# Transparent Checkpointing for Automatic Differentiation of Program Loops through Expression Transformations

Michel Schanen[1][0000−0002−4164−027X], Sri Hari Krishna Narayanan[1][0000−0003−0388−5943], Sarah Williamson[2][0000−0002−0583−1546], Valentin Churavy[3][0000−0002−9033−165X], William S. Moses[3][0000−0003−2627−0642], and Ludger Paehler[3][0000−0002−7200−7637]

[1] Argonne National Laboratory, Lemont, IL 60439, USA
{mschanen,snarayan}@anl.gov
[2] Oden Institute for Computational Engineering and Sciences, University of Texas at Austin, Austin, TX 78712, USA swilliamson@utexas.edu
[3] MIT CSAIL, Cambridge, MA 02139, USA, {vchuravy,wmoses}@mit.edu
[4] Technical University of Munich, Munich 78712, Germany
ludger.paehler@tum.de

**Abstract.** Automatic differentiation (AutoDiff) in machine learning is largely restricted to expressions used for neural networks (NN), with the depth rarely exceeding a few tens of layers. Compared to NN, numerical simulations typically involve iterative algorithms like time steppers that lead to millions of iterations. Even for modest-sized models, this may yield infeasible memory requirements when applying the adjoint method, also called backpropagation, to time-dependent problems. In this situation, checkpointing algorithms provide a trade-off between recomputation and storage. This paper presents the package *Checkpointing.jl* that leverages expression transformations in the programming language Julia and the package ChainRules.jl to automatically and transparently transform loop iterations into differentiated loops. The user may choose between various checkpointing algorithm schemes and storage devices. We describe the unique design of *Checkpointing.jl* and demonstrate its features on an automatically differentiated MPI implementation of Burgers' equation on the Polaris cluster at the Argonne Leadership Computing Facility.

**Keywords:** Julia · Automatic differentiation · Checkpointing

## 1 Introduction

Automatic differentiation [8] (AutoDiff) is a technique for generating derivatives of a given implemented function $y = f(x)$ with input $x \in \mathbb{R}^n$ and output $y \in \mathbb{R}^m$, by differentiating the code at the statement level and applying the chain rule of derivative calculus. The differentiated code is required in optimization, nonlinear partial differential equations (PDE), sensitivity analysis, inverse problems, and

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_37

machine learning. The associativity of the chain rule leads to two main modes of code differentiation: the forward mode and the reverse mode. The forward mode computes the Jacobian-vector product $\dot{y} = \nabla J(x) \cdot \dot{x}$ with $\dot{}$ denoting the tangents or directional derivatives. The reverse mode, also known as backpropagation in machine learning, computes the transposed Jacobian-vector product $\bar{x} = \bar{y} \cdot \nabla J(x)$, with $\bar{}$ denoting adjoints. Note that the adjoint of the input $\bar{x}$ is computed with respect to the adjoint of the output $\bar{y}$. This implies a data flow reversal throughout the entire program. During the forward run $y = f(x)$, all the *intermediate values* of $x$ at each statement need to be stored for the reverse run $\bar{x} = \bar{y} \cdot \nabla J(x)$. This comes at a high cost of memory, increasing its complexity to at least the runtime complexity when assuming nonlinear functions $f$. The upside of the reverse mode is that the gradient of a scalar function $f$ with $m = 1$ can be computed at $\mathcal{O}(1) \cdot cost(f)$ versus $\mathcal{O}(n) \cdot cost(f)$ for the forward mode. As a remedy, checkpointing in AutoDiff refers to a trade-off between recomputation and the memory requirement for storing the intermediate values.

In this paper, we will focus on the common pattern of time-stepping loops or iterative loops in general that appear in numerical simulations further explained in Section 1.1 and apply it to the Burgers' equation (see Figure 1). For the first time, through expression transformations and code reflection in Julia, we make checkpointing for iterative loops in AutoDiff fully transparent to the user.

