# Scrutinizing Variables for Checkpoint Using Automatic Differentiation

Xin Huang
*Kobe University*
*xin.huang@a.riken.jp*

Wubiao Xu, Shiman Meng, Weiping Zhang, Xiang Fu
Nanchang Hangkong University
{*wxu, smeng, wzhang, fuxiang*}*@nchu.edu.cn*

Luanzheng Guo
*Pacific Northwest National Laboratory*
*lenny.guo@pnnl.gov*

Kento Sato
*R-CCS, RIKEN*
*kento.sato@riken.jp*

***Abstract*—We propose a systematic approach that leverages automatic differentiation (AD) to scrutinize every element within variables (e.g., arrays) necessary for checkpointing. This allows us to identify critical and uncritical elements and eliminate uncritical elements from checkpointing. Specifically, we inspect every single element within a variable necessary for checkpointing with an AD tool to determine whether the element has an impact on the application output (numerical results) or not. We validate our approach with all benchmarks from the NAS Parallel Benchmark (NPB) suite. We successfully visualize the distribution of critical and uncritical elements within a variable with respect to its binary impact (yes or no) on the application output. We find patterns and distributions of critical and uncritical elements quite interesting. We find that all elements that have no impact on the output are not engaged in computation; it is not the fact that those elements are involved in computation but have no impact on the output. Finally, the evaluation of NPB benchmarks shows that our approach saves storage for checkpointing by up to 19%.**

## I. Introduction

As High Performance Computing (HPC) systems continue to advance in both scale and complexity, failures are observed at a higher frequency at leadership HPC systems [9], [12], [14]–[19], [26], which affects system reliability significantly. On the other hand, HPC systems are constrained in terms of storage capacity. Although the storage capacity is growing rapidly, the datasets are also ever-growing in size and speed [49], [50]. For example, the digital twin of earth workflow running on Summit at Oak Ridge National Laboratory can generate over 500 TB of data every 15 minutes [40]. As a consequence, effective storage management continues to be a critical challenge.

In response, we investigate application-level C/R approaches [2], [20], [36] and aim to reduce the application states (i.e., variables) necessary for checkpointing. Checkpoint/Restart is an essential fault-tolerant approach that stores the running state of the programs periodically and restarts from the latest stored state, which advances system reliability upon failures. However, the storage consumption of C/R checkpoints can be very large without careful inspection. For example, system-level C/R methods (e.g., BLCR [20]) save all corresponding system state which leads to notable storage consumption. Moreover, at the application level, a variable like a high-dimensional tensor can incur substantial costs in terms of storage consumption. (e.g., the GPT-3 model which is a tensor that can take 700 GB [7] if fully checkpointed). However, based on our observation, not every element within the variable participates in computation or can impact the result even if involved in computation.

Our goal is to identify the critical/uncritical elements within the variables necessary for checkpointing. An uncritical element is defined as an element that has no impact on the output; and vice versa. In particular, we propose an effective approach that uses AD to scrutinize every element within a variable (e.g., arrays) and determine whether it is critical or uncritical. This allows uncritical elements to be eliminated from checkpointing for storage efficiency. Automatic differentiation (AD) is a technology that computes the derivative of the programs. We further visualize the distribution of the critical/uncritical elements within each variable for checkpointing. We also attempt to find the reason why they become critical/uncritical by delving into the source code and algorithm. We find that in some cases the critical/uncritical elements are determined by the algorithms. For example, some elements are declared and initialized but not engaged in computation because of the algorithm (e.g., sampling). In other cases, uncritical elements are caused by imperfect programming. For instance, extra space is allocated in the declaration but not engaged in computation. Finally, we evaluate our approach on the NPB [1] benchmark suite and the results show that by eliminating uncritical elements from checkpointing the storage for checkpointing is reduced by an average of 13% and up to 19%.

In this paper, our contributions are listed as follows:

- A novel method that can identify critical elements within variables necessary for checkpointing that have an impact on the execution output without checkpointing the entire variable.
- Visualization of the distribution of critical-uncritical elements within variables necessary for checkpointing and investigation into its relation to the source code and algorithm.
- Evaluation of the proposed method on the NAS Parallel Benchmarks (NPB) [1] benchmarks.

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00056

## II. Background

### A. Checkpoint/Restart

Checkpoint/Restart has become a representative of fault-tolerant measures to address system failures throughout the years. Checkpoint/Restart enables programs to store the running state to checkpoint files at a specific interval, and recover from the latest storage once a failure occurs. Due to the different resilience requirements of users, the frequency of checkpointing varies in different HPC applications.

However, as HPC systems grow in scale, size, and complexity, the occurrence of system failures is observed more frequently. To cope with this, the frequency of writing checkpoints has been increasingly higher, which makes the checkpointing overhead higher than ever. On the other hand, the amount of data and the number of variables for checkpointing are growing which can also have a significant impact on performance.

Various C/R libraries have been created for various purposes through the years [6], [22], [25], [34]–[36], [39], [42]. Michel et al. [41] presented a C/R package that supports automated generation of the store and restore for the checkpointed object type. Moody et al. [32] developed a scalable checkpoint/restart (SCR) library, a multi-level checkpoint system that can save checkpoints to the computing node's RAM, Flash, disk, and parallel file system. Hargrove et al. [20] created the Berkeley Lab Checkpoint/Restart (BLCR) library, which provides system-level checkpointing for the Linux kernel. Gholami et al. [13] combined XOR and partner checkpointing to develop a stable and efficient C/R approach. Vasavada et al. [46] proposed a page-based incremental checkpoint approach by which memory writes are tracked by trapping dirty pages to be saved. The Fault Tolerant Interface (FTI) [2] is a three-level checkpoint scheme with a topology-aware Reed-Solomon encoding integration. In contrast, in this work, we further delve into the variables necessary for checkpointing to identify critical and uncritical elements for checkpointing.

