Discrete adjoints on many cores

Algorithmic differentiation of accelerated fluid simulations

by

Jan Christian Hückelheim

School of Engineering and Materials Science

Submitted in partial fulfilment of the requirements
of the Degree of Doctor of Philosophy.

DECEMBER 2016

Abstract

Simulations are used in science and industry to predict the performance of technical systems. Adjoint derivatives of these simulations can reveal the sensitivity of the system performance to changes in design or operating conditions, and are increasingly used in shape optimisation and uncertainty quantification. Algorithmic differentiation (AD) by source-transformation is an efficient method to compute such derivatives.

AD requires an analysis of the computation and its data flow to produce efficient adjoint code. One important step is the activity analysis that detects operations that need to be differentiated. An improved activity analysis is investigated in this thesis that simplifies build procedures for certain adjoint programs, and is demonstrated to improve the speed of an adjoint fluid dynamics solver. The method works by allowing a context-dependent analysis of routines.

The ongoing trend towards multi- and many-core architectures such as the Intel XeonPhi is creating challenges for AD. Two novel approaches are presented that replicate the parallelisation of a program in its corresponding adjoint program. The first approach detects loops that naturally result in a parallelisable adjoint loop, while the second approach uses loop transformation and the aforementioned context-dependent analysis to enforce parallelisable data access in the adjoint loop. A case study shows that both approaches yield adjoints that are as scalable as their underlying primal programs.

Adjoint computations are limited by their memory footprint, particularly in unsteady simulations, for which this work presents incomplete checkpointing as a method to reduce memory usage at the cost of a slight reduction in accuracy.

Finally, convergence of iterative linear solvers is discussed, which is especially relevant on accelerator cards, where single precision floating point numbers are frequently used and the choice of solvers is limited by the small memory size. Some problems that are particular to adjoint computations are discussed.

Statement of Originality

I, Jan Christian Hückelheim, confirm that the research included within this thesis is my own work or that where it has been carried out in collaboration with, or supported by others, that this is duly acknowledged below and my contribution indicated. Previously published material is also acknowledged below. I attest that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge break any UK law, infringe any third party's copyright or other Intellectual Property Right, or contain any confidential material. I accept that the College has the right to use plagiarism detection software to check the electronic version of the thesis. I confirm that this thesis has not been previously submitted for the award of a degree by this or any other university. The copyright of this thesis rests with the author and no quotation from it or information derived from it may be published without the prior written consent of the author.

Signature: Date: December 2016

Details of collaboration and publications:
Chapter 4 contains material that was accepted for publication in a paper entitled 'Algorithmic differentiation of code with multiple context-specific activities', first-authored by me. The coauthors Laurent Hascoët and Jens-Dominik Müller contributed through modifications in the inference rules, improvements to implementation and concept.
Chapter 6 contains material that was previously presented [83].
Chapter 5 contains material that was submitted for publication in a paper entitled 'Reverse-mode algorithmic differentiation of an OpenMP-parallel compressible flow solver', first-authored by me. The coauthors Paul Hovland, Michelle Strout and Jens-Dominik Müller made contributions to the concept, defining the applicability, and showing the correctness of the method.

Acknowledgements

I would like to express my deepest gratitude to everyone who made this thesis happen. This certainly includes Jens-Dominik Müller, my research supervisor who never failed to inspire me with ideas, and gave me the freedom and courage to choose my own direction whenever I wanted to.

I had the honour to work with very supportive and highly knowledgable people in these years, whom I can not thank enough. I am especially grateful to Laurent Hascoët and Paul Hovland for the fruitful discussions, their guidance and constructive feedback on my work, and for hosting me for several months at INRIA and Argonne National Laboratory. My colleagues at Queen Mary University of London were a pleasure to work with, and I am indebted to Mateusz Gugała for taking on the task of keeping our flow solver running smoothly, and for the productive conversations and coffee breaks.

This thesis would not have been possible without my friends and family, who managed to keep me sane throughout this process (or so I hope). Most importantly, my father Klaus, whose help was never more than a phone call away, my brother Arne, who came to visit me in London more often than I can count, and Shirlito, who was always by my side.