### 1.1 Adjoint Timestepping Checkpointing

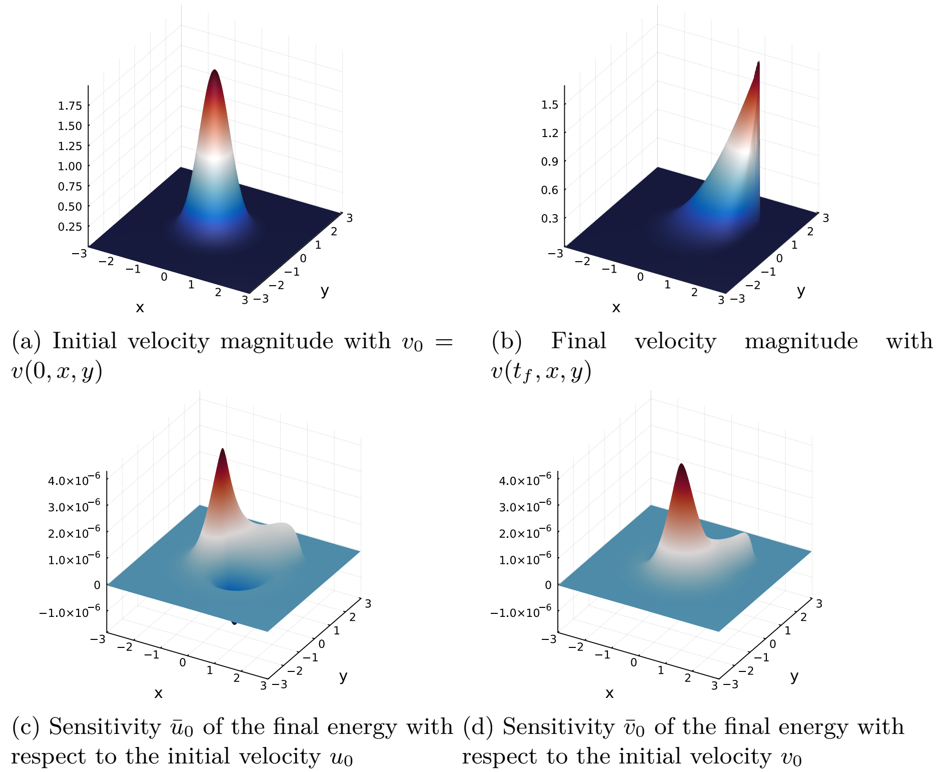
Most numerical problems require the evaluation of nonlinear expressions, either due to direct nonlinear function expressions (e.g., polynomials, trigonometric functions, etc.) or due to the evaluation of conditional expressions (e.g., IF-ELSE). Furthermore, these expressions are found in iterative sequences, either as part of a time-stepping model or an iterative solver (or both). In reverse-mode AutoDiff, these variables are required in reverse order compared to the execution of the nonlinear primal model (see $f$ and $\bar{f}$ in Figure 2). Two extreme approaches exist to access these variables, either storing all (see Figure 2) or recomputing all that are necessary. For complex models, neither of these approaches is practical. Checkpointing provides a computational solution that can help circumvent these issues by reducing the amount of storage at the expense of increased run time.

One well-known use is the computation of the so-called adjoint (gradient) of a model-data misfit (or cost) function, as is done in data assimilation based on gradient-based, PDE-constrained optimization. For example, the gradient of a cost function with respect to a very high-dimensional space of control variables via minimization of a Lagrangian,

$$\mathcal{L} = J - \sum_{t=1}^{t_f} \bar{\mathbf{x}}_t \left( \mathbf{x}_t - f(\mathbf{x}_{t-1}) \right), \tag{3}$$

where $J$ is a previously defined cost function and, in general, requires knowledge of all forward steps. In this notation, $\mathbf{x}_t$ refers to the model state at time $t$, and $f$ is a nonlinear model that steps the state from time $t-1$ to time $t$. In this

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_37




(a) Initial velocity magnitude with $v_0 = v(0, x, y)$

(b) Final velocity magnitude with $v(t_f, x, y)$

(c) Sensitivity $\bar{u}_0$ of the final energy with respect to the initial velocity $u_0$

(d) Sensitivity $\bar{v}_0$ of the final energy with respect to the initial velocity $v_0$

Fig. 1: Adjoint solution to Burgers' equation with $dx = 3e - 2$, $dy = 3e - 2$, $dt = 3e - 3$, and $\nu = 1e - 2$ on a grid $(Nx, Ny) = (1\,000, 1\,000)$ and $10\,000$ timesteps. This requires around $1000^2 \times 10\,000 \times 4$ fields $\times$ 8B $=$ 320GB of memory to **store** all the intermediate timesteps for the adjoint computation. Our solution enables a user to transparently trade this high memory footprint for a runtime overhead of around $10 - 12$ while reducing the footprint to 1.6GB

(and other examples), where the numerical state at each time step $t$ may be of the order $10^5 - 10^7$, and with iteration (i.e., time-stepping) loops of the order $10^4 - 10^6$ keeping the required state in memory is not feasible. The solution is to use checkpointing. Instead of storing all system states during the forward pass, "checkpoints" at specified intervals are stored on disk, which can subsequently be restored to recompute future states.

**Adjoint method** The adjoint method aims to minimize the Lagrangian described in (3) to compute the adjoint variables, $\bar{\mathbf{x}}_t$. Say the cost function is given by $J(\mathbf{x}_{t_f})$, and we wish to know how $J$ depends on the initial condition $\mathbf{x}_0$. This sensitivity is captured in $\bar{\mathbf{x}}_0$, the adjoint variable at the initial time. Taking a derivative of (3) with respect to $\bar{\mathbf{x}}_t$, one finds the first normal equation, (1), the

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_37

$$\mathbf{x}_{t+1} = f(\mathbf{x}_t) \tag{1}$$

$$\bar{\mathbf{x}}_t = \bar{f}(\mathbf{x}_t, \bar{\mathbf{x}}_{t+1}) = \frac{\partial f(\mathbf{x}_t)}{\partial \mathbf{x}_t} \bar{\mathbf{x}}_{t+1} \tag{2}$$

