Parallel Adjoint Taping using MPI

Markus Towara[1 [0000−0002−7520−7175]], Jannick Kremer[1 [0009−0004−7667−5060]], and Uwe Naumann[1 [0000−0002−7518−5922]]

Software and Tools for Computational Engineering,
RWTH Aachen University, Germany
towara@stce.rwth-aachen.de
https://www.stce.rwth-aachen.de

Abstract. The adjoint reversal of long evolutionary calculations (e.g. loops), where each iteration depends on the output of the previous iteration, is a common occurrence in computational engineering (e.g. computational fluid dynamics (CFD) simulation), physics (e.g. molecular dynamics) and computational finance (e.g. long Monte Carlo paths). For the edge case of a scalar state, the execution, as well as adjoint control flow reversal, are inherently serial operations, as there is no spatial dimension to parallelize. Our proposed method exploits the run time difference between passive function evaluation and augmented forward evaluation, which is inherent to most adjoint AD techniques. For high dimensional states, additional parallelization of the primal computation can and should be exploited at the spatial level. Still, for problem sizes where the parallelization of the primal has reached the barrier of scalability, the proposed method can be used to better utilize available computing resources and improve the efficiency of adjoint reversal.
We expect this method to be especially useful for operator-overloading AD tools. However, the concepts are also applicable to source-to-source transformation and handwritten adjoints, or a hybrid of all approaches. For illustration, C++ reference implementations of a low dimensional evolution (lorenz attractor) and a high dimensional evolution (computational fluid dynamics problem in OpenFOAM) are discussed. Both theoretical bounds on the speedup and run time measurements are presented.

Keywords: Algorithmic Differentiation, Adjoints, MPI

1 Introduction

Motivation

Adjoint sensitivities are desired in a variety of academic and industrial applications due to their high accuracy and low computational cost compared to forward methods (e.g. finite differences), for functions which map a high number n of input parameters to a low (if not scalar) number of outputs m ($f : \mathbb{R}^n \to \mathbb{R}^m$ with $m \ll n$). Such sensitivities can be efficiently computed with Algorithmic Differentiation (AD, commonly also referred to as Automatic Differentiation). With adjoint AD, the full Jacobian $\nabla f(x) : \mathbb{R}^n \to \mathbb{R}^{m \times n}$ can be obtained at a computational complexity $\mathcal{O}(m) \cdot cost(f)$.

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_29

The goal of this paper is to propose a novel way to better utilize parallel computing resources in the context of adjoint control flow reversal. Our proposed method exploits the run time difference between passive function evaluation and augmented forward evaluation (see definitions in Section 2).

The computation of adjoints typically consists of two distinct phases (outlined in more detail in the main matter):

Augmented forward evaluation Execution of function f, but all intermediate values required for the reverse propagation of adjoints are cached. Efficient AD tools will pre-accumulate[32,10] partial Jacobians required for the reversal process already during this step, reducing the amount of data to be stored and the number of operations to be performed during reverse propagation.

Reverse propagation of adjoints Evaluation of the adjoint model[11], restoring cached values as necessary and utilizing pre-accumulated Jacobians. Sensitivities will be propagated from the outputs back to the inputs and parameters.

The augmented forward evaluation is typically significantly slower than the *passive* (i.e. regular) evaluation of the same code, due to added memory bandwidth usage for the caching of values, added computational operations for the pre-accumulation of Jacobians, and (for some AD tools) added overhead due to the overloading of operators. The proposed method exploits this run time factor between *augmented forward* and *passive* evaluation by distributing the work to multiple processors and only performing a specific part of the calculation in *augmented forward* mode and remaining in *passive* mode for the remaining part. Depending on the implementation, this introduces additional *passive* computations, however, these are hidden in the parallelization and do not influence the wall clock time.

We mainly focus on C++ codes, where the overhead for operator-overloading is low, due to inlining and compile-time optimization. For interpreted languages (e.g. Python, MATLAB) the overhead can be much higher. General purpose operator-overloading tools include ADOL-C[33] (`C++`), CoDi-Pack[27] (`C++`), `dco/c++`[16], Sacado[25], and AdiMat[2] (Matlab).

General purpose AD tools which use the source code transformation approach[11] include TAPENADE[12] (C and Fortran), dcc[8] (C) and Tangent[9] (python). Recently, a lot of research has focused on providing source code transformation for languages based on the LLVM stack[15] (e.g. `C++`, Rust, but also interpreted languages like Julia). This research has produced frameworks such as Enzyme[20], Zygote[13], and Diffractor[6]. Furthermore, domain-specific tools exist, e.g. DolfinAdjoint[5] for the Dolfin[17] / FEniCS[1] packages for solving differential equations using FEM. Especially in the context of machine learning, a variety of new tools, often geared toward specific linear algebra operations, have been developed (e.g. JAX[3]).

For high dimensional states (e.g. discretization points in numerical simulations), additional parallelization of the primal computation can and should be exploited on the spatial level. Still, for problem sizes where the parallelization of the primal has reached the barrier of scalability, this method can be used to utilize remaining computing resources and improve the efficiency of adjoint reversal.

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_29

We expect the proposed method to be especially useful for operator-overloading AD tools, however, the concepts are also applicable to source-to-source transformation and handwritten adjoints.