None of this research would have happened without the funding, training, networking opportunities and open borders provided by the European Union. Finally, I would like to thank Bruce Christianson and Shaun Forth for their detailed review and constructive suggestions.

Most of this work has been conducted within the About Flow project on “Adjoint-based optimisation of industrial and unsteady flows”. The project has received funding from the European Union’s Seventh Framework Programme for research, technological development, and demonstration under grant agreement no 317006.

The material in Chapter 5 and Chapter 7 is also based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics and Computer Science programs under contract number DE-AC02-06CH11357.

This research utilised Queen Mary University of London’s MidPlus computational facilities, supported by QMUL Research-IT and funded by EPSRC grant EP/K000128/1.

Contents

CHAPTER 1

Introduction

Simulations are sometimes called 'the third pillar of science', bridging the gap between theoretical models and practical experiments. Simulations can still be feasible when experiments are too costly, time consuming, or risky. On the other hand, even if a problem is too complex for a theoretical analysis, simulations can increase our understanding of a system in ways that an experiment can not.

This is especially true if the simulation is combined with adjoint differentiation, which not only predicts the behaviour of the system, but also produces derivatives that identify the most influential aspects of the system's design and operating conditions.

1.1 Adjoint derivatives

Adjoint derivatives serve as a crucial ingredient for methods as diverse as gradient-based optimisation, uncertainty quantification, parameter identification, inverse modelling, and mesh smoothing. For example, adjoint derivatives of a computational fluid dynamics (CFD) simulation of an aircraft in flight can provide the derivatives of the aircraft's overall air resistance with respect to displacement of each of its surface points, which directly reveals how the shape can be changed to improve the aerodynamic efficiency of the aircraft. Adjoints gained popularity in optimal control [99, 101] before making their way into other application domains, including geophysics [131], finance [28, 90, 118], neural networks [43], structural mechanics [24, 125], CFD [61, 87], and multi-physics applications that combine several of these disciplines [108, 146].

The strategies to compute adjoints for discretised models fall broadly in the two categories *continuous* and *discrete*, which are equivalent in the limit of infinitely high resolution, but differ in practice [114]. Continuous adjoint methods are developed by adjoint differentiation of the underlying material model, followed by a discretisation of the resulting adjoint model. In contrast, discrete adjoint models are the result of an

adjoint differentiation of an already discretised material model. While the continuous approach has been used with good success [3, 89, 126], discrete adjoints are guaranteed to yield *consistent* results, a desirable property that is further explained in Section 2.2, and this work is solely concerned with discrete methods, although at least some of the presented results are applicable to discrete and continuous methods alike.

Discrete adjoint methods come in several flavours, which are mathematically equivalent in the limit of vanishing roundoff errors. One can differentiate the discretised adjoint flow equations, and then implement a program that solves these equations either by hand [110] or by using code generation [46]. Neither of this was attempted in this work. Instead, one can generate an adjoint solver from the source code of a CFD program, a process that can be made less error-prone and more convenient by using algorithmic differentiation (AD).

The differentiation of computer programs is almost as old [122] as the programmable computer itself [40]. Conceptually, computer programs are straightforward to differentiate, as they are nothing but a mathematical function, broken down into a sequence of simple operations such as additions, multiplications, or a handful of transcendental functions, whose derivatives are known. Given this sequence, one must simply use the chain rule of calculus to accumulate all their derivatives. If this is done in reverse, starting from the program result and going back to the program inputs, one obtains the adjoint derivative in a process called 'reverse differentiation'. The adjoint derivative can be used to reconstruct the Jacobian matrix of the underlying function, and is an efficient way to do so for programs with many inputs and few outputs.

As computers are becoming more powerful, the demand for more accurate simulations is growing along with the size and complexity of simulation programs [88]. This has transformed AD from a simple exercise in calculus into a challenging problem in computer science and software engineering, and has motivated the development of many AD tools. Some of them use the *operator overloading* capabilities of most modern programming languages [66, 104] to compute derivatives, which is straightforward, but often leads to excessive runtimes and memory footprints. Other AD tools transform a given input source code into a code that implements the corresponding adjoint program. Such *source-transformation* AD tools [14, 55, 74, 120] have a long history [150], and can produce adjoint programs with very competitive performance [113]. They are routinely used in some practical applications, despite the fact that they are restricted to few languages (or a subset of their language features).