Fig. 2: Evaluation process of iteratively applying function $f$ for $t = 1 : 9$ iterations, $f$ is called with state $x_t$ as the input and state $x_{t+1}$ as the output. The adjoint function $\bar{f}$ of $f$ computes state $\bar{x}_t$ with respect to state $\bar{x}_{t+1}$ and $x_t$. The red down and up arrows mark a stored and restored state, respectively.

forward evolution. The second normal equation (2) is found via the derivative of (3) with respect to $\mathbf{x}_t$, and gives a rule for stepping backward to compute the adjoint variables. The initial value for the back-propagation described by (2) is found as

$$\bar{\mathbf{x}}_{t_f} = \frac{\partial J}{\partial \mathbf{x}_{t_f}}. \tag{4}$$

Equation (2) shows why all states are necessary for computation of the adjoint variables when $f_t$ is nonlinear (i.e. computation of $f_t(x_t)$ will require knowledge of prior states), and thus why checkpointing is an essential tool. A schematic of computing the forward and backward problems is given in Figure 2.

The adjoint method has many applications throughout geophysical sciences. Most notable are data assimilation, in which the cost function is a data misfit, and sensitivity analysis, where the cost function is a physical quantity of interest. In this paper, we employ the adjoint method for sensitivity analysis of solutions to the Burgers' equation.

### 1.2 Contribution

Checkpointing capability has been implemented in source transformation AutoDiff tools as well as popular differentiable programming frameworks for machine learning. In this work, we show how languages that support code reflection or metaprogramming can be leveraged to make checkpointing for AutoDiff of loops fully transparent to the user. While we use the programming language Julia, the various constraints and generalizations laid out in the design section Section 2 can be extrapolated to other programming languages. This significantly improves the user experience for inexperienced AutoDiff users who run into memory bottlenecks when differentiating their time-dependent numerical code.

We implemented our design in the software package *Checkpointing.jl* [5]. It currently supports

[5] https://github.com/Argonne-National-Laboratory/Checkpointing.jl

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_37

- automated generation of the store and restore for the checkpointed object type,
- modular support of three checkpointing schemes: periodic, binomial, and online,
- and modular support of two storage devices Array and HDF5 files.

### 1.3 Use Case: Burgers' Equation

Checkpointing will be applied to the two-dimensional Burgers' equation [6]

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = \nu \nabla^2 u \tag{5}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = \nu \nabla^2 v \tag{6}$$

where $u$ and $v$ represent the $x$ and $y$ velocities of a fluid and $\nu$ is the viscosity coefficient. The equation is solved on a square domain, $(x, y) \in [-L, L] \times [-L, L]$, with the initial velocities

$$u(0, x, y) = \exp\left(-x^2 - y^2\right), \quad v(0, x, y) = \exp\left(-x^2 - y^2\right),$$

and Dirichlet conditions on all four boundaries

$$u(t, x, -L) = u(t, x, L) = u(t, -L, y) = u(t, L, y) = 0.$$

An identical boundary condition is imposed on $v$.

To discretize the system, we use a centered finite difference scheme in space and an explicit forward Euler scheme in time.

**Adjoint example** Let $N_x$, $N_y$ be the total number of grid points in $x$ and $y$, respectively. Using the notation from Section 1.1 we define the cost function

$$J = \frac{1}{N_x \cdot N_y} \sum_{j=1}^{N_x} \sum_{k=1}^{N_y} \left(u(t_f, x_j, y_k)^2 + v(t_f, x_j, y_k)^2\right), \tag{7}$$

a measure of total kinetic energy in the system at the final time $t_f$. The interest then lies in computing

$$\bar{u}_0 = \frac{\partial J}{\partial u(0, x, y)}, \quad \bar{v}_0 = \frac{\partial J}{\partial v(0, x, y)}, \tag{8}$$

the sensitivity of the final energy with respect to the initial velocities. A schematic of computing the forward and backward problems is given in Figure 2, and Equations 1, 2 demonstrate why all states are necessary for computation of the adjoint variables when $f_t$ is nonlinear (i.e. computation of $f_t(x_t)$ will require knowledge of prior states), and thus why checkpointing is an essential tool.

[6] https://github.com/DJ4Earth/Burgers.jl

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_37

## 2 Design Of *Checkpointing.jl*

The goal of *Checkpointing.jl* is to implement a fully transparent and flexible solution for adjoint checkpointing in timestepping loops. This includes (1) the automated store and restore of the checkpointed variables, (2) support of multiple checkpointing schemes, and (3) support for various types of storage devices.

*Model Object* To achieve this goal, we define a standard structure in Julia of such timestepped models. These codes use a context model object where the state of the current model is stored. This style of writing code is very common in Julia as it allows to dispatch methods on the object type using the language's multiple dispatch feature. In our case, the model type is `Burgers` as shown in Listing 1.1.