Limitation of state-of-the-art approaches

Many parallelization schemes utilized in computational engineering face challenges once the number of involved processes gets very large due to data access patterns, communication overheads, and other factors[7]. Existing strategies for parallelizing adjoint computations often focus on assembling individual stencils (e.g.[19]), or involve reversing the communication patterns in MPI (e.g.[28,27]). Both of these approaches require significant code changes in order to incorporate adjoint computations. Our approach, though limited to evolution-style algorithms, does not require many changes below the outer iteration level, as mostly a black box differentiation[11] approach is retained.

Limitations of the proposed approach

The current implementation of the proposed method is limited to MPI parallelization. Conceptually, it can also be extended to OpenMP or GPU parallelism (e.g. CUDA), however, memory bandwidth limitations might reduce the effectiveness of the approach. As shown in Section 4.2, the scaling of the proposed method is bound by a constant factor. To achieve scaling onto a high number of processors it needs to be combined with a parallelization strategy for the underlying problem.

Data availability

The code for the case study in Section 5 and further illustrative code is available at `github.com/STCE-at-RWTH/parallel_taping`.

2 Notation and Foundations

We denote scalar variables in italics (e.g. $y \in \mathbb{R}$) and vectors in bold (e.g. $\mathbf{x} \in \mathbb{R}^n$). The corresponding adjoints are identified by $\bar{\mathbf{x}} \in \mathbb{R}^n$ and $\bar{y} \in \mathbb{R}$ (Notation of[11]). For a multivariate function $y = f(\mathbf{x})$ the adjoint model reads

$$\bar{\mathbf{x}} = \bar{y} \cdot \nabla f(\mathbf{x}) \,, \tag{1}$$

where $\nabla f(\mathbf{x})$ denotes the gradient of f evaluated at location $\mathbf{x}$. For reference, we also introduce the tangent model with tangents $\dot{\mathbf{x}} \in \mathbb{R}^n$ and $\dot{y} \in \mathbb{R}$

$$\dot{y} = \nabla f(\mathbf{x}) \cdot \dot{\mathbf{x}} \,. \tag{2}$$

Equations (1) and (2) can be evaluated for a given code implementation of f explicitly (by writing down the tangent/adjoint statements line by line, see code in GitHub) or implicitly by means of Algorithmic Differentiation (AD).

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_29

The gradient $\nabla f(\mathbf{x})$ can be calculated at cost $\mathcal{O}(1) \cdot cost(f)$, where $cost(f)$ is a measure proportional to the run time needed to evaluate f, by choosing $\bar{y} = 1$ and evaluating the adjoint model. To acquire the same gradient we need n calls to the tangent model (by letting $\dot{\mathbf{x}}$ range over the unit vectors e_i), yielding cost $\mathcal{O}(n) \cdot cost(f)$. Thus, the adjoint method is the natural choice for calculating gradients whenever $1 \ll n$.
For the adjoint mode, the data flow of the program has to be reversed to propagate adjoints from the program outputs back to the inputs. With the adjoint model, the program execution decomposes into a distinct forward and reverse section.

Conceptually, all computer codes can be transformed into the execution of elemental functions φ_j operating on a single vector of data $\mathbf{v} \in \mathbb{R}^{n+p+m}$, using n inputs (v_0 to v_{n-1}), p intermediate values (v_n to v_{n+p-1}) and m outputs (v_{n+p} to $v_{n+p+m-1}$)[10,21]. The corresponding adjoints can then be accumulated in $\bar{\mathbf{v}}$. In the following procedure $i \prec j$ denotes a (direct) dependence of variable v_j on v_i (that is $\frac{\partial \varphi_j}{\partial v_i} \neq 0$). Then the forward section computes the intermediates and output variables; Only after this is finished the reverse section starts to propagate the adjoints of outputs and intermediates back to the inputs.

$$\left.\begin{aligned} &\text{for } j = n, \dots, n+p+m-1 \\ &\quad v_j = \varphi_j(\mathbf{v}_i)_{i \prec j} \end{aligned}\right\} \text{forward section,}$$
$$\left.\begin{aligned} &\text{for } i = n+p-1, \dots, 0 \\ &\quad \bar{v}_i = \sum_{j : i \prec j} \frac{\partial \varphi_j(\mathbf{v}_i)}{\partial v_i} \cdot \bar{v}_j \end{aligned}\right\} \text{reverse section.} \tag{3}$$

Note, that the derivatives $\frac{\partial \varphi_j}{\partial v_i}$ required in the reverse section still potentially depend on $\mathbf{v}_i$ (if φ_j is non-linear in v_i). Thus, either the values $\mathbf{v}_i$, or the Jacobian $\frac{\partial \varphi_j}{\partial v_i}$ (pre-accumulation), need to be cached until they are consumed in the reverse section. This caching is where the majority of the adjoint methods' memory penalty stems from. Figure 1 illustrate the concept of Jacobian pre-accumulation on a per-statement level. This technique reduces the amount of data that needs to be stored for the reverse section and also allows to perform most of the calculations already during the forward phase. Statement level pre-accumulation can efficiently be implemented by `C++` expression template techniques[16].

3 Serial Adjoint Reversal of Evolution

First, we consider a serial evolution with state $\mathbf{x} \in \mathbb{R}^{n_x}$ (states are repeatedly overwritten by an iteration formula) and parameter $\mathbf{p} \in \mathbb{R}^{n_p}$ (parameters remain constant throughout the computation). Each iteration step only depends on the state $\mathbf{x}$ from the previous iteration and global parameters $\mathbf{p}$ (within each iteration additional intermediate variables might be used). To uniquely identify the states $\mathbf{x}$ during the different phases of the computation we introduce an upper index for each iteration. In practice, the state may use only a single memory location (thus overwriting the existing memory). However, for the adjoint reversal, it may be required to cache the states $\mathbf{x}$.

