with jeff bonwick and bill moore,” 2007.
- [5] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder, “An analysis of data corruption in the storage stack,” *ACM Transactions on Storage (TOS)*, vol. 4, no. 3, p. 8, 2008.
- [6] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, *et al.*, “Addressing failures in exascale computing,” *IJHPCA*, 2014.
- [7] S. Di and F. Cappello, “Adaptive impact-driven detection of silent data corruption for hpc applications,” *IEEE Transactions on Parallel and Distributed Systems*, 2016.
- [8] A. R. Benson, S. Schmit, and R. Schreiber, “Silent error detection in numerical time-stepping schemes,” *International Journal of High Performance Computing Applications*, p. 1094342014532297, 2014.
- [9] S. Chen, G. Bronevetsky, M. Casas-Guix, and L. Peng, “Comprehensive algorithmic resilience for numeric applications,” Tech. Report LLNL-CONF-618412, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2013.
- [10] T. Peterka, R. Ross, B. Nouanesengsy, T.-Y. Lee, H.-W. Shen, W. Kendall, and J. Huang, “A study of parallel particle tracing for steady-state and time-varying flow fields,” in *IPDPS*, pp. 580–591, IEEE, 2011.
- [11] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, *LINPACK users’ guide*. Siam, 1979.
- [12] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*, vol. 1. MIT press, 1999.
- [13] “MPICH2.” www.mcs.anl.gov/mpich2.
- [14] M. Team, “Mvapich2 2.2 b user guide,” 2001.
- [15] E. Constantinescu, “Estimating global errors in time stepping,” *Numerische Mathematik, Submitted*, 2015.

- [16] R. D. Skeel, “Thirteen ways to estimate global error,” *Numerische Mathematik*, vol. 48, no. 1, pp. 1–20, 1986.
- [17] L. Stoller and D. Morrison, “A method for the numerical integration of ordinary differential equations,” *Mathematical Tables and Other Aids to Computation*, pp. 269–272, 1958.
- [18] F. Ceschino and J. Kuntzmann, *Numerical solution of initial value problems*. Prentice-Hall, 1966.
- [19] J. Butcher and P. Johnston, “Estimating local truncation errors for Runge-Kutta methods,” *Journal of Computational and Applied Mathematics*, vol. 45, no. 1, pp. 203–212, 1993.
- [20] J. C. Butcher, *The numerical analysis of ordinary differential equations: Runge-Kutta and general linear methods*. Wiley-Interscience, 1987.
- [21] HAIRER, E., NORSETT, S. P., WANNER, G., *Solving Ordinary ,Differential Equations I, Nonstiff problems/E. Hairer, S. P. Norsett, G. Wanner, Second Revised Edition with 135 Figures, Vol.: 1*. 2Ed. Springer-Verlag, 2000, 2000. Index.
- [22] O. Abraham and G. Bolarin, “On error estimation in runge-kutta methods,” *Leonardo Journal of Sciences*, vol. 18, pp. 1–10, 2011.
- [23] R. Guerraoui and A. Schiper, “Software-based replication for fault tolerance,” *Computer*, no. 4, pp. 68–74, 1997.
- [24] D. Fiala, “Detection and correction of silent data corruption for large-scale high-performance computing,” in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pp. 2069–2072, May 2011.
- [25] R. E. Lyons and W. Vanderkulk, “The use of triple-modular redundancy to improve computer reliability,” *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962.
- [26] S. Ghosh, S. Basu, and N. A. Touba, “Selecting error correcting codes to minimize power in memory checker circuits,” *Journal of Low Power Electronics*, pp. 63–72, 2005.
- [27] K.-H. Huang and J. Abraham, “Algorithm-based fault tolerance for matrix operations,” *Computers, IEEE Transactions on*, vol. 100, no. 6, pp. 518–528, 1984.
- [28] S. Pauli, P. Arbenz, and C. Schwab, “Intrinsic fault tolerance of multi-level monte carlo methods,” *Journal of Parallel and Distributed Computing*, vol. 84, pp. 24–36, 2015.
- [29] S. Di, E. Berrocal, and F. Cappello, “An efficient silent data corruption detection method with error-feedback control and even sampling for hpc applications,” in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pp. 271–280, IEEE, 2015.