### B. Automatic differentiation

Automatic differentiation (AD) [38] is used for computing the derivative of a function. Automatic differentiation is widely employed in many fields [8], [11], [23], [28], [43], [44] , such as partial differential equation (PDE) solutions [4], stability analysis [27], uncertainty quantification (UQ) [48], silent data corruption (SDC) prediction [29]. Paszke et al. [37] provide a library that aims at rapid research on various machine-learning models, an automatic differentiation module of Pytorch. Minkov et al. [30] use an automatic differentiation library to optimize the ultrasmall cavity's quality factor and the dispersion of a photonic crystal waveguide. Krieken et al. [24] propose a stochastic automatic differentiation framework that minimizes the gradient estimates' variance. Virmaux et al. [47] provide an algorithm working with automatic differentiation, which benefits the computations for extending and improving estimation methods. Crooks [5] provides a study on the optimization of quantum circuits using automatic differentiation for the maximum cut problem.

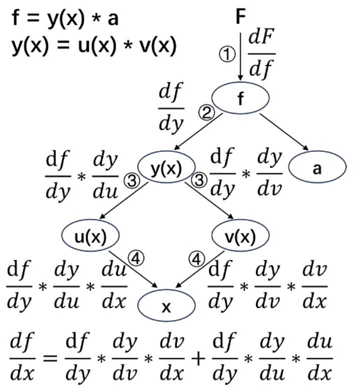

f = y(x) * a
y(x) = u(x) * v(x)
F
①
$\frac{dF}{df}$
f
$\frac{df}{dy}$
②
y(x)
a
$\frac{df}{dy} * \frac{dy}{du}$
③
③
$\frac{df}{dy} * \frac{dy}{dv}$
u(x)
v(x)
④
④
$\frac{df}{dy} * \frac{dy}{du} * \frac{du}{dx}$
$\frac{df}{dy} * \frac{dy}{dv} * \frac{dv}{dx}$
x
$\frac{df}{dx} = \frac{df}{dy} * \frac{dy}{dv} * \frac{dv}{dx} + \frac{df}{dy} * \frac{dy}{du} * \frac{du}{dx}$

Fig. 1: An example of AD workflow. $a$ is a constant.

AD considers a computer program as a function, which is a combination of a series of basic arithmetic operations that are regarded as primitive functions. AD computes the derivative of the primitive functions to compute the derivative of the output to the program. In this process the chain rule of differential calculus was used, so that the program can compute the derivative of the desired target. For example, a function looks like:

$$F = f(y) = f(u(x), v(x)) \tag{1}$$

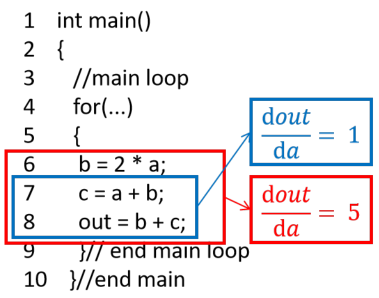
Assume that we want to compute the derivative $\frac{\mathrm{d}f}{\mathrm{d}x}$ . There are two main strategies of AD tools: forward mode and reverse mode. The forward mode computes $\frac{\mathrm{d}u}{\mathrm{d}x}$ and $\frac{\mathrm{d}v}{\mathrm{d}x}$ at first, then computes $\frac{\mathrm{d}y}{\mathrm{d}u}$ and $\frac{\mathrm{d}y}{\mathrm{d}v}$ , finally computes $\frac{\mathrm{d}f}{\mathrm{d}x}$ based on the chain rule. On the contrary, the reverse mode computes $\frac{\mathrm{d}f}{\mathrm{d}y}$ and then $\frac{\mathrm{d}y}{\mathrm{d}u}$ and $\frac{\mathrm{d}y}{\mathrm{d}v}$.

We give an example of the reverse mode of AD in Figure 1. In reverse mode, AD first sweeps through the forward execution and obtains the information about the program such as variables, branches, and iterations. With the information obtained, AD computes the partial derivative of each operation and calculates the final derivative by the chain rule.

There are different AD tools like ADIC [3], Tapenade [21], OpenAD [45], Enzyme [33], etc. Enzyme [33] is an LLVM compiler-based AD tool that performs reverse mode. Enzyme is a state-of-the-art AD tool that supports various optimizations on different hardware platforms and programming models. We thus use it and leverage AD to pinpoint critical and uncritical elements within the variables necessary for checkpointing.