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_29

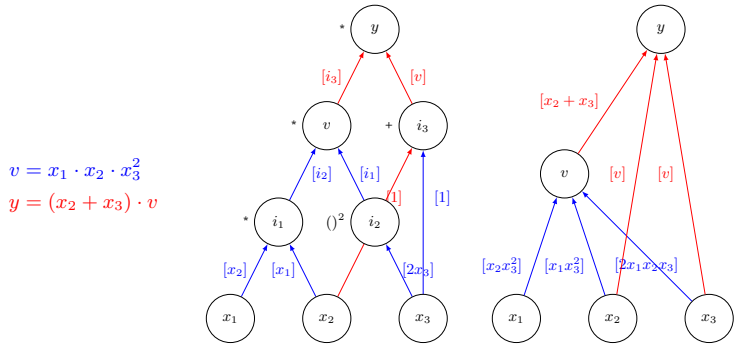


Fig. 1. Example for pre-accumulation of two statements. Elimination of intermediate nodes and edges from the middle graph saves memory for three nodes and four edges. Edges are labeled with partial derivatives.

```
1  double f(double x, const double p){ return p*sin(x); }
2
3  double evolution(int n, double x, const double p){
4    for(int i=0; i<n; i++)
5      x = f(x,p);
6    return x;
7  }
```

Listing 1.1. Example of evolution with function f, scalar input x, and scalar parameter p

First, we consider scalar evolutions ($n_x = 1, n_p = 1$) of the form:

$$x^n = f^n(x^{n-1}, p) \circ f^{n-1}(x^{n-2}, p) \circ \ldots \circ f^2(x^1, p) \circ f^1(x^0, p)$$

with arbitrary (differentiable) functions $f^i : \mathbb{R} \times \mathbb{R} \to \mathbb{R} \in C^1$. An example of such an evolution, where f is always the same iteration procedure $x = p \cdot \sin x$, is given in Listing 1.1. This can easily be generalized to vector-valued evolutions of the form:

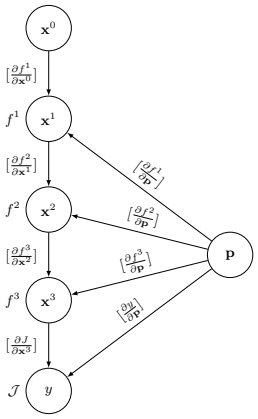
$$y = \mathcal{J} \circ f^n(\mathbf{x}^{n-1}, \mathbf{p}) \circ f^{n-1}(\mathbf{x}^{n-2}, \mathbf{p}) \circ \ldots \circ f^2(\mathbf{x}^1, \mathbf{p}) \circ f^1(\mathbf{x}^0, \mathbf{p})$$

with arbitrary (differentiable) functions $f^i : \mathbb{R}^{n_\mathbf{x}} \times \mathbb{R}^{n_\mathbf{p}} \to \mathbb{R}^{n_\mathbf{x}} \in C^1$ and a final reduction $\mathcal{J} : \mathbb{R}^{n_\mathbf{x}} \to \mathbb{R} \in C^1$ that reduces the state $\mathbf{x}$ to a scalar, such that the adjoint model only needs to be run once to get a full gradient of $\mathcal{J}$ with respect to $\mathbf{x}$ and $\mathbf{p}$. A graphical representation of the vector evolution, as well as the corresponding derivatives, are given in Figure 2.

3.1 Reversal of serial evolution

We define the augmented forward run $\hat{f}$ of a function f as a function evaluation, which produces the same outputs, but caches all information required for the adjoint data flow reversal.

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_29



$$
\begin{aligned}
\frac{\mathrm{d}\mathcal{J}}{\mathrm{d}\mathbf{p}} &= \frac{\partial \mathcal{J}}{\partial \mathbf{p}} \\
&\quad + \frac{\partial \mathcal{J}}{\partial \mathbf{x}^3} \cdot \frac{\partial f^3}{\partial \mathbf{p}} \\
&\quad + \frac{\partial \mathcal{J}}{\partial \mathbf{x}^3} \cdot \frac{\partial f^3}{\partial \mathbf{x}^2} \cdot \frac{\partial f^2}{\partial \mathbf{p}} \\
&\quad + \frac{\partial \mathcal{J}}{\partial \mathbf{x}^3} \cdot \frac{\partial f^3}{\partial \mathbf{x}^2} \cdot \frac{\partial f^2}{\partial \mathbf{x}^1} \cdot \frac{\partial f^1}{\partial \mathbf{p}} \\
&= \frac{\partial \mathcal{J}}{\partial \mathbf{p}} + \frac{\partial \mathcal{J}}{\partial \mathbf{x}^3} \left(\frac{\partial f^3}{\partial \mathbf{p}} + \frac{\partial f^3}{\partial \mathbf{x}^2} \left(\frac{\partial f^2}{\partial \mathbf{p}} + \frac{\partial f^2}{\partial \mathbf{x}^1} \cdot \frac{\partial f^1}{\partial \mathbf{p}} \right) \right) \\
\frac{\mathrm{d}\mathcal{J}}{\mathrm{d}\mathbf{x}^0} &= \frac{\partial \mathcal{J}}{\partial \mathbf{x}^3} \cdot \frac{\partial f^3}{\partial \mathbf{x}^2} \cdot \frac{\partial f^2}{\partial \mathbf{x}^1} \cdot \frac{\partial f^1}{\partial \mathbf{x}^0}
\end{aligned}
$$

Fig. 2. Directed acyclic graph (DAG) representation of a three-step evolution (left), and the total derivative of J w.r.t. $\mathbf{p}$ (right). Note how the derivative can be found by multiplying the partial derivatives along paths from $\mathbf{p}$ to J, adding up parallel paths.