Source transformation tools can improve the performance of derivative code significantly through an analysis of the input code, to determine properties such as data dependencies, intermediate results that must be stored or recomputed [73], or pointer destinations and aliasing [129]. *Activity analysis*, which is the process of identifying

operations and variables that need to be differentiated, is especially important to reduce the amount of derivative computations [45, 74, 96, 142]. One of the central results of this thesis is an improvement to this analysis.

1.2 Recent challenges

There is an ongoing trend towards massively parallel computations at low clock speeds, caused by a stall in achievable processor clock rates [135], as well as a growing concern for energy-efficient computing. This challenge is unlikely to disappear in the near future, as information technology is already surpassing air travel in its CO2 emissions and its electricity use is growing fast [48]. As simulation programs are adapted to new, more power-efficient multicore CPUs [102] or accelerator cards such as Intel XeonPhi [140] or Nvidia Tesla, differentiation techniques have to follow suit.

1.2.1 Too many cores

On modern processors such as the XeonPhi accelerator with 60 or more cores and 240 or more available threads, a program that does not utilise parallelism will neither use much of that processor's computational power, nor much of the available memory bandwidth. With the growing number of cores, work needs to be partitioned in ever-smaller portions that can be done in parallel, and synchronisation overheads are a growing concern [19]. OpenMP [35] is often used to create parallel programs for such a shared-memory parallel environment.

Research on parallel discrete adjoint computations using distributed memory and message passing has been ongoing for a long time [81, 119, 139, 157, 159], while differentiation techniques suitable for shared-memory programs are only recently becoming a research subject [49, 50, 56]. Reverse-differentiation causes a reversal of the data flow, which may cause race conditions in an adjoint code even if the primal computation is embarrassingly parallel. This is a particular problem for shared-memory-parallel code, where communication between threads is not always detectable at compile-time. It is thus not surprising that most previous results are restricted to the forward-mode [21–23], or result in a conservative parallelisation with poor performance or high memory footprint in many practical cases [49, 50, 56, 138].

At the same time, previous work presented a highly scalable hand-implementation of discrete adjoint solvers using CUDA [38], and scalable hand-coded continuous adjoint solvers [4]. Automatic generation of discrete adjoint solvers has been presented using the finite element method, albeit with an approach that is neither applicable to finite volume methods, nor any existing industrial code [46]. Nevertheless, these success stories show

that scalable, efficient adjoint solvers can be developed in principle. Methods to generate such scalable adjoint solvers using AD are another important result of this thesis.

1.2.2 The memory gap

Memory size, bandwidth, and latency of modern computers are improving more slowly than the computational power [174]. Accelerators take this trend to a new extreme. A XeonPhi offers 8 Gigabytes of memory in the currently most common configuration, an unimpressive number that is rivalled by consumer grade laptops and high-end smartphones. At the same time, the XeonPhi achieves one trillion (10^{12}) floating point operations per second in double precision, which is more than an order of magnitude above most CPUs on the market today. As a result, the memory footprint of a program is a particular concern on accelerators. To make matters worse, adjoint computations are notorious for their large memory footprint, as they need to store intermediate states of the original computation, so-called 'checkpoints', which are required during the reverse-accumulation of derivatives.

Reducing the memory footprint of adjoint codes, particularly when applied to time-dependent unsteady flow problems, has been a long-standing research problem. Previous work depends on the storage of checkpoints at selected time steps, combined with a recomputation of required program states from the nearest available checkpoint. Most famously, this was presented in the revolve-algorithm [67] for a previously known total number of time steps, and later for cases where the number of time steps is not known a-priori [169]. Other work has used checkpoint compression [154, 171]. All these methods have in common that they trade memory for computational cost. In contrast, a method developed in this thesis offers a tradeoff between memory footprint and accuracy.