## III. Approach

Our goal is to identify uncritical elements within selected variables necessary for checkpointing that have no impact on the output and critical elements that impact the output. *A variable is defined as a memory location that is paired with an associated symbolic name. This symbolic name is then referenced in the source code and invoked during execution.* For example, a float-point type array $arr$ declared by $double\ arr[5]$ is a variable and $arr[0]$ is an element. An uncritical element is defined as an element that has zero impact on the output; and vice versa.



```
1  int main()
2  {
3    //main loop
4    for(...)
5    {
6      b = 2 * a;
7      c = a + b;
8      out = b + c;
9    }// end main loop
10 }//end main
```

Fig. 2: Derivatives with different encapsulations

**Specifically, we want to pinpoint, given an element $x$, whether it has an impact on the output or not. Leveraging AD, we can calculate the derivative of the output with respect to the element $x$. The binary impact of each element is determined by its derivative with respect to the output. If the derivative is zero, we believe it has no impact; otherwise, it has an impact.**

The main computational loop *(the outermost loop within the main function)* may be encapsulated within a specific function or it may be an independent loop structure code block within the main function. To enable AD to analyze the impact of each element within a variable on the output, it is essential to 1) identify the code region that includes the variable as input, the computation the variable is involved, and the output; 2) adjust the code region (required by Enzyme), encapsulated by a function, where the variable in question is made as input parameter; the output is returned. This step is crucial because the results of AD can vary using distinguished encapsulations. When the analysis scope defined for AD changes, the derivative can be different. Figure 2 illustrates an example of distinct encapsulations. When the encapsulation is specified in lines 7-8 (blue box), the derivative of variable $out$ with respect to $a$ is 1. When the encapsulation is specified in lines 6-8 (red box), the derivative $\frac{\mathrm{dout}}{\mathrm{da}} = \frac{\mathrm{dc}}{\mathrm{da}} + \frac{\mathrm{db}}{\mathrm{da}} = 5$. We assume that no failure occurs during AD analysis, the result of AD can guide future C/R practice. For each different input, we need to perform AD again.

## IV. Evaluation

We evaluate our approach with the NPB [1] benchmarks (the C version by Seoul National University). *First*, we determine the variables necessary for checkpointing in each benchmark. *Second*, the proposed AD approach is used to identify critical/uncritical elements within checkpointing variables. *Third*, we visualize the distributions of the critical/uncritical elements within each variable necessary for checkpointing. *Finally*, we verify the AD results.

### A. Variables necessary for checkpointing

All the variables necessary for checkpointing in the benchmarks are determined using AutoCheck [10], an automatic tool to identify variables for checkpointing, following principles defined by the recent work of Fu et al. [9]. They are listed in

TABLE I: Identified variables necessary for checkpointing. All the variable sizes are determined by the input class of $S$ which is easier to visualize.

| Benchmark | Variables and their data types |
|---|---|
| BT | double $u[12][13][13][5]$, int $step$ |
| SP | double $u[12][13][13][5]$, int $step$ |
| MG | double $u[46480]$, double $r[46480]$, int $it$ |
| CG | double $x[1402]$, int $it$ |
| LU | double $u[12][13][13][5]$, double $rho\_i[12][13][13]$, double $qs[12][13][13]$, double $rsd[12][13][13][5]$, int $istep$ |
| FT | dcomplex $y[64][64][65]$, dcomplex $sums[6]$, int $kt$ |
| EP | double $sx$, double $sy$, double $q[10]$, int $k$ |
| IS | int $passed\_verification$, int $key\_array[65536]$, int $bucket\_ptrs[512]$, int $iteration$ |

Table I. We consider all of them in our study. The variables are described as follows:

BT and SP: $u$ is the solution to the nonlinear partial differential equations (PDEs); $step$ is the main loop index.

MG: $u$ is the solution to the three-dimensional discrete Poisson equation; $r$ is the residual of the equation; $it$ is the main loop index.

CG: $x$ is the input vector of the linear system of equations; $it$ is the main loop index.

LU: $u$ is the solution to the nonlinear partial differential equations (PDEs); $rho\_i$ is the relaxation factor in the Symmetric Successive Over-Relaxation method; $qs$ is a variable for computing the flux differences; $rsd$ is a variable for computing the final residual; $istep$ is the main loop index.

FT: $y$ is the output signal of Fast Fourier Transform at frequency domain; $sums$ aggregates the sums computed from all iterations; $y$ and $sums$ are of custom data type $dcomplex$, containing two attributes, $real$ of `double` and $imag$ of `double`. $kt$ is the main loop index.

EP: $sx$ and $sy$ are the sums of independent Gaussian deviates at the $X$ and $Y$ dimensions respectively; $q$ is the number of pairs of coordinates at $X$ and $Y$; $k$ is the main loop index.

IS: $passed\_verification$ is the verification counter; $key\_array$ is the array that stores the keys of bucket sort; $bucket\_ptrs$ is the pointer of the 'bucket' in bucket sort; $iteration$ is the main loop index.

### B. Automatic differentiation analysis

**BT [1]**: BT is one of the NPB benchmarks that implements a Block Tri-diagonal solver to tackle three sets of equations. For BT, there are two variables necessary for checkpointing, a four-dimensional array $u$ and an integer $step$. $step$ is the index of the main loop. $step$ is a scalar that has an impact on the output as it is necessary for checkpointing. Its impact is obvious as the index variable of a for-loop. $u$ contains 10,140 elements with the input class of $S$. We calculate the derivative at each element of $u$ by AD to pinpoint their impact on the output. There are 8640 critical elements (there is impact) and 1500 uncritical elements (no impact). The uncritical elements are accounted for 14.8% of the total elements within $u$.