Conceptually, the adjoint reversal of a (scalar) evolution loop always consists of the following steps: The evolutionary loop, where all variables which are required for the adjoint reversal are cached

$$x^n = \hat{f}^n(x^{n-1}, p) \circ \hat{f}^{n-1}(x^{n-2}, p) \circ \ldots \circ \hat{f}^2(x^1, p) \circ \hat{f}^1(x^0, p)$$

is followed by the calculation of adjoints of the inputs $\bar{x}^0$, $\bar{p}$, as well as all intermediate adjoints $\bar{x}^i$ (either implicitly using an AD tool, or by explicitly calculating the required gradients):

$$\bar{x}^0 = \frac{\mathrm{d}}{\mathrm{d}x} \left[f^n(x^{n-1}, p) \circ f^{n-1}(x^{n-2}, p) \circ \ldots \circ f^1(x^0, p) \right] \cdot \bar{x}^n$$

as well as the adjoint of parameters p:

$$\bar{p} = \frac{\mathrm{d}}{\mathrm{d}p} \left[f^n(x^{n-1}, p) \circ f^{n-1}(x^{n-2}, p) \circ \ldots \circ f^1(x^0, p) \right] \cdot \bar{x}^n \quad .$$

An example code on how to implement such a reversal using handwritten adjoints is given in the code on GitHub. For a general-purpose AD tool the workflow is always similar, where the adjoint statements are built automatically during the second step:

1. Mark initial values x^0 and parameters p as active inputs.
2. Execute augmented forward section for full n-step evolution, storing required intermediate values.
3. Seed the outputs, e.g. final state x^n.
4. Propagate adjoints back from $\bar{x}^n$ to $\bar{x}^0$ and $\bar{p}$.
5. Harvest adjoints $\bar{x}^0$ and $\bar{p}$.

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_29

4 Parallel Adjoints of Evolution

Building on the serial reversal procedure, we now expand it to a parallel setting. For notational simplicity, we focus on scalar states and parameters, however, the procedure can be applied to vector-valued states and parameters virtually unchanged.

4.1 Parallel Reversal Procedure

To transition from a serial evolution to a parallel adjoint evolution, we define an evolutionary loop where the first $n-1$ iterations are calculated in passive mode and only the last $\hat{f}^n$ is calculated in augmented forward mode:

$$x^n = \hat{f}^n(x^{n-1}, p) \circ f^{n-1}(x^{n-2}, p) \circ \ldots \circ f^2(x^1, p) \circ f^1(x^0, p)$$

As only one iteration step is cached, *only* the following adjoints can be calculated:

$$\bar{x}^{n-1} = \frac{\mathrm{d}f(x^{n-1}, p)}{\mathrm{d}x^{n-1}} \cdot \bar{x}^n$$

as well as a partial adjoint update for p:

$$\bar{p}^n = \frac{\mathrm{d}}{\mathrm{d}p} f(x^{n-1}, p) \cdot \bar{x}^n$$

where, due to the incremental nature[11] of the adjoint:

$$\bar{p} = \sum_{i=0}^{n} \bar{p}^i \quad .$$

For the following discussion of parallel adjoint propagation, we assume that the n iteration steps can be distributed onto n processors. If less processors are available ($P < n$), one can, without loss of generality (for evenly divisible n), redistribute the loop iterations, such that each processor reverses n/n_p steps of the iterations.

As before, each processor calculates only one iteration step $\hat{f}_i$ in augmented primal mode, such that it can later calculate the adjoint reversal $\bar{\mathbf{x}}^{i-1} = \bar{\mathbf{x}}^i \cdot \frac{\partial f^i}{\partial \mathbf{x}^{i-1}}$. If we have n processors, then all steps can be reversed. However, all processors, except the last, have to wait for adjoint $\bar{\mathbf{x}}^i$ to be calculated and sent to it by processor $(i+1)$. Thus, the reverse propagation phase remains a serial operation, as the propagation of the next iteration can not start until the current one has finished. All gains come from the fact that each processor has to perform only one active forward evaluation and can rely on faster passive evaluations for the remaining iterations up to the active one. Each processor except the first has to perform some amount of redundant passive computation (f^0 to f^{i-1}) before it can execute $\hat{f}^i$. A graphical representation of the procedure is given in Figure 3.

In the following we outline the procedure for each processor i:

1. Execute passive evolution up to step $(i-1)$:
   $x^{i-1} = f^{i-1}(x^{i-2}, p) \circ \ldots \circ f^2(x^1, p) \circ f^1(x^0, p)$;

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_29

2. Execute augmented forward section $x^i = \hat{f}^i(x^{i-1}, p)$;
3. Seed outputs $\bar{x}^i$ (received from processor $(i+1)$ if $i < n$, else $\bar{x}^n = 1$);
4. Propagate adjoints back from $\bar{x}^i$ to $\bar{x}^{i-1}$ and $\bar{p}^i$;
5. Harvest adjoints $\bar{x}^{i-1}$ and $\bar{p}$;
6. Send calculated adjoints $\bar{x}^{i-1}$ to processor $(i-1)$ (if $i > 1$);
7. Reduce partial parameter adjoints $\bar{p} = \sum_{i=0}^{n} \bar{p}^i$ (e.g. MPI Allreduce).