Burgers struct

```julia
mutable struct Burgers
  nextu::Matrix{Float64}
  nextv::Matrix{Float64}
  lastu::Matrix{Float64}
  lastv::Matrix{Float64}
  Nx::Int
  Ny::Int
  μ::Float64
  dx::Float64
  dy::Float64
  dt::Float64
  tsteps::Int
  ...
end
```

**Listing 1.1:** Model datatype that the timestepping loop will be dispatched on.

The model includes a field of type Matrix for $u$ and $v$ and for each a next and last storage place for the stencil where `next` is computed from `last`. In addition, it includes all the model parameters $\nu$, $dt$, $dx$, $dy$, and the grid size $N_x$ and $N_y$. Only the fields $u$ and $v$ need to be checkpointed. However, this requires the user to manually specify all the variables that are required in the adjoint computation. To alleviate this, we checkpoint the entire struct. This is an overestimation, but it enables us to automate the adjoint checkpointing, rendering it fully transparent. The assumption is that most of the memory required to store the struct is associated with variables required in the adjoint computation.

To store the checkpoints *Checkpointing.jl* currently implements two storage types. `ArrayStorage <: AbstractStorage` is used to store the checkpoints in RAM whereas `HDF5Storage <: AbstractStorage` is used to store them in an HDF5 file. For binary storage in a file we use Julia's built-in `Serialization` module to serialize the `Burgers` struct into disk-storable data. To extend *Checkpointing.jl* with additional storage devices, one can easily add another storage type derived from `AbstractStorage` and add an implementation of `getindex` and `setindex` method for the storage device, which allows the `[]` operator to be used for all stores and restores of a checkpoint with index $i$ (see Listing 1.2).

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_37

```julia
# Array storage implementation of
# right-hand read [] operator
Base.getindex(
  storage::ArrayStorage{MT}, i
) where {MT} = storage._storage[i]

# HD5 storage storage implementation
# of right-hand read [] operator
function Base.getindex(
  storage::HDF5Storage{MT}, i
)::MT where {MT}
  blob = read(storage.fid["$i"])
  return Serialize.deserialize(blob)
end
```

**Listing 1.2:** Example of `[]` operator (`getindex`) for restoring `ArrayStorage` and `HD5Storage`

*Loops* Relying on this abstraction, our timestepping loop is written as a `for` loop over the number of timesteps with an advance function and a halo exchange for the MPI implementation (see Listing 1.3). Note that the loop's body can be composed of any arbitrary code. In addition, *Checkpointing.jl* also supports `while` loops. It is important that the loop iterator bounds for the for loop and the variables in the evaluation of the while condition belong to the model object, here `burgers.tsteps`.

Final energy with `final_energy`

```julia
function final_energy(
  burgers::Burgers,
  scheme::Scheme,
)
  @checkpoint_struct scheme burgers
    for i in 1:burgers.tsteps
      advance(burgers)
      halo(burgers)
      copyto!(burgers.lastu, burgers.nextu)
      copyto!(burgers.lastv, burgers.nextv)
    end
  return energy(burgers)
end
```

**Listing 1.3:** Timestepping loop implementation with a single time step (`advance`), halo exchange using MPI (`halo`), and field swaps with Julia's `copyto!` function.

*Differentiation of Loops via Expression Transformations* In *Checkpointing.jl* we treat `for` and `while` loops as just another function that can be differentiated with the additional benefit of applying a checkpointing scheme that drastically reduces the memory footprint for storing the intermediate values. To achieve this we create a marco `@checkpoint_struct` that transforms *for* loops into function calls (see Listing 1.4). Using this macro as a decorator in Listing 1.3 allows the user to mark a loop to be differentiated using *Checkpointing.jl* by transforming it into a function call that is differentiated based on a rule defined in Section 2. In

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_37

addition to this transformation, we make a copy of the original model object and create a *s*hadow copy that is used to store the adjoints of the adjoint evaluation.

`checkpoint_struct` macro

```
macro checkpoint_struct(alg, model,
  loop)
  if loop.head == :for
    ex = quote
      shadowmodel = deepcopy($model)
      function range()
        $(loop.args[1])
      end
      $model = checkpoint_struct_for(
        $alg,
        $model,
        shadowmodel,
        range
      ) do $model
        $(loop.args[2])
      end
    end
  elseif loop.head == :while
    ...
  end
  esc(ex) # Return expression
end
```