1.2.3 Reliability and roundoff

Many CFD methods such as implicit schemes for compressible [17, 176] or semi-implicit schemes for incompressible flow [47, 163] require solving linear systems. The convergence of the popular CG linear solver is well understood for exact or floating point arithmetic [64, 123], and CG has been shown to remain accurate even if its intermediate steps are affected by large roundoff errors [151]. This property however does not extend to differentiated CG, as has been shown for practical problems [112]. Asymmetric systems require other methods such as BiCG, where roundoff is known to have a large influence [9], and which can even break down in the absence of roundoff errors [155] for certain systems. The lack of formal convergence guarantees is a particular problem for adjoint solvers, which are in practice often used as a black box with poor initial guesses and little understanding of the physical meaning of its intermediate values.

More stable iterative solvers, such as GMRES with a large restart number, have memory requirements that are prohibitive for accelerator cards. Some results in this thesis point out open problems related to the use of iterative solvers in adjoint computations.

1.3 Contributions

Most results in this thesis are meant as stepping stones towards the routine use of reverse-mode differentiation for shared-memory parallel code. They mostly reduce the time to solution and memory footprint and improve the parallel scalability of adjoint solvers generated by AD. While the contributions are relevant for all parallel computing architectures, they are of particular value on accelerators. The work makes use of the preexisting CFD and adjoint solver `stamps` [29], which is described in more detail in Section 2.1. Other adjoint flow solvers have been developed by various authors, notably the open-source adjoint solvers SU2 [128, 177] and OpenFoam [127, 156].

Chapter 4 discusses multi-activity differentiation, a refined activity analysis that allows the creation of specialised differentiated procedures for each call site of the original procedure. Such a strategy was mentioned in a previous paper [14], but its advantages have never been investigated, and the previous implementation was restricted to forward-mode AD. Since the detection of active and passive variables is the basis for many other analysis steps, a refined activity analysis can lead to significant savings in memory footprint and computational cost. In the course of this work, the method was implemented in the Tapenade [74] AD tool, and a case study shows performance improvements when the method is applied to the `stamps` CFD solver. Multi-activity differentiation can be beneficial for serial and parallel computations on most platforms, and is shown to be beneficial for the performance of `stamps`. In addition, the method has an interesting use case for shared-memory-parallel adjoint computations.

As mentioned above, the reverse-differentiation of parallel code results in a reversal of the data dependencies, which can preclude parallel execution of the generated adjoint code. To address this problem, Section 5.3 presents a novel method, referred to as *reflexive shared-memory parallelism* (RSMP). The method uses multi-activity differentiation to selectively compute certain partial derivatives of the body of a parallel loop, and a transformation technique that enforces the computation of these partial derivatives in an order that preserves the data access pattern of the primal loop. This allows the same parallelisation to be used in the adjoint and primal computations, and a case study shows that RSMP can generate adjoint code that achieves the same scalability as its underlying primal code, and outperforms code generated by previously published reverse-differentiation methods.

A closely related method, referred to as *symmetric shared-memory parallelism* (SSMP), is studied in the remaining parts of Chapter 5. Just like RSMP, the adjoint codes generated with SSMP preserve the memory access pattern and scalability of their underlying primal codes. However, SSMP does not require loop transformations, and instead detects parallel loops in the primal code that naturally yield a parallel adjoint loop. This works even for unstructured stencil computations, for which previous methods fail to generate efficient parallel adjoint code. SSMP is incorporated into the build procedure of `stamps`, and results in an efficient OpenMP-parallel adjoint CFD solver that scales as well as the primal code on a CPU and a XeonPhi accelerator card.

While SSMP and RSMP can generate scalable code for many-core architectures, the memory footprint of unsteady adjoint computations still precludes the use of these processors. This is addressed in Chapter 6, where a low-cost compression scheme is developed to reduce the checkpoint size. In contrast to the checkpointing or compression methods that were previously presented, the new scheme, referred to as *incomplete checkpointing*, does not significantly increase the computational effort, as it simply discards some data that can be reconstructed approximately using interpolation during the derivative computation. This is shown to result in large memory savings, while still maintaining an acceptable accuracy.