A possible implementation for the parallel adjoint reversal of the previous evolution is available on GitHub.

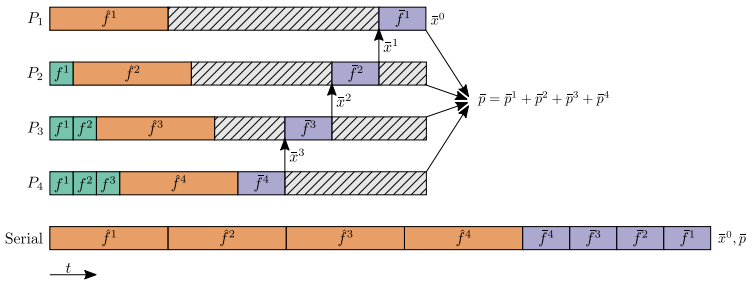


Fig. 3. Illustration of serial and parallel adjoint reversal for $n = 4, \lambda = 5, \alpha = 2$. Adjoints for the state x are passed along from processor to processor, as necessary requirements for the further reversal of the evolution. Adjoints for parameter p can be calculated thread local and are summed up with an `MPI_Allreduce` operation at the end. For n=4 Equation (4) predicts a speedup of $28/18 \approx 1.55$ and an upper limit for the speedup of $7/3 \approx 2.33$.

4.2 Run time analysis

For the following analysis of the run time of the algorithm, we assume the following:

- MPI communication cost (Send, Receive, Allreduce) is negligible compared to computation cost;
- Cost of each *active* function evaluations $\hat{f}^i$ is constant: $\mathit{cost}(\hat{f}^i) = \mathit{cost}(\hat{f}) =: c_{\hat{f}}$;
- Cost of each *passive* function evaluations f^i is constant: $\mathit{cost}(f^i) = \mathit{cost}(f) =: c_f$;
- Active (augmented forward) evaluation is *slower* than passive by a factor of $\lambda > 1$: $c_{\hat{f}} = \lambda c_f$;
- Adjoint propagation has cost $c_{\bar{f}}$, with a run time factor $\alpha > 0$: $c_{\bar{f}} = \alpha c_f$.

With the above assumptions we obtain the following costs:

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_29

– Total serial execution cost (n augmented forward evaluations and consecutive reverse propagations):

$$T_1 = n \cdot c_{\hat{f}} + n \cdot c_{\bar{f}} = n \cdot (c_{\hat{f}} + c_{\bar{f}}) = n \cdot (\lambda + \alpha) \cdot c_f$$

– Total parallel execution cost ($n-1$ passive evaluations, one augmented forward evaluations, and n reverse propagations)

$$T_n = (n-1) \cdot c_f + 1 \cdot c_{\hat{f}} + n \cdot c_{\bar{f}} = (n + \lambda + \alpha n - 1) \cdot c_f$$

With T_1 and T_n we can calculate the possible speedup (for $n \to \infty$) as:

$$S_\infty = \frac{\lim_{n\to\infty} T_1}{\lim_{n\to\infty} T_n} = \lim_{n\to\infty} \frac{\lambda + \alpha}{1 + \alpha + \frac{\lambda - 1}{n}} = \frac{\lambda + \alpha}{1 + \alpha} \tag{4}$$

Looking at the derived formula, we can see, that this approach benefits from shifting as much work as possible from the reverse propagation phase to the augmented forward phase (decreasing α, increasing λ). Operator-overloading AD tools are especially suited for this approach, as they have a higher factor between passive and augmented forward execution, compared to source-to-source tools. All except one processor have some amount of downtime during their blocking receive. This time can potentially be used to run further optimizations on the recorded data. An additional benefit of the parallel taping approach is the reduction of (per processor) peak memory required for the caching of the augmented forward section (only one instead of n steps have to be held in memory). While the overall memory consumption (for the cache) stays the same, using MPI it can be straightforwardly spread over multiple compute nodes.

5 Case Study: Lorenz Attractor

To demonstrate the feasibility of the proposed method we model a simple Lorenz attractor[18] using explicit forward Euler time integration of the form:

$$\mathbf{x}^{i+1} = \mathbf{x}^i + \Delta t \begin{pmatrix} \sigma * (x_1^i - x_0^i) \\ x_0^i * (\rho - x_2^i) - x_1^i \\ x_0^i * x_1^i - \beta * x_2^i \end{pmatrix} \quad .$$

Thus, we have state $\mathbf{x} = (x_0, x_1, x_2) \in \mathbb{R}^3$ and parameters $\mathbf{p} = (\beta, \rho, \sigma) \in \mathbb{R}^3$. We run the iteration with a small time-step Δt for $2 \cdot 10^7$ iterations. In terms of floating point operations, this computation is not very demanding, as only multiplications and subtractions are executed.
The iteration is implemented in `C++` and differentiation is handled by the operator-overloading AD tool `dco/c++`. We obtain a scalar output by evaluating the distance of the final iterate from the origin: $\mathcal{J} = \sqrt{x_0^2 + x_1^2 + x_2^2}$. The gradient $\frac{\mathrm{d}J}{\mathrm{d}\mathbf{p}}$ is then calculated by one evaluation of the adjoint model.