**Listing 1.4:** Loop transformation

Now, we must make the AutoDiff tool aware of how to differentiate the `checkpoint_struct_for` function call. Multiple efforts exist to standardize differentiation rules. Most AutoDiff tools differentiate the language's intrinsic operations like arithmetic operations (e.g., multiplication, addition) and certain special functions (e.g., cosine, sine). However, higher-level functions (e.g., linear solvers) or rarely used special functions like BesselK [1] are rarely supported out of the box and have to be defined as *external functions*. In Julia, the popular package ChainRules.jl [11] allows the specification of differentiation rules which AutoDiff tools may then rely on to apply the chain rule. That way, the differentiation rules do not have to be reimplemented for each AutoDiff tool. We refer the reader to the manual of ChainRules.jl for the details on defining such differentiation rules. In summary, it requires a user to define a rule for forward mode differentiation (`frule`) and a reverse mode differentiation rule (`rrule`). By defining those two rules, any combination of higher-order models using, for example, a forward over forward or forward over reverse model, may be generated by an AutoDiff tool. Our reverse rule is presented in Listing 1.5. ChainRules.jl implements joint reversal (Figure 3) for external functions (see [8] for more details). The outer loop AutoDiff tool will execute the *augmented forward run* of the 'Before" block (green) and store all intermediate values. When this tool hits the checkpointed loop it will apply our rule. The rule is composed of the forward run implementing the original function (orange). Then it defines a callback or pullback function that the outer AutoDiff tool will execute once it has executed the *reverse run* (blue) of the 'After' block. This pullback will set the adjoints of the time loop shadow model to zero and then copy the computed adjoints of the 'After' block into the starting adjoints or *seeds* of the time loop. Now, the augmented forward run (green) of the time loop will be executed in `checkpoint_struct_for`, followed by the reverse run (blue)

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_37

ChainRules.jl implementation

```
function ChainRulesCore.rrule(::typeof(Checkpointing.checkpoint_struct_for),
    body::Function, alg::Scheme, model::MT, shadowmodel::MT,
    range::Function) where {MT}
  model_input = deepcopy(model)
  for i in 1:alg.steps
    body(model)
  end
  function checkpoint_struct_pullback(dmodel)
    set_zero!(shadowmodel)
    copyto!(shadowmodel, dmodel)
    model = checkpoint_struct_for(body, alg, model_input, shadowmodel, range)
    dshadowmodel = create_tangent(shadowmodel)
    return NoTangent(), NoTangent(), NoTangent(), dshadowmodel, NoTangent(),
      NoTangent()
  end
  return model, checkpoint_struct_pullback
end
```

**Listing 1.5:** Reverse rule for time loop

based on the selected checkpointing scheme. After the adjoints are computed, they are again copied back into the respective seeds for the 'Before' block using `create_tangent`. Note that all other arguments of `checkpoint_struct_for` are passive and do not need to be differentiated. This is marked by `NoTangent()`.

Such a differentiation rule may be defined for other differentiation rule systems that may be general or AutoDiff tool specific. ChainRules.jl covers the most general case, while other rule systems may include other attributes.

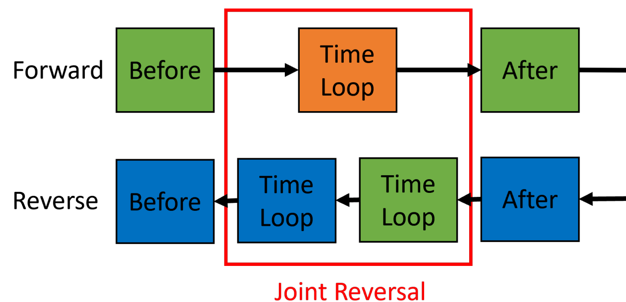



Fig. 3: Adjoining a time loop embedded into another code using ChainRules.jl. (green) denotes an *augmented forward* run where all the intermediate variables are stored. (blue) denotes a *reverse* run where the intermediate variables of the augmented forward run are used for the evaluation of the adjoints. (orange) is the original undifferentiated function evaluation.

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_37

*Modular Support of Schemes* The package currently provides three checkpointing schemes

- **Periodic Checkpointing**: For a computation consisting of $l$ timesteps and $c$ available checkpoints, the periodic checkpointing scheme that stores the input and the output of each $\lfloor \frac{l}{c} \rfloor$ iterations and restores them for computing the adjoint [3].
- **Binomial Checkpointing**: For a computation consisting of $N$ time steps with the availability of $c$ checkpoints, binomial checkpointing [2] gives a formulation for the minimal number of time steps $t(l,c)$, evaluated during the adjoint calculation with $t(l,c) = rl - \beta(c+1, r-1)$ where $\beta(c,r) = \binom{c+r}{c}$ and the repetition number $r$ is the unique integer, such that $\beta(c, r-1) < l \leq \beta(c,r)$. We have ported the software `revolve` for providing an implementation of the binomial checkpointing algorithm.
- **Online Checkpointing**: In adaptive time-stepping procedures, the number of time steps, $l$, is not known a priori. Periodic checkpointing and binomial checkpointing are therefore not appropriate here. The online checkpointing scheme determines during the first forward integration where a checkpoint must be placed. Given the number of available checkpoints $c$, and the repetition number $r$, it is possible to determine the range of timesteps $l$ for which the online checkpointing scheme generates an optimal schedule [10]. We have currently implemented the cases where $r = 1$ and $r = 2$.

Listing 1.6 gives an overview of the supported checkpointing schemes and storage devices. The created scheme `Scheme <: AbstractScheme` has to be passed to the macro `@checkpoint_struct` together with the checkpointed object.