The restrictive memory size of accelerator cards also motivates further investigation into lower-precision computations and iterative linear solvers that use less memory. However, the findings in Chapter 7 show that the convergence of iterative linear solvers can be a particular problem for adjoint solvers. The chapter highlights some questions where future research is needed.
In summary, the contributions of this thesis are

1. the context-dependent activity analysis for reverse-differentiation in Chapter 4,
2. the methods to preserve the memory access pattern of parallel code in Chapter 5,
3. the temporal and spatial checkpoint coarsening in Chapter 6, and
4. the observations about the influence of transposition and right hand sides on iterative solvers in Chapter 7.

CHAPTER 2

Background

This chapter briefly introduces the simulation of compressible flows in computational fluid dynamics (CFD) in Section 2.1. Following that, Section 2.2 explains the differentiation of computer programs in general, and CFD codes in particular. A discussion on linear systems solvers and their differentiation in Section 2.3, followed by an overview of CFD-specific implementation and parallelisation techniques in Section 2.4 and Section 2.5, complete the chapter.

2.1 Simulating fluids with computer programs

There is to date no one-size-fits-all solution to CFD. Solution methods vary on all levels, from the physical models that they assume the flow to follow, through the discretisation, to their implementation. Section 2.1.1 describes laminar compressible flow, which is an important example of a physical model in CFD. The model needs to be discretised in some way to make it tractable for a digital computer, and the finite volume method shown in Section 2.1.2 is a popular choice for this. Finally, the discretisation needs to be implemented in a computer program, and `stamps` is presented as an example for this in Section 2.4.

2.1.1 Conservation equations for fluid flow

A mathematical model for the behaviour of fluid flow is given by conservation equations. The conservation of momentum for an arbitrary control volume Ω reads [17]

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_{\Omega}\rho\vec{v}\,\mathrm{d}\Omega+\int_{S}\rho\vec{v}(\vec{v}\cdot\vec{n})\mathrm{d}S=\int_{\Omega}\rho\vec{f}_{e}\,\mathrm{d}\Omega+\int_{S}\vec{\tau}\cdot\vec{n}\,\mathrm{d}S-\int_{S}p\cdot\vec{n}\,\mathrm{d}S, \tag{2.1}$$

and applies to compressible and incompressible flow alike. In this equation, ρ, p and $\vec{v}$ denote density, pressure and velocity, $\vec{f}_e$ contains external body forces like gravity or

electromagnetic forces for charged fluids, and $\vec{\tau}$ is the viscous stress tensor, which for Newtonian fluids is proportional to the shear rate of the flow, and has an additional dependence on the divergence field in the case of compressible fluids. The evaluation of $\vec{\tau}$ requires spatial gradients of the velocity field, whose computation will be discussed in Section 2.1.3.

Some CFD solvers artificially increase the viscous stress in places where a high amount of turbulence is predicted, using a model such as Spalart Allmaras, DES, or LES [44, 148, 149], and often produce usable results even if the spatial discretisation is too coarse to resolve turbulent flow.

The defining feature of compressible flows is the compression and expansion of the fluid. To model it, one needs to take into account the conversion between mechanical energy and heat, as well as heat transport, given by the energy equation

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_\Omega \rho E \,\mathrm{d}\Omega + \int_S \rho H(\vec{v}\cdot\vec{n})\mathrm{d}S = \int_S k_t(\nabla T\cdot\vec{n})\mathrm{d}S + \int_\Omega (\rho f_e\cdot\vec{v} + q_h)\mathrm{d}\Omega + \int_S (\vec{\tau}\cdot\vec{v})\cdot\vec{n}\,\mathrm{d}S, \quad (2.2)$$

which describes the conservation of energy E, with the total enthalpy H, thermal conductivity coefficient k_t, temperature T, and a volume heat source q_h, for example from radiation or chemical reactions. The model for compressible flow is completed by the conservation of mass, which reads

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_\Omega \rho \,\mathrm{d}\Omega + \int_S \rho(\vec{v}\cdot\vec{n})\mathrm{d}S = 0. \quad (2.3)$$

Although the conservation laws apply equally to incompressible flow, it is common for incompressible flow solvers to use a modified set of equations, where the conservation of mass is enforced by a pressure equation, and the conservation of energy is not modelled explicitly, since it is invariant. The properties of the equations are different enough for incompressible flow to require different solver schemes, with semi-implicit schemes that solve the momentum in each direction and the pressure separately being a popular choice [47]. Although the numerical experiments in this work are carried out with a compressible flow solver, the methods developed in this thesis are probably applicable to compressible and incompressible flow solvers alike.