To further understand the distribution of critical and uncritical elements, we aim to visualize the distribution of critical-uncritical elements. This is very challenging as $u$ is a $12 \times 13 \times 13 \times 5$ four-dimensional array. Fortunately, we find

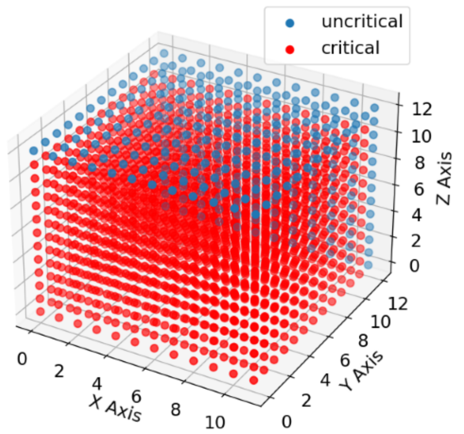


Fig. 3: A typical critical-uncritical distribution in NPB benchmarks(red: critical, blue: uncritical). Variables following this distribution: $u(BT)$, $u(SP)$, $u[x][y][z][0](LU)$, $u[x][y][z][1](LU)$, $u[x][y][z][2](LU)$, $u[x][y][z][3](LU)$, $rho\_i(LU)$, $qs(LU)$, $rsd(LU)$

```
1  for (k = 0; k <= grid_points[2]-1; k++) {
2   zeta = (double)(k) * dnzm1;
3    for (j = 0; j <= grid_points[1]-1; j++) {
4      eta = (double)(j) * dnym1;
5      for (i = 0; i <= grid_points[0]-1; i++) {
6        xi = (double)(i) * dnxm1;
7        exact_solution(xi, eta, zeta, u_exact);
8
9       for (m = 0; m < 5; m++) {
10          add = u[k][j][i][m]-u_exact[m];
11          rms[m] = rms[m] + add*add;
12        }
13      }
14    }
15  }
```

Fig. 4: A code snippet of the function $error\_norm$ in $BT$

that $u$ can be decomposed into five three-dimensional arrays of $12 \times 13 \times 13$. We find that all five three-dimensional arrays share the same critical-uncritical distribution pattern. We show one of the three-dimensional arrays in Figure 3.

The critical-uncritical distribution is quite interesting, in which uncritical elements are represented in blue and critical elements are represented in red; the uncritical elements are distributed on the two surfaces of the cube at $y = 12$ and $z = 12$; the remaining elements are critical.

We attempt to understand the underlying logic of the distribution by making connections to the source code and algorithm. After digging into the source code, we find that the $error\_norm$ function is the one that creates the specific critical-uncritical distribution in $u$. The $error\_norm$ function can be found at Line 41 in the `error.c` file of the BT source code. We provide the part of the source code of $error\_norm$ that operates on $u$ in Figure 4, where $u$ is used at Line 10. We observe that the range variables, $grid\_points[0]$ (see Line 5), $grid\_points[1]$ (see Line 3), and $grid\_points[2]$ (see Line 1), all have a value of 12. Given fixed $m$, recalling that $u$ is $u[12][13][13][5]$ while the access range is from zero to 11 for $k$, $j$, and $i$, we observe that the elements are not used at $j = 12$ and $i = 12$, mapping to axes Y and Z in Figure 3. Therefore, the elements at $y = 12$ and $z = 12$ are uncritical because they did not participate in computation. This is an interesting finding when uncritical elements are caused by imperfect coding.

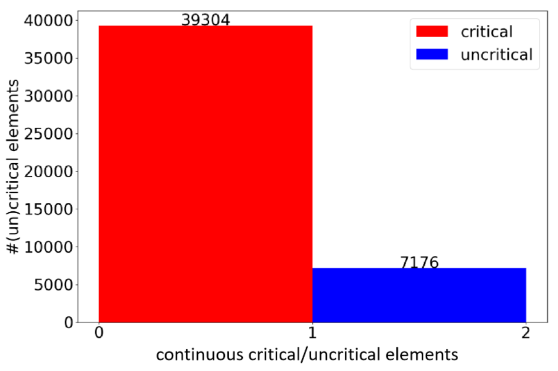


Fig. 5: Critical-uncritical distribution of array $u$ in MG. The red bar represents 39304 continuous critical elements in the data structure, followed by the 7176 continuous uncritical elements.

**SP** [1]: $SP$ is a Scalar Pentadiagonal solver similar to $BT$. The analysis process is similar to $BT$ because of the similar code structure to $BT$. The two variables necessary for checkpointing in $SP$ are the same as $BT$, a four-dimensional array u, and an integer $step$. We find the exactly same critical-uncritical distribution in $u$ as we found in $u$ in BT. Again, the $error\_norm$ function operates on $u$ and creates the particular critical-uncritical distribution in $u$. $SP$ invokes the same function $error\_norm$ at Line 41 in `error.c`, which is the exactly same as is invoked in $BT$. $step$ is the index of the main loop that is needed for checkpointing.

**MG** [1]: The MultiGrid algorithm employs the V-cycle multigrid method to efficiently solve a three-dimensional discrete Poisson equation. Our method is evaluated on $MG$ of NPB benchmarks with the input class of $S$. The variables necessary for checkpointing in this program are integer $it$, array $u$ of 46480 elements, and array $r$ of 46480 elements. $it$, the index of the main loop, is critical for checkpointing.

We find 7176 uncritical elements in $u$ and 10543 uncritical elements in $r$, respectively, accounting for 15.3% and 22.4% of all the elements. We find that the variable $u$ is transformed into a $34 \times 34 \times 34$ three-dimensional array in $MG$ to be used in the computation. In effect, there are only $34 \times 34 \times 34$ elements of $u$ participating in the computation.

We visualize the critical-uncritical distribution of $u$ in Figure 5, which shows that there are 39304 ($34 \times 34 \times 34$) continuous critical elements, followed by 7176 continuous uncritical ones within array $u$. However, $r$ has a more complex pattern of critical-uncritical distribution as the invoked elements are triggered by an integer array $ir$. Figure 6 shows a repetitive pattern **as part of** the critical-uncritical distribution of $r$.