Example code

```
1  checkpoints = 50
2  tsteps = 10000
3  # Storage in RAM
4  storage = ArrayStorage{Burgers}(checkpoints)
5  # Storage to disk with HDF5
6  rank = MPI.Comm_rank(MPI.COMM_WORLD)
7  storage=HDF5Storage{Burgers}(snaps, filename="$rank.chkp")
8  # Storage to on-node SSD with HDF5
9  storage=HDF5Storage{Burgers}(snaps, filename="/local/scratch/$rank.chkp")
10 # Our three currently supported checkpointing schemes
11 scheme = Revolve{Burgers}(tsteps, snaps, verbose=1, storage=storage)
12 scheme = Periodic{Burgers}(tsteps, snaps, verbose=1, storage=storage)
13 # No tsteps needed for Online scheme!
14 scheme = Online_r2{Burgers}(snaps, verbose=1, storage=storage)
```

**Listing 1.6:** Example of checkpointing schemes and storage object instantiations based on the `Burgers` type

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_37

## 3 Implementation

Our implementation is available at [9]. It currently supports three checkpointing schemes (Periodic, Revolve, and Online_R2). It distinguishes between an outer AutoDiff tool for differentiating the code outside the loop and an inner AutoDiff tool that is used to differentiate the actual loop body. Both tools can be the same; however, the outer AutoDiff tool has to support differentiation rules through ChainRules.jl. Consider the Burgers' example, we apply the `@checkpoint_struct` to the timestepping loop and the code computes the energy with the `energy` function (Listing 1.3). The outside code uses the AutoDiff package Zygote.jl while the timestepping loop is differentiated with Enzyme.jl. The loop body consists of an `advance` function implementing one forward time step and `halo` implementing the halo exchange (see Section 3).

Enzyme [6,7,5] is an AutoDiff tool acting on the LLVM IR. It, therefore, supports C++ and Julia alike, with the Julia package Enzyme.jl providing Julia-specific support. The novel advantage of Enzyme is its optimization capabilities. AutoDiff tools are usually not integrated directly into a compiler, but use either language-inherent features like operator overloading or are implemented as a separate parsing and generation process before the code is passed to the compiled (source transformation). Enzyme, on the other hand, uses parts of the LLVM optimization pipeline, then differentiates the code, and finally, this IR is again optimized before the code is finally passed to the machine code generation. This three-stage process adds unique performance capabilities to Enzyme that other AutoDiff tools have trouble achieving.

In our example, we use Enzyme to differentiate the inner loop body. Any AutoDiff tool can be used here if it implements Jacobian-transposed vector products, which is the basic operation in the reverse mode of AutoDiff. Although Enzyme.jl does currently not support ChainRules.jl, it is not a requirement for the inner AutoDiff tool in *Checkpointing.jl*, and we can use it in our test case. A similar differentiation rule system is in development for Enzyme.

Zygote.jl [4] is an AutoDiff package originally designed for machine learning. As such, it lacks the support of mutation. This implies that in-place manipulation of array elements is impossible and requires a code to be written without any mutation, which our code outside the loop adheres to. Zygote.jl treats the underlying LLVM IR as static single assignment code and allows the compiler to highly optimize the generated differentiated code. However, due to its limitation to immutable code, it is not well suited for numerical simulations where in-place manipulation of values is common.

*MPI* The `halo` function uses MPI send and receives to do the nearest neighbor halo exchange in all 4 directions of the 2D discretized Burgers' equation. The outside code uses MPI for the summation reduction of the local energy to the global energy of the velocity fields $u$ and $v$. Enzyme.jl has intrinsic support of MPI, whereas Zygote is not aware of MPI. We added a ChainRules.jl rule for the MPI reduction that allows Zygote to differentiate through this method for the summation.

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_37

## 4 Results

Our experiments are conducted on an HPE Apollo 6500 Gen 10+ based system. Each node has a single 2.8GHz AMD EPYC Milan 7543P 32-core CPU with 512GB of DDR4 RAM and four Nvidia A100 GPUs connected via NVLink, a pair of local 1.6TB of SSDs in RAID0 as on-node scratch disks, and a pair of slingshot network adapters. For the timings, we used the Julia 1.8 built-in macro `@time` and the BenchmarkTools.jl provided `@btime`. To maximize the throughput of our code and avoid any overhead, we optimized it gradually based on PProf.jl results. An estimate of the total memory footprint was reported by the maximum resident memory through `/usr/bin/time`.