- [30] J. C. Butcher, *Numerical Methods for Ordinary Differential Equations*. Wiley Online Library, 2005.
- [31] B. Zhou, X.-L. Yang, R. Liu, and W. Wei, “Image segmentation with partial differential equations,” *Information Technology Journal*, vol. 9, no. 5, pp. 1049–1052, 2010.
- [32] L. Bautista-Gomez and F. Cappello, “Detecting and correcting data corruption in stencil applications through multivariate interpolation,” in *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pp. 595–602, IEEE, 2015.
- [33] P.-L. Guhur, H. Zhang, T. Peterka, E. Constantinescu, and F. Cappello, “Lightweight and accurate silent data corruption detection in ordinary differential equation solvers,” No. ANL/MCS-P5582-0316, 2016.
- [34] K. Krishnamoorthy and T. Mathew, *Statistical tolerance regions: theory, applications, and computation*, vol. 744. John Wiley & Sons, 2009.
- [35] T. H. Gronwall, “Note on the derivatives with respect to a parameter of the solutions of a system of differential equations,” *Annals of Mathematics*, pp. 292–296, 1919.
- [36] H. Guo, W. He, T. Peterka, H.-W. Shen, S. M. Collis, and J. J. Helmus, “Finite-time lyapunov exponents and lagrangian coherent structures in uncertain unsteady flows,” *IEEE TVCG (Proc. PacificVis 16)*, vol. 22, 2016. to appear.
- [37] T. McLoughlin, R. S. Laramée, R. Peikert, F. H. Post, and M. Chen, “Over two decades of integration-based, geometric flow visualization,” in *Eurographics 2009 State of the Art Report*, (Munich, Germany), pp. 73–92, 2009.
- [38] F. X. Giraldo, J. F. Kelly, and E. Constantinescu, “Implicit-explicit formulations of a three-dimensional nonhydrostatic unified model of the atmosphere (NUMA),” *SIAM Journal on Scientific Computing*, vol. 35, no. 5, pp. B1162–B1194, 2013.
- [39] G.-S. Jiang and C.-W. Shu, “Efficient implementation of weighted ENO schemes,” *Journal of Computational Physics*, vol. 126, no. 1, pp. 202–228, 1996.
- [40] D. Ghosh and J. D. Baeder, “Compact reconstruction schemes with weighted ENO limiting for hyperbolic conservation laws,” *SIAM Journal on Scientific Computing*, vol. 34, no. 3, pp. A1678–A1706, 2012.
- [41] “HyPar,” 2015. <https://hypar.github.io>.
- [42] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, and H. Zhang, “PETSc Web page.” <http://www.mcs.anl.gov/petsc>, 2015.

- [43] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, and H. Zhang, “PETSc users manual,” Tech. Rep. ANL-95/11 - Revision 3.6, Argonne National Laboratory, 2015.
- [44] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, “Efficient management of parallelism in object oriented numerical software libraries,” in *Modern Software Tools in Scientific Computing* (E. Arge, A. M. Bruaset, and H. P. Langtangen, eds.), pp. 163–202, Birkhäuser Press, 1997.
- [45] O. Subasi, S. Di, L. Bautista-Gomez, P. Balaprakash, O. Unsal, J. Labarta, A. Cristal, and F. Cappello, “Spatial support vector regression to detect silent errors in the exascale era,” in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing CCGrid*, pp. 413–424, IEEE, 2016.

8 Appendix

I lead three papers on this work:

- *Lightweight and Accurate Silent Data Corruption Detection in Ordinary Differential Equation Solvers*. Guhur, P. L., Zhang, H., Peterka, T., Constantinescu, E., & Cappello, F. In Euro-Par 2016. mcs.anl.gov/papers/P5582-0316.pdf
- *Detection of Silent Data Corruption in Adaptive Numerical Integration Solvers*. Guhur, P.L., Constantinescu, E., Ghosh, D., Peterka, T., & Cappello, F.
- *Controlling lossy compression from error estimates in numerical integration solvers*. Guhur, P.L., Calhoun, J., Constantinescu, E., Peterka, T., & Cappello, F.

This work has also been presented during several talks:

- a plenary session at CoDA 2016 by Franck Cappello, entitled *Improving the Trust in Results of Numerical Simulations and Scientific Data Analytics*;
- an one-hour seminar entitled *Error Estimation for Fault Tolerance in Numerical Integration Solvers* at Argonne National Laboratory;
- the lightning talk at JLESC 5th Workshop, entitled *Detecting Silent Data Corruptions with Error Estimations in Numerical Integration Solvers*.



Mathematics and Computer Science Division

Argonne National Laboratory
9700 South Cass Avenue, Bldg. 240
Argonne, IL 60439

www.anl.gov



Argonne National Laboratory is a U.S. Department of Energy
laboratory managed by UChicago Argonne, LLC