**CG** [1]: The Conjugate Gradient method utilizes the conjugate gradient and inverse iteration methods as a solution

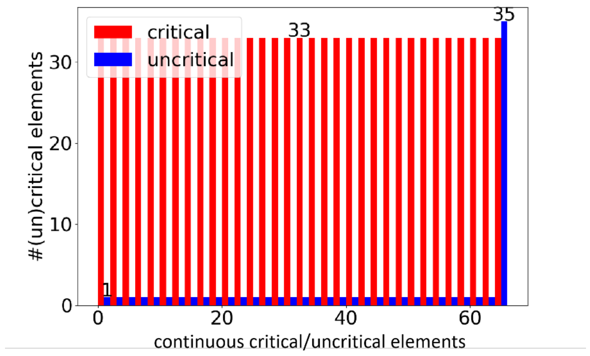


Fig. 6: Critical-uncritical distribution of of array $r$ in MG. The bars represent a certain pattern followed by a blue bar of 35 uncritical elements.

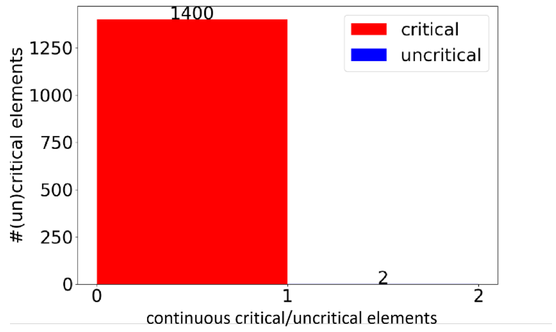


Fig. 7: Critical-uncritical distribution of array $x$ in CG. The red bar represents 1400 continuous critical elements in the data structure, followed by the 2 continuous uncritical elements.

to linear equations. There are two variables necessary for checkpointing in CG: an integer $it$ that is the index of the main loop, and an array $x$ of length 1402. Figure 7 shows the distribution of critical-uncritical regions of $x$ in CG. The first 1400 continuous elements are critical, and the remaining 2 elements are uncritical. $x$ is first read as an input and then updated (written) within the main loop. After delving into the source code, we find that $x$ has a size of $NA + 2$, in which $NA$ is a macro that has the value of 1400 for the $S$ class, and only the first $NA$ elements participate in computation. $it$ is required for checkpointing.

**LU [1]**: LU is the Lower-Upper Symmetric Gauss-Seidel solver, which is a numerical method to solve linear systems of equations. There are five array variables necessary for checkpointing in LU, which are $u$, $rho\_i$, $qs$, $rsd$, and $istep$. $istep$ is the index of the main loop necessary for checkpointing.

$u$: There are 10140 elements in array $u$, in which 1628 elements are uncritical, accounting for 16% of all elements. The variable $u$ here is slightly different from $u$ in $BT$ and $SP$, although they are in the same data type and size (i.e., double $u[12][13][13][5]$). There are five $12 \times 13 \times 13$ 3D arrays in $u$. We find that each of the first four $12 \times 13 \times 13$ 3D arrays follows the critical-uncritical distribution in Figure 3

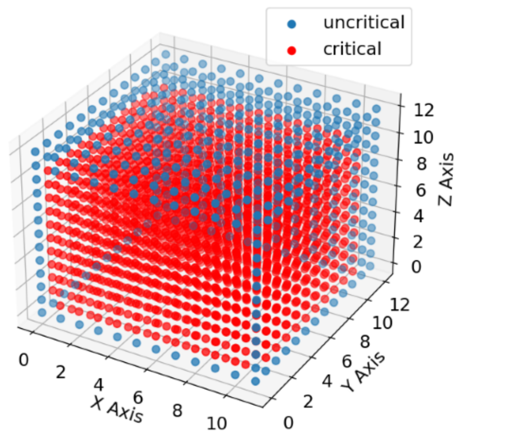


Fig. 8: Critical-uncritical distribution of $u[x][y][z][4]$ in $LU$ (red: critical, blue: uncritical)

exactly. However, the fifth, $u[x][y][z][4]$, follows a different critical-uncritical distribution shown in Figure 8.

After delving into the source code, we find that $u[x][y][z][4]$ participates in multiple discontinuous computations unlike $u[x][y][z][0-3]$ that is utilized in separate computations akin to Figure 4. We find that the components of $u[x][y][z][4]$ used in multiple discontinuous computations are $u[1-10][1-10][0-11][4]$, $u[1-10][0-11][1-10][4]$, and $u[0-11][1-10][1-10][4]$, which constitute the critical (red) area in Figure 8. We also find that there are 128 more uncritical elements (on the edges) not participating in computation compared with the critical-uncritical distribution in Figure 3.

$rho\_i$ **and** $qs$: We find that there are 300 uncritical out of 2028 elements in $rho\_i$. Particularly, there are $12 \times 12 \times 12$ out of $12 \times 13 \times 13$ elements participating in computation, which leads to the same critical-uncritical distribution in Figure 3. It is the same case for $qs$.

$rsd$: $rsd$ is the same as $u$ in BT: they are in the same size and participate in the same computation that results in the same critical-uncritical distribution (Figure 3).