2.1.2 Discretisation in space and time

The finite volume method (FV) is a popular way of discretising partial differential equations. In FV, the space is subdivided into non-overlapping elements, the collection of which is called *mesh*. This task is not trivial for practical applications, and is a research area of its own [153]. The flow field is stored as density, momentum in three directions, energy, and possibly a turbulence variable for each of the N control volumes

in a *state vector* $\vec{U} \in \mathbb{R}^{6N}$. The control volumes can be defined to be identical with the mesh elements. The `stamps` solver uses instead a dual cell approach, where a control volume is centred at the corners of all mesh elements. Unless otherwise specified, the terms *cell* and *element* are used to refer to control volumes throughout this work.

The volume and surface integrals for equations (2.1-2.3) are approximated in each cell. For the volume integrals, this could be as simple as taking the product of the element volume $\|\Omega\|$ and the state $\vec{U}$ at the midpoint x_ζ as

$$\int_\Omega \vec{U} \mathrm{d}\Omega \approx \|\Omega\| \cdot \vec{U}(x_\zeta).$$

More accurate quadrature formulas can be used to achieve higher order accuracy [144], and similar quadrature formulas can be used for element surfaces. Surface integrals for polyhedral cells can be computed as the sum over all faces S_k as

$$\int_S \vec{U} \mathrm{d}S = \sum_{S_k \in S} \int_{S_k} \vec{U} \mathrm{d}S,$$

where k is most often $1 \ldots 4$ for tetrahedral or $1 \ldots 6$ for hexahedral meshes. For each face, the integral can be approximated by the product of the value at the face midpoint x_k and the face area $\|S_k\|$ as

$$\int_{S_k} \vec{U} \mathrm{d}S \approx \|S_k\| \cdot \vec{U}(x_k),$$

and the function value at the midpoint of a face can be computed by interpolation from the element midpoint value and the value at the midpoint of the neighbour element x_{nk}

$$\vec{U}(x_k) \approx \frac{\|x_{nk} - x_k\|}{\|x_{nk} - x_\zeta\|} \cdot \vec{U}(x_{nk}) + \frac{\|x_k - x_\zeta\|}{\|x_{nk} - x_\zeta\|} \cdot \vec{U}(x_\zeta).$$

The linear interpolation leads to a stable, but inaccurate scheme. Better interpolation schemes have been developed, some of which also take into account the spatial gradient of the flow field [17], whose computation is shown in Section 2.1.3. Special care needs to be taken at boundaries, where no neighbour elements are present.

Regardless of the mesh type, quadrature formulas, interpolation schemes, or whether finite element, finite volume, finite difference or discontinuous Galerkin schemes were used, the discretisation of spatial variables transforms the partial differential equations into a system of ordinary differential equations.

The time is discretised by representing the solution only at selected points in time. The time-dependent terms in the conservation laws are approximated, for example with a second-order backward differencing formula (BDF2) given by

$$\frac{\partial \vec{U}}{\partial t} \approx \frac{\vec{U}_{t-2} - 4 \cdot \vec{U}_{t-1} + 3 \cdot \vec{U}_t}{2\Delta t}.$$

As with the spatial discretisation, boundaries require special care. On the time axis, this affects the first two time steps, for which some other scheme must be used. For all other time steps, BDF2 requires that the two previous time steps are stored in memory. Many other time discretisation schemes exist [25], but BDF2 is a popular choice as it is easy to implement, second-order accurate, and unconditionally stable [47].