Timing the execution on 8 of 36 cores of an *Intel Xeon E5-2695 v4*[4] system we get the following run-time factors (as defined in Section 4):

Augmented forward to passive ratio: $\lambda = 6.83$

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_29

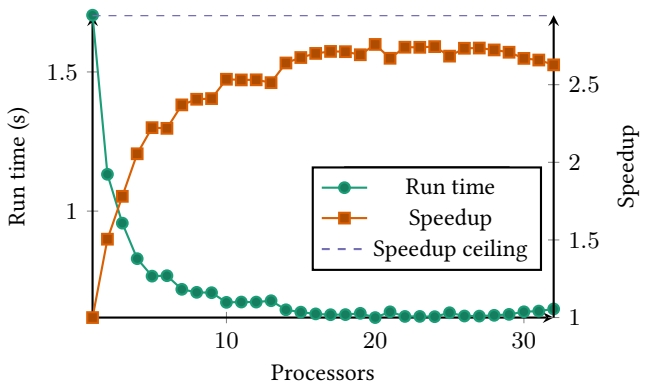


Fig. 4. Run time of $2 \cdot 10^7$ iterations of the (adjoint) Lorenz attractor (left axis) and speedup compared to serial execution (right axis). The limit predicted by Eq. 4 is marked as a dashed line. Run time average of 20 executions.

Adjoint propagation to passive ratio: $\alpha = 2.05$

Thus, Equation (4) limits the theoretical speedup to approximately three. For $P = 20$ we see an actual speedup of 2.75. The run time and speedup for up to 32 processors are presented in Figure 4. After a certain number of processors no more improvement is achieved and even a slight slowdown is observed. As the problem dimension is not scaled with the number of processors, the amount of (traced) work per processor shrinks, making communication and memory allocation overheads more visible.

The time the MPI parallel program spends in the different phases of the computation is outlined in Figure 5. All processors, except the last, spend time waiting for incoming adjoints from the next processor (gray parts of the bars). This time can conceivably be used for productive waiting, e.g. optimizing the recorded tape, such that once the reverse propagation starts (green part) it is executed faster. In the current implementation, passive calculations of the same iteration are performed on multiple processors (f_0 on all but the first, f_1 on all but the first and second, and so on). The passive computations could be bundled on a single processor, at the expense of additional communication and some further implementation work (processors need to be able to advance their state forward without actually performing the iteration). The resulting code would not necessarily be faster, but certainly more energy efficient (even more if idle processors are used in some other way or are allowed to enter a low power state). The approach of utilizing a dedicated process for all passive calculations is used in the following example.

6 Case Study: Parallel Taping in OpenFOAM

Here we present the reversal of evolutions as encountered in OpenFOAM[22]. OpenFOAM is an open-source computational fluid dynamics solver suite, based on the finite

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_29

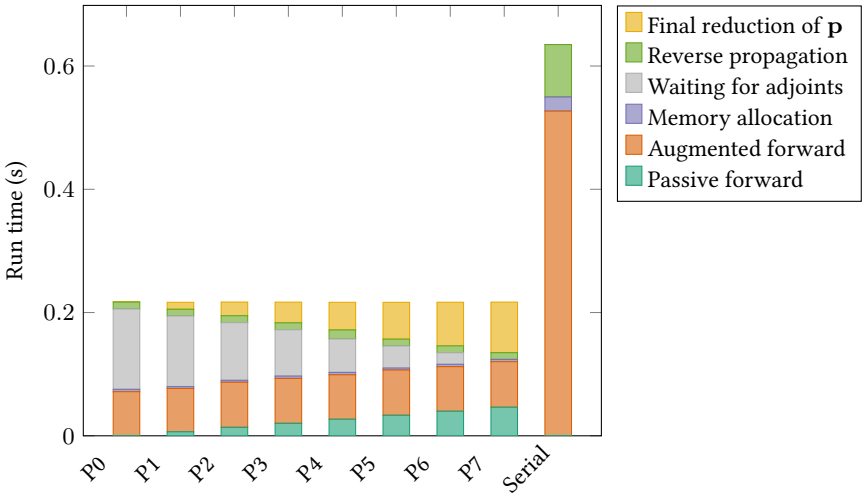


Fig. 5. Time spent in different phases of the computation for eight processors. For reference also the serial execution without any MPI communication is given on the right. Reverse interpretation starts on P_7 and finishes on P_0.

volume method. A version of OpenFOAM augmented with features of algorithmic differentiation has been developed by the author's institution[29].

For this case study, we use a variation of the `simpleFoam` solver, which implements the SIMPLE algorithm[24] with added penalty parameters to facilitate topology optimization[23]. The solver is run for a fixed number of time steps, all of which are traced and reversed by AD (except for linear solver calls, which are differentiated symbolically[30]).

Two different versions of the OpenFOAM solver, implementing the (MPI-) parallel taping approach, were developed[14]:

1. A version that operates with AD data types throughout (even if a time step does not need to be reversed, the AD data types are retained, but the tape recording is switched off).
2. A version where the time steps required to kick-start the adjoint recording are calculated on a different thread, running a different set of binaries.