**FT [1]**: FT executes a 3D Fast Fourier Transform on a grid of three-dimensional points. There are three variables necessary for checkpointing in $FT$: a $64 \times 64 \times 65$ three-dimensional array $y$, containing 266,240 elements, and each element is a custom data structure $dcomplex$ with two attributes $imag$ and $real$ of floating point type; a $dcomplex$ array $sums$ of length 6; and an integer $kt$. $kt$ is the index of the main loop required for checkpointing.

$y$: we find 4096 uncritical of all 266240 elements. Figure 9 shows the critical-uncritical distribution, in which only the top layer (in blue) at $k = 64$ does not participate (not used) in computation. This is due to imperfect coding, which can be avoided not only for code safety but also for efficient usage of memory and storage and also for checkpointing efficiency.

$sums$: $sums$ stores the result computed at each iteration of the main loop. Therefore, it is critical to write it to storage to avoid loss of computed results upon a failure.

**EP [1]**: The embarrassingly parallel (EP) generates a pair

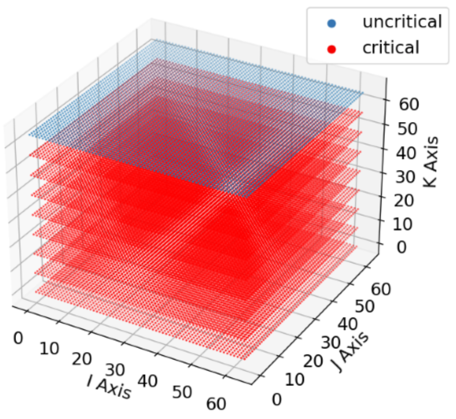


Fig. 9: Critical-uncritical distribution of $y$ in $FT$

of random numbers that follow a normal distribution. The variables necessary for checkpointing are floating-point $sx$ and $sy$, array $q$, and integer $k$. Both $sx$ and $sy$ are write-after-read which is necessary for checkpointing. $k$ is the index of the main loop that is required for checkpointing. $q$ stores the output at each iteration of the main loop, so we must write it to storage to avoid recomputation upon a failure.

**IS** [1]: $IS$ is a bucket sorting method specifically designed for sorting small integers. The variables necessary for checkpointing in $IS$ are integer $passed\_verification$, integer $iteration$, integer arrays $key\_array$, and $bucket\_ptrs$.

Again, $passed\_verification$ is write-after-read, which is necessary for checkpointing. $iteration$ is the index of the main loop that is a critical variable for checkpointing. Like $iteration$, $key\_array$ and $bucket\_ptrs$ store the indexes for other arrays which makes them critical for checkpointing.

| Benchmark(variable) | Uncritical | Total | Uncritical rate |
|---|---|---|---|
| BT(u) | 1500 | 10140 | 14.8% |
| SP(u) | 1500 | 10140 | 14.8% |
| MG(u) | 7176 | 46480 | 15.4% |
| MG(r) | 10543 | 46480 | 22.7% |
| CG(x) | 2 | 1402 | 0.1% |
| LU(qs) | 300 | 2028 | 14.8% |
| LU(rsd) | 300 | 2028 | 14.8% |
| LU(rho_i) | 1500 | 10140 | 14.8% |
| LU(u) | 1628 | 10140 | 16.0% |
| FT(y) | 4096 | 266240 | 1.5% |

TABLE II: Number of uncritical elements

We summarize the number of uncritical elements along with the respective percentage they represent in relation to the total elements for all variables necessary for checkpointing in Table II.

### C. Verifying AD results

To verify that the uncritical elements detected by AD are not needed and the critical elements are critical for checkpointing, we implement a homemade checkpointing library that writes only critical elements to checkpoint files. In contrast to comprehensive C/R libraries, such as SCR [31], our homemade C/R library only supports C/R by writing checkpoints to a local file and restarting the execution from the latest checkpoint file. This C/R library is basic but sufficient for verifying the identified variables' correctness using our methodology. All NPB benchmarks have their own verification phase, which uses a margin of error to determine if the computation is successful or failed. We rely on their verification to determine the AD result's correctness. In principle, the uncritical elements should not impact the computation correctness even if their values are altered by system failures. On the other hand, the critical elements must impact the execution output, and the verification is expected to fail if they cannot recover from failures. It turned out that, all benchmarks restarted successfully and passed the verification upon only checkpointing the critical elements. This demonstrates the effectiveness of the AD analysis for scrutinizing variables for checkpointing.

| Benchmark | Original | Optimized | Storage saved |
|---|---|---|---|
| BT | 79.4kb | 67.7kb | 14.8% |
| SP | 79.4kb | 67.7kb | 14.8% |
| MG | 727kb | 588kb | 19.1% |
| CG | 10.9kb | 10.9kb | 0.1% |
| LU | 191kb | 161kb | 15.7% |
| FT | 4161kb | 4097kb | 1% |

TABLE III: Storage consumption for checkpointing

### D. Storage for checkpointing

We show the comparison of checkpointing storage before and after eliminating uncritical elements in Table III. As it shows, the storage saved is consistent with the uncritical rate in Table II.

## V. Discussion

The uncritical elements for checkpointing we find in the NAS Parallel Benchmark suite are not involved in the computation, which is mostly caused by programming defects (i.e., extra space is allocated in the declaration but not engaged in computation). Those programming defects can be avoided not only for high-quality code and code safety but for efficient usage of memory and storage and also for high-performance checkpointing.