If the space was discretised using N cells, the discretised conservation equations can be expressed in a function $F : \mathbb{R}^{6N} \to \mathbb{R}^{6N}$ such that a solution to the equations satisfies

$$F(\vec{U}) = 0,$$

or equivalently, the *residual vector*

$$\vec{R} := F(\vec{U}) \tag{2.4}$$

has norm zero if $\vec{U}$ is a solution. It is easy to see that the iterative scheme

$$\vec{U}^{\tilde{t}+1,t} = \vec{U}^{\tilde{t},t} + c \cdot F(\vec{U}^{\tilde{t},t}) \tag{2.5}$$

has a fixed point at the solution to the conservation equations. The scheme can in many cases be made stable by selecting a sufficiently small constant c. Alternatively, a Newton-like scheme given by

$$\vec{U}^{\tilde{t}+1,t} = \vec{U}^{\tilde{t},t} - (\nabla F(\vec{U}^{\tilde{t},t}))^{-1} \cdot F(\vec{U}^{\tilde{t},t}) \tag{2.6}$$

can often find the solution in fewer iterations, but requires a linearisation of F and the solution to a linear system. It is often preferable to use a *preconditioner* $\vec{\mathscr{P}} \approx \nabla F(\vec{U})$ as

$$\vec{U}^{\tilde{t}+1,t} = \vec{U}^{\tilde{t},t} - \vec{\mathscr{P}}^{-1} \cdot F(\vec{U}^{\tilde{t},t}) \tag{2.7}$$

and chose $\vec{\mathscr{P}}$ such that it resembles $\nabla F(\vec{U}^{\tilde{t}})$ enough to accelerate convergence, but is as sparse and easy to invert as possible. Good preconditioners must strike a balance between reducing the number of iterations, the cost of each iteration, and stability in a wide range of applications, possibly with poor initial guesses.

This work uses a *dual time stepping* scheme, where an *outer loop* computes one time step after another, for which each time step t corresponds to a point in physical time. For each time step, an *inner loop* implements the iterative process (2.7) that evolves the state vector in a series of pseudo time steps $\tilde{t}$ until an accurate solution has been found. The end result of this inner loop represents the actual flow state at that point in time, and enters the BDF2 term for the following two time steps. For readability, the pseudo time step will not be explicitly shown in the remainder of this work.

2.1.3 Spatial gradient calculation

Spatial gradients describe how a quantity changes with spatial location in a given flow field, and play an important role in CFD, for instance during high-order interpolation of flow properties from element centre points to face centre points, or for the evaluation of viscous stresses. One way of computing gradients is given by the Green-Gauss theorem, which states that for an element Ω, the volume integral over the gradient $\nabla\vec{U}$ is equal to the surface integral over $\vec{U}\cdot\vec{n}$:

$$\int_{\Omega}\nabla\vec{U}\,\mathrm{d}\Omega = \int_{\partial\Omega}\vec{U}\cdot\vec{n}\,\mathrm{d}S \tag{2.8}$$

Assuming that $\nabla\vec{U}$ is constant in Ω, the gradient can be written as

$$\nabla\vec{U} = \frac{1}{\|\Omega\|}\int_{\partial\Omega}\vec{U}\cdot\vec{n}\,\mathrm{d}S, \tag{2.9}$$

which can be computed using the surface integration techniques discussed in Section 2.1.2.

2.2 Differentiating computer programs

The differentiation of computer programs transforms a given program, referred to as the *primal program*, into a *derivative program* such that the result of the derivative program represents the mathematical derivative of the primal program outputs with respect to its inputs. This section gives a brief summary of this topic. For a more detailed discussion, the reader is referred to [65, 68, 117].

Not all mathematical functions are differentiable everywhere. Famous counter-examples are the square root or absolute value functions. There are even mathematical functions that are not differentiable anywhere, with the Dirichlet or Weierstrass function being notable examples. Indeed, differentiability requires that a function is continuous, making all integer functions non-differentiable. It is not surprising, then, that not all computer programs are differentiable either, and that the differentiation of a computer programs requires that this program implements some piecewise differentiable mathematical function.