Due to architectural limitations within OpenFOAM, we can currently not instantiate the full codebase with different floating-point types at the same time (e.g. with a passive floating point type `double` and an adjoint AD type `ad::adjoint_t<double>`). Using the first approach, the execution time for evaluating a single SIMPLE iteration step is not as fast as with fully passive (e.g. double) data types. That is, the observed run time factor lies somewhere in between c_f and $c_{\hat{f}}$.

However, it is possible to compile different sets of binaries for different types. In the second implementation, we use this to calculate all time steps on a single processor running a binary with true passive `double` data types. This data is used to kick-start

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_29

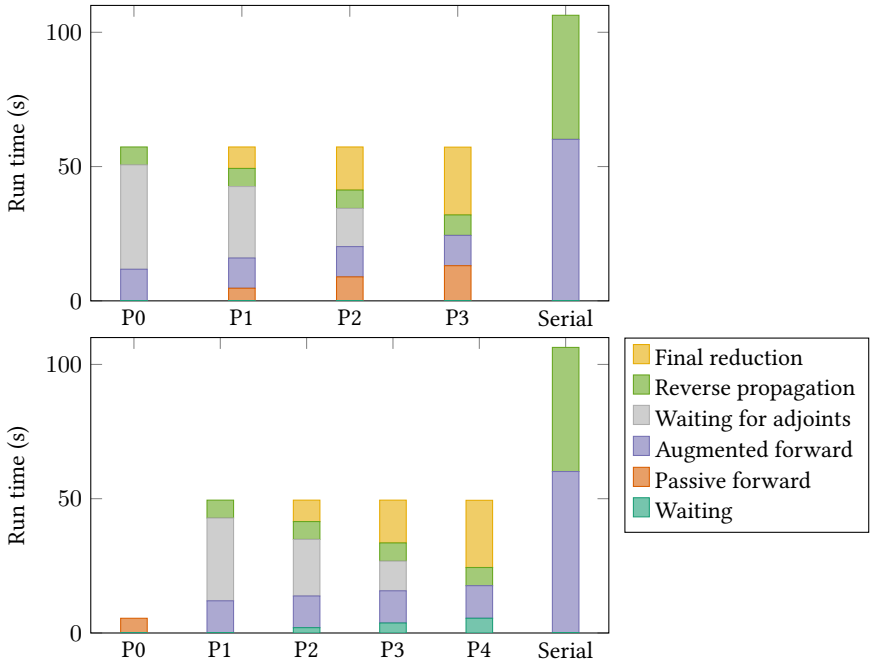


Fig. 6. Run time (in s) for a run of code variant 1.) on 4 nodes (top) and a run of code variant 2.) on 4+1 nodes (bottom). Run times are averaged over 10 runs.

the calculation of augmented forward and reverse propagation phases on the remaining processes, which are running the version of OpenFOAM instantiated with adjoint data types. As most processors have idle time after the recording of the tape, it is possible to move the extra process calculating with true double data types onto a processor with such idle time, eliminating the need for an extra processor.

On the investigated hardware (Intel Skylake Platinum 8160), the passive run time is about $\lambda_1 = 2.5$ times lower than the active run time (4.4s vs 11.3s per 64 time steps). With an observed interpretation factor of $\alpha_1 = 1.54$ we get the scaling limit according to Equation (4) as $S_1 \approx 1.6$. As expected, for version 2 the factor $\lambda_2 = 6.0$ is higher (1.86s vs 11.3s). With an observed interpretation factor of $\alpha_2 = 3.6$ we get the scaling limit according to Equation (4) as $S_2 \approx 2.1$.

Figure 6 shows an example run with AD tool `dco/c++` on the common pitzDaily[26] OpenFOAM case (flow over a backward-facing step). The case is run and adjoined over 256 time steps, obtaining the gradient w.r.t. roughly 50 000 design parameters. Here we distribute the calculation over four processors, thus each processor is responsible for recording and adjoining 64 time steps. We observe, that each recorded time step requires about 400 MB of memory. The state that needs to be communicated by MPI is just about 600 kB big, which is way below the interconnect capacity on the used system to have any non-negligible influence on the run time.

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_29

The observed adjoint recording factors are lower because a significant amount of run time is spent in the solution of linear equation systems, which for efficiency are differentiated symbolically, which removes complexity from the recording phase and the solution of additional systems of linear equations to the interpretation phase[31], effectively lowering the factor λ and increasing α.

7 Summary & Outlook

We demonstrated the feasibility of the approach to scalar and low-dimensional evolutions (lorenz attractor), as well as high dimensional ones (OpenFOAM). With operator-overloading AD tools, typical achievable speedups lie between two and three. To achieve scaling which extends to larger numbers of threads, the approach should be combined with parallelization of the underlying problem. This has the added benefit, that the parallel taping approach still gives benefits if primal and adjoint scaling by domain decomposition has stopped. The proposed method is not limited to first derivatives but is also applicable to all higher derivatives schemes, that at least contain one instance of the adjoint model (e.g. tangent over adjoint mode for obtaining second derivatives). While currently implemented as a manual approach, this technique could also be integrated into the AD tool itself, also removing the limitation to evolutions.

8 Acknowledgements

The authors would like to thank the reviewers for their detailed comments, which we hope we addressed adequately. OpenFOAM simulations were performed with computing resources granted by RWTH Aachen University under project thes1094.