## VI. Conclusion

We propose a systematic approach that utilizes automatic differentiation (AD) to meticulously examine each element within variables, such as arrays, for checkpointing. This enables us to distinguish between critical and uncritical elements and exclude the latter from the checkpointing process. To be more specific, our method involves a thorough inspection of each individual element within a variable using an AD tool to determine its impact on the application output. We validate our approach using all benchmarks from the NAS Parallel Benchmark (NPB) suite. The patterns and distributions of critical and uncritical elements pique our interest. We find that uncritical elements are not engaged in computation at all. It is not the case that they participate in computation but have no impact on the output. Finally, the evaluation of our approach on NPB benchmarks reveals reduction in storage consumption for checkpointing. For future work, we aim to apply the methodology to real-world scientific or computational problems to assess its practical utility and impact.

## References

[1] Bailey, D.H., Barszcz, E., Dagum, L., Simon, H.D.: Nas parallel benchmark results. IEEE Parallel & Distributed Technology: Systems & Applications **1**(1), 43–51 (1993)

[2] Bautista-Gomez, L.A., Tsuboi, S., Komatitsch, D., Cappello, F., Maruyama, N., Matsuoka, S.: FTI: high performance fault tolerance interface for hybrid systems. In: Conference on High Performance Computing Networking, Storage and Analysis (SC) (2011)

[3] Bischof, C.H., Roh, L., Mauer-Oats, A.J.: Adic: an extensible automatic differentiation tool for ansi-c. Software: Practice and Experience **27**(12), 1427–1456 (1997)

[4] Borggaard, J., Verma, A.: On efficient solutions to the continuous sensitivity equation using automatic differentiation. SIAM Journal on Scientific Computing **22**(1), 39–62 (2000)

[5] Crooks, G.E.: Performance of the quantum approximate optimization algorithm on the maximum cut problem. arXiv preprint arXiv:1811.08419 (2018)

[6] Di, S., Bouguerra, M.S., Bautista-Gomez, L., Cappello, F.: Optimization of multi-level checkpoint model for large scale hpc applications. In: 2014 IEEE 28th international parallel and distributed processing symposium. pp. 1181–1190. IEEE (2014)

[7] Floridi, L., Chiriatti, M.: Gpt-3: Its nature, scope, limits, and consequences. Minds and Machines **30**, 681–694 (2020)

[8] Fournier, D.A., Skaug, H.J., Ancheta, J., Ianelli, J., Magnusson, A., Maunder, M.N., Nielsen, A., Sibert, J.: Ad model builder: using automatic differentiation for statistical inference of highly parameterized complex nonlinear models. Optimization Methods and Software **27**(2), 233–249 (2012)

[9] Fu, X., Huang, X., Xu, W., Zhang, W., Meng, S., Guo, L., Sato, K.: Benchmarking variables for checkpointing in hpc applications. In: 2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 406–413. IEEE (2024)

[10] Fu, X., Zhang, W., Huang, X., Meng, S., Xu, W., Guo, L., Sato, K.: Autocheck: Automatically identifying variables for checkpointing by data dependency analysis. Conference on High Performance Computing Networking, Storage and Analysis (SC) (2024)

[11] Fushimi, T., Yamamoto, K., Ochiai, Y.: Acoustic hologram optimisation using automatic differentiation. Scientific reports **11**(1), 12678 (2021)

[12] Georgakoudis, G., Guo, L., Laguna, I.: Reinit: Evaluating the performance of global-restart recovery methods for mpi fault tolerance. In: International Conference on High Performance Computing. pp. 536–554. Springer (2020)

[13] Gholami, M., Schintke, F.: Combining xor and partner checkpointing for resilient multilevel checkpoint/restart. In: 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 277–288. IEEE (2021)

[14] Guo, L., Georgakoudis, G., Parasyris, K., Laguna, I., Li, D.: Match: An mpi fault tolerance benchmark suite. In: 2020 IEEE International Symposium on Workload Characterization (IISWC). pp. 60–71. IEEE (2020)

[15] Guo, L., Li, D.: Moard: Modeling application resilience to transient faults on data objects. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 878–889. IEEE (2019)

[16] Guo, L., Li, D., Laguna, I.: Paris: Predicting application resilience using machine learning. Journal of Parallel and Distributed Computing **152**, 111–124 (2021)

[17] Guo, L., Li, D., Laguna, I., Schulz, M.: Fliptracker: Understanding natural error resilience in hpc applications. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 94–107 (2018). https://doi.org/10.1109/SC.2018.00011

[18] Guo, L., Li, D., Laguna, I., Schulz, M.: Fliptracker: Understanding natural error resilience in hpc applications. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 94–107. IEEE (2018)

[19] Guo, L., Lofstead, J., Ren, J., Laguna, I., Kestor, G., Pouchard, L., Oryspayev, D., Jeon, H.: Understanding system resilience for converged computing of cloud, edge, and hpc. In: International Conference on High Performance Computing. pp. 221–233. Springer (2023)

[20] Hargrove, P.H., Duell, J.C.: Berkeley lab checkpoint/restart (blcr) for linux clusters. In: Journal of Physics: Conference Series. vol. 46, p. 494. IOP Publishing (2006)

[21] Hascoet, L., Pascual, V.: The tapenade automatic differentiation tool: principles, model, and specification. ACM Transactions on Mathematical Software (TOMS) **39**(3), 1–43 (2013)

[22] Kale, L.V., Krishnan, S.: Charm++ a portable concurrent object oriented system based on c++. In: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications. pp. 91–108 (1993)

[23] Kaminski, T., Giering, R., Scholze, M., Rayner, P., Knorr, W.: An example of an automatic differentiation-based modelling system. In: International Conference on Computational Science and Its Applications. pp. 95–104. Springer (2003)