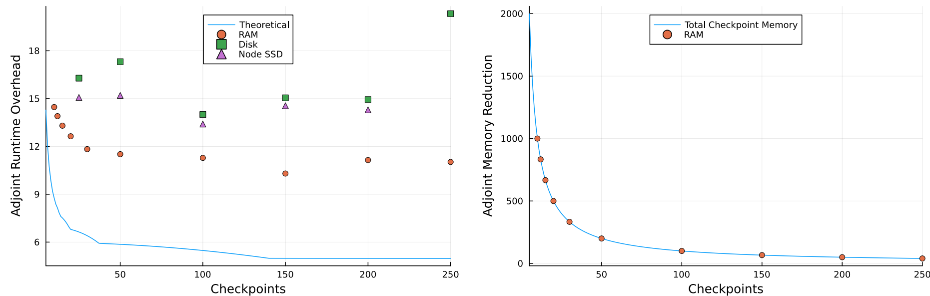
For the large-scale runs, we increase the grid size to $(N_x, N_y) = (10\,000, 10\,000)$. To achieve the same final state as in Figure 1 the number of timesteps needs to be increased to 100 000. However, due to compute time limitations, we reduce this to 10 000. This has no effect on the overall claims of this paper. Other runtime parameters are $dx = 0.01$, $dy = 0.01$, $dt = 0.001$ with 100 ranks and a checkpoint size of 32MB.

The entire case of $10\,000 \times 10\,000$ grid points is partitioned among 100 MPI ranks. The goal of the increased resolution is to reduce the numerical error introduced by the sharp gradients at the shock boundary in Figure 1. Each node has 32 cores, so we distribute the 100 MPI ranks over 4 nodes which in total have 2TB of RAM. Each rank gets a partition of the $10^6$ points, which amounts to roughly 8MB. Each `Burgers` object includes 4 of these fields: `nextu`, `nextv`, `lastu`, and `lastv`. This gives us a total checkpoint size of 32MB, which agrees with our observed disk checkpoint file sizes. To store all 10 000 time steps, this would amount to 320GB per process or 32 TB for all 100 processes. This implies that we cannot run our case without checkpointing at all because we have only 2TB of RAM available. In Figure 4a and Figure 4b, we compare the relative runtime and memory overhead of the adjoint computation compared to the primal evaluation of the final energy. In addition, we use checkpointing to RAM, to disk, and to a local on-node SSD drive.

The theoretical runtime is derived from the sum of additional forward steps that binomial checkpointing requires and the joint adjoint reversal that is implemented using ChainRules.jl (see Figure 3). Joint reversal incurs a cost of at least a factor of 3 in integrating the loop function into ChainRules.jl. In addition, we add the forward steps necessary for binomial checkpointing. If the number of checkpoints equals the number of time steps *tsteps*, binomial checkpointing still executes *tsteps* forward steps. So in the most optimistic case, we end up with an overhead factor of 4. With fewer checkpoints than time steps, we can compute the required forward steps laid out in the binomial checkpointing analysis in [2]. The sum of all required steps gives us the theoretical overhead factor in Figure 4a.

Each data point in the graph is computed from the average execution time of 3 separate runs. We did not obtain results for the 250 checkpoints data point of the "Node SSD" storage device due to instability with HDF5. In general, we are impacted by noise in our test runs. Due to compute time limitations,

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_37




(a) The *adjoint runtime overhead* is the ratio between the total adjoint computation and the total forward computation of the original code $\frac{T_f}{T_r}$ for a given number of checkpoints. The theoretical overhead is based on an analytic formula in binomial checkpointing and assumes no memory latency. The random memory access of binomial checkpointing is expected to introduce an overhead using RAM, disk, and on-node SSD checkpointing.

(b) The memory reduction can be accurately predicted as the memory needed without checkpointing equal to the number of checkpoints times the object size divided by the number of checkpoints time memory times the object size $\approx \frac{\text{tsteps}}{\text{snaps}}$. Due to the object size being dominated by the TBR variables of $u$ and $v$, the difference between theoretical and practical overhead is negligible.

Fig. 4: Results of adjoint runtime overhead (a) and memory consumption (b)

we are unable to provide a thorough statistical analysis of this noise. However, we can extract some patterns based on the results. First, we observe that RAM checkpointing yields the fastest results with an overhead of around 10-12 between 50 and 250 checkpoints. There is no substantial benefit to increasing the number of checkpoints beyond 50. Second, we have a general pattern from fast to slowest of RAM, on-node, and disk checkpointing, with disk checkpointing being the slowest and the one most impacted by noise since it is most affected by other jobs running on the system.

The memory reduction is the total number of time steps divided by the number of checkpoints. We measured 32MB per checkpoint per process. So multiplying the number of checkpoints by 32MB and by 100 processes gives the actual memory requirement ranging from 64GB for 20 checkpoints up to 0.8TB for 250 checkpoints. Due to MPI parallelism, this memory requirement is divided among 4 compute nodes. This is a dramatic reduction from the 32TB required for storing all 10 000 time steps. Moreover, on-node SSD and disk checkpointing have the additional benefit of reducing the RAM overhead to zero, providing more RAM for the actual application. This allows for a decrease in the number of partitions and potentially reduces the required compute time spent on the run despite exhibiting a higher wall clock time. On-node SSD checkpointing provides an overhead of around 13 with more regular runtime results than disk checkpointing. Thus, it may provide the right compromise for this application.