There are a variety of methods in the literature that aim at computing derivatives of computer programs, as illustrated in Figure 2.1. The methods differ in accuracy (see Section 2.2.4), implementation effort, and runtime (see Section 2.2.1). A related method, called the *continuous adjoint method*, is not actually a method to differentiate computer programs, but rather a method to differentiate physical models. It competes with the discrete adjoint method used in this work, and successful applications of continuous adjoint solvers on accelerator cards have been achieved by other authors [4], but have certain drawbacks discussed in Section 2.2.4.

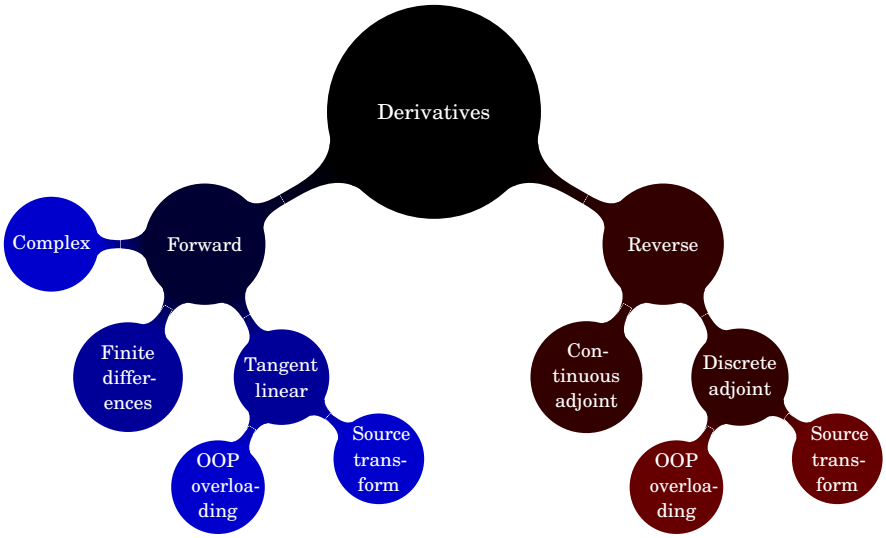


Figure 2.1: Many roads lead to derivatives. Forward and reverse methods have a fundamentally different runtime complexity, which is the subject of Section 2.2.1. Finite differences and tangent-linear derivatives have the same runtime, but different accuracies, and similarly, continuous and discrete adjoints have the same complexity, but differ in accuracy. This is discussed in Section 2.2.4. Within discrete methods, operator-overloading and source-transformation approaches differ in their implementation effort, and while both have the same asymptotic runtime complexity, their runtimes typically differ by some constant factor. These matters are discussed in Section 2.2.3.

2.2.1 Forward and reverse

Given a computer program with runtime T that implements

$$J = P(\vec{\alpha}), \quad J \in \mathbb{R}^m, \quad \vec{\alpha} \in \mathbb{R}^n, \tag{2.10}$$

a forward method creates a program, referred to as *tangent-linear program*, that computes the product of the Jacobian matrix ∇P with some seed vector $\dot{\vec{\alpha}} \in \mathbb{R}^n$ as

$$\dot{J} = (\nabla P(\vec{\alpha}))\dot{\vec{\alpha}}. \tag{2.11}$$

In contrast, a reverse method creates a program, referred to as *adjoint program* that computes the product of the transpose Jacobian matrix with some seed vector $\bar{J} \in \mathbb{R}^m$ as

$$\bar{\vec{\alpha}} = (\nabla P(\vec{\alpha}))^T \bar{J}. \tag{2.12}$$

In both cases, the evaluation of the matrix vector product is performed without computing or storing the full Jacobian matrix ∇P, and with a runtime of $\mathcal{O}(T)$. Forward methods can be implemented in various ways, for example using finite differences or algorithmic differentiation. Reverse methods can be implemented by using algorithmic

differentiation. Note that the word *gradient* is used in this work to refer to spatial gradients as described in gradient, whereas the result of differentiation of computer programs is referred to as a *derivative* throughout.

A user who is interested in obtaining ∇