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_29

References

1. M. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells. The FEniCS project version 1.5. *Archive of Numerical Software*, 3(100):9–23, 2015.
2. C. H. Bischof, H. M. Bücker, and A. Vehreschild. A macro language for derivative definition in ADiMat. In *Automatic Differentiation: Applications, Theory, and Implementations*, pages 181–188. Springer, 2006.
3. J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2022.
4. I. Corporation. Intel xeon prozessor e5-2695 v4, 2022. `https://ark.intel.com/content/www/de/de/ark/products/91316/intel-xeon-processor-e52695-v4-45m-cache-2-10-ghz.html`.
5. P. E. Farrell, D. A. Ham, S. W. Funke, and M. E. Rognes. Automated derivation of the adjoint of high-level transient finite element programs. *SIAM Journal on Scientific Computing*, 35(4):C369–C393, 2013.
6. K. Fischer et al. Diffractor - next generation ad, 2022. `https://github.com/JuliaDiff/Diffractor.jl`.
7. P. F. Fischer. Scaling limits for pde-based simulation. In *22nd AIAA Computational Fluid Dynamics Conference*, page 3049, 2015.
8. M. Förster. *Algorithmic Differentiation of Pragma-Defined Parallel Regions: Differentiating Computer Programs Containing OpenMP*. Springer, 2014.
9. Google LLC. Tangent: Source-to-source debuggable derivatives, 2017. `https://research.googleblog.com/2017/11/tangent-source-to-source-debuggable.html`.
10. A. Griewank, D. Juedes, and J. Utke. Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software*, 22(2):131–167, 1996.
11. A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2008.
12. L. Hascoët and V. Pascual. The Tapenade automatic differentiation tool: Principles, model, and specification. *ACM Transactions on Mathematical Software*, 39(3):20:1–20:43, 2013.
13. M. Innes. Don't unroll adjoint: Differentiating ssa-form programs. *CoRR*, abs/1810.07951, 2018.
14. J. Kremer. *Parallel adjoint taping approaches with OpenFOAM*. Bachelors Thesis, RWTH Aachen University, 2022.
15. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.
16. K. Leppkes, J. Lotz, and U. Naumann. Derivative Code by Overloading in C++ (`dco/c++`): Introduction and Summary of Features. Technical Report AIB-2016-08, RWTH Aachen University, Sept. 2016.
17. A. Logg and G. N. Wells. DOLFIN: Automated finite element computing. *ACM Transactions on Mathematical Software (TOMS)*, 37(2):20, 2010.
18. E. N. Lorenz. Deterministic nonperiodic flow. *Journal of atmospheric sciences*, 20(2):130–141, 1963.
19. F. Luporini, M. Louboutin, M. Lange, N. Kukreja, P. Witte, J. Hückelheim, C. Yount, P. H. Kelly, F. J. Herrmann, and G. J. Gorman. Architecture and performance of devito, a system for automated stencil computation. *ACM Transactions on Mathematical Software (TOMS)*, 46(1):1–28, 2020.

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_29

20. W. S. Moses, V. Churavy, L. Paehler, J. Hückelheim, S. H. K. Narayanan, M. Schanen, and J. Doerfert. 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*, pages 1–16, 2021.
21. U. Naumann. *The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation.* SIAM, 2012.
22. OpenFOAM Ltd. OpenFOAM - The Open Source Computational Fluid Dynamics (CFD) Toolbox. `http://openfoam.com/`.
23. C. Othmer, E. de Villiers, and H. G. Weller. Implementation of a continuous adjoint for topology optimization of ducted flows. In *18th AIAA Computational Fluid Dynamics Conference, June*, 2007.
24. S. V. Patankar and D. Spalding. A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows. *Int. J. of Heat and Mass Transfer*, 15(10):1787–1806, 1972.
25. E. T. Phipps, R. A. Bartlett, D. M. Gay, and R. J. Hoekstra. Large-scale transient sensitivity analysis of a radiation-damaged bipolar junction transistor via automatic differentiation. In C. H. Bischof, H. M. Bücker, P. D. Hovland, U. Naumann, and J. Utke, editors, *Advances in Automatic Differentiation*, pages 351–362. Springer, 2008.
26. R. W. Pitz and J. W. Daily. Combustion in a turbulent mixing layer formed at a rearward-facing step. *AIAA Journal*, 21(11):1565–1570, 1983.
27. M. Sagebaum, T. Albring, and N. R. Gauger. High-performance derivative computations using codipack. *ACM Transactions on Mathematical Software (TOMS)*, 45(4):1–26, 2019.
28. M. Schanen. *Semantics Driven Adjoints of the Message Passing Interface.* Dissertation, RWTH Aachen University, 2014.
29. M. Towara. *Discrete adjoint optimization with OpenFOAM.* Dissertation, RWTH Aachen University, Aachen, 2018.
30. M. Towara and U. Naumann. Simple adjoint message passing. *Optimization Methods and Software*, pages 1–18, 2018.
31. M. Towara, M. Schanen, and U. Naumann. MPI-parallel discrete adjoint OpenFOAM. *Procedia Computer Science*, 51:19 – 28, 2015. 2015 International Conference On Computational Science.
32. J. Utke, A. Lyons, and U. Naumann. Efficient reversal of the intraprocedural flow of control in adjoint computations. *Journal of Systems and Software*, 79(9):1280–1294, 2006. Selected papers from the fourth Source Code Analysis and Manipulation (SCAM 2004) Workshop.
33. A. Walther and A. Griewank. Getting started with Adol-C. *U. Naumann and O. Schenk, Combinatorial Scientific Computing, Chapman-Hall CRC Computational Science*, pages 181–202, 2012.

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_29