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_37

## 5 Conclusion

We have implemented an extendable and flexible time-loop checkpointing package in Julia that can be integrated into any code that supports AutoDiff based on ChainRules.jl. According to their webpage, 6 AutoDiff tools currently support ChainRules.jl, with more in the works. Our macro-based solution is non-invasive and only requires the user to create a checkpointing scheme object with the desired parameters and decorate the checkpointed loop with our `@checkpoint_struct` macro. It relies on a common abstraction found in numerical simulations where models are encapsulated in a single context object based on a model type. Our results show the flexibility and performance of *Checkpointing.jl* illustrated by a canonical nonlinear PDE implementation of the Burgers' equation that runs on a state-of-the-art supercomputer with minimal development effort and without introducing any domain-specific language. It allows for fast testing of various checkpointing schemes and storage devices. The user may implement their own scheme or storage devices with a few lines of code without worrying about the technicalities of the underlying AutoDiff tool. In theory, such a package may be implemented in any programming language. However, the access to expression transformation in Julia reduces the complexity for both users and developers significantly, increases the modularity of the code, and avoids any restriction to a domain-specific language. All available storage options in *Checkpointing.jl* are currently implemented using synchronous reads and writes. Furthermore, although binomial checkpointing has a random access pattern, it does access the memory locations deterministically according to the binomial checkpointing algorithm. We will investigate the asynchronous prefetching of the next checkpoint concurrently with the adjoint computation relative to the last checkpoint.

**Acknowledgements** We would like to thank Paul Hovland and Jan Hückelheim for their valuable suggestions and discussions. This work was funded and/or supported by NSF Cyberinfrastructure for Sustained Scientific Innovation (CSSI) award numbers: 2104068, 2103942, and 2103804, Argonne Leadership Computing Facility, which is a U.S. Department of Energy (DOE) Office of Science User Facility supported under Contract DE-AC02-06CH11357, DOE Computational Sciences Graduate Fellowship, NSF (grants OAC-1835443, AGS-1835860, and AGS-1835881), DARPA under agreement number HR0011-20-9-0016 (PaPPa), Schmidt Futures program, Paul G. Allen Family Foundation, Charles Trimble, Audi Environmental Foundation, DOE, National Nuclear Security Administration under Award Number DE-NA0003965, LANL grant 531711, and German Research Council (DFG) under grant agreement No. 326472365. Research was sponsored in part by the US Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United

States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein. This material is based upon work supported by the DOE, Office of Science, Office of Advanced Scientific Computing Research.

## References

1. Geoga, C.J., Marin, O., Schanen, M., Stein, M.L.: Fitting matérn smoothness parameters using automatic differentiation. Statistics and Computing **33**(2), 48 (2023). https://doi.org/10.1007/s11222-022-10127-w
2. Griewank, A., Walther, A.: Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. ACM Trans. Math. Softw. **26**(1), 19–45 (mar 2000). https://doi.org/10.1145/347837.347846
3. Griewank, A., Walther, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. No. 105 in Other Titles in Applied Mathematics, SIAM, Philadelphia, PA, 2nd edn. (2008), http://bookstore.siam.org/ot105/
4. Innes, M.: Don't unroll adjoint: Differentiating ssa-form programs (2018). https://doi.org/10.48550/ARXIV.1810.07951
5. Moses, W.S., Narayanan, S., Paehler, L., Churavy, V., Schanen, M., Huckelheim, J., Doerfert, J., Hovland, P.: Scalable automatic differentiation of multiple parallel paradigms through compiler augmentation. In: SC22: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–18. IEEE Computer Society, Los Alamitos, CA, USA (nov 2022). https://doi.org/10.1109/SC41404.2022.00065
6. Moses, W., Churavy, V.: Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M.F., Lin, H. (eds.) Advances in Neural Information Processing Systems. vol. 33, pp. 12472–12485. Curran Associates, Inc. (2020), https://proceedings.neurips.cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf
7. 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. SC '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3458817.3476165
8. Naumann, U.: The Art of Differentiating Computer Programs. Society for Industrial and Applied Mathematics (2011). https://doi.org/10.1137/1.9781611972078
9. Schanen, M., Narayanan, S.H.K.: Argonne-National-Laboratory/Checkpointing.jl: v0.6.3 (Feb 2023). https://doi.org/10.5281/zenodo.7607916
10. Stumm, P., Walther, A.: New algorithms for optimal online checkpointing. SIAM Journal on Scientific Computing **32**(2), 836–854 (2010). https://doi.org/10.1137/080742439
11. White, F.C., Abbott, M., Zgubic, M., Revels, J., Axen, S., Arslan, A., Schaub, S., Robinson, N., Ma, Y., Dhingra, G., Tebbutt, W., Heim, N., Widmann, D., Schmitz, N., Rosemberg, A.D.W., Rackauckas, C., Lucibello, C., Heintzmann, R., frankschae, Noack, A., Fischer, K., Robson, A., de Cossio-Diaz, J.F., Ling, J., mattBrzezinski, Finnegan, R., Zhabinski, A., Wennberg, D., Besançon, M., Vertechi, P.: JuliaDiff/ChainRules.jl: v1.45.0 (Nov 2022). https://doi.org/10.5281/zenodo.7312560

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_37