[24] Krieken, E., Tomczak, J., Ten Teije, A.: Storchastic: A framework for general stochastic automatic differentiation. Advances in Neural Information Processing Systems **34**, 7574–7587 (2021)

[25] Laadan, O., Nieh, J.: Transparent checkpoint-restart of multiple processes on commodity operating systems. In: USENIX Annual Technical Conference. pp. 323–336 (2007)

[26] Li, Z., Menon, H., Mohror, K., Liu, S., Guo, L., Bremer, P.T., Pascucci, V.: A visual comparison of silent error propagation. IEEE Transactions on Visualization and Computer Graphics (2022)

[27] Mader, C.A., Martins, J.R.: Computation of aircraft stability derivatives using an automatic differentiation adjoint approach. AIAA journal **49**(12), 2737–2750 (2011)

[28] Mazza, D., Pagani, M.: Automatic differentiation in pcf. Proceedings of the ACM on Programming Languages **5**(POPL), 1–27 (2021)

[29] Menon, H., Mohror, K.: Discvar: Discovering critical variables using algorithmic differentiation for transient faults. ACM SIGPLAN Notices **53**(1), 195–206 (2018)

[30] Minkov, M., Williamson, I.A., Andreani, L.C., Gerace, D., Lou, B., Song, A.Y., Hughes, T.W., Fan, S.: Inverse design of photonic crystals through automatic differentiation. Acs Photonics **7**(7), 1729–1741 (2020)

[31] Moody, A., Bronevetsky, G., Mohror, K., De Supinski, B.R.: Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–11. IEEE (2010)

[32] Moody, A., Fernandez, J., Petrini, F., Panda, D.: Scalable NIC-based Reduction on Large-Scale Clusters. In: SC '03 (November 2003)

[33] Moses, W.S., Churavy, V., Paehler, L., Hückelheim, J., Narayanan, S.H.K., Schanen, M., Doerfert, J.: Reverse-mode automatic differentiation and optimization of gpu kernels via enzyme. In: Proceedings of the international conference for high performance computing, networking, storage and analysis. pp. 1–16 (2021)

[34] Ni, X., Meneses, E., Jain, N., Kalé, L.V.: Acr: Automatic checkpoint/restart for soft and hard error protection. In: Proceedings of the international conference on high performance computing, networking, storage and analysis. pp. 1–12 (2013)

[35] Nicolae, B., Cappello, F.: Blobcr: Efficient checkpoint-restart for hpc applications on iaas clouds using virtual disk image snapshots. In: SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–12. IEEE (2011)

[36] Nicolae, B., Moody, A., Gonsiorowski, E., Mohror, K., Cappello, F.: Veloc: Towards high performance adaptive asynchronous checkpointing at large scale. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 911–920. IEEE (2019)

[37] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A.: Automatic differentiation in pytorch (2017)

[38] Rall, L.B., Corliss, G.F.: An introduction to automatic differentiation. Computational Differentiation: Techniques, Applications, and Tools **89**, 1–18 (1996)

[39] Rodd, R.L.: Doe data days 2022 report . https://doi.org/10.2172/1889523, https://www.osti.gov/biblio/1889523

[40] Rodd, R.L.: Doe data days 2022 report. Tech. rep., Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States) (2022)

[41] Schanen, M., Narayanan, S.H.K., Williamson, S., Churavy, V., Moses, W.S., Paehler, L.: Transparent checkpointing for automatic differentiation of program loops through expression transformations. In: International Conference on Computational Science. pp. 483–497. Springer (2023)

[42] Shahzad, F., Thies, J., Kreutzer, M., Zeiser, T., Hager, G., Wellein, G.: Craft: A library for easier application-level checkpoint/restart and automatic fault tolerance. IEEE Transactions on Parallel and Distributed Systems **30**(3), 501–514 (2018)

[43] Su, J., Renaud, J.E.: Automatic differentiation in robust optimization. AIAA journal **35**(6), 1072–1079 (1997)
[44] Townsend, J., Koep, N., Weichwald, S.: Pymanopt: A python toolbox for optimization on manifolds using automatic differentiation. arXiv preprint arXiv:1603.03236 (2016)
[45] Utke, J., Naumann, U., Fagan, M., Tallent, N., Strout, M., Heimbach, P., Hill, C., Wunsch, C.: Openad/f: A modular open-source tool for automatic differentiation of fortran codes. ACM Transactions on Mathematical Software (TOMS) **34**(4), 1–36 (2008)
[46] Vasavada, M., et al.: Innovative schemes to suppport incremental checkpointing. (2010)
[47] Virmaux, A., Scaman, K.: Lipschitz regularity of deep neural networks: analysis and efficient estimation. Advances in Neural Information Processing Systems **31** (2018)
[48] Wang, M., Lin, G., Pothen, A.: Using automatic differentiation for compressive sensing in uncertainty quantification. Optimization Methods and Software **33**(4-6), 799–812 (2018)
[49] Wang, Y., Shi, W., Berrocal, E.: On performance resilient scheduling for scientific workflows in hpc systems with constrained storage resources. In: Proceedings of the 6th Workshop on Scientific Cloud Computing. pp. 17–24 (2015)
[50] Zhang, J., Zhuo, X., Moon, A., Liu, H., Son, S.W.: Efficient encoding and reconstruction of hpc datasets for checkpoint/restart. In: 2019 35th Symposium on Mass Storage Systems and Technologies (MSST). pp. 79–91. IEEE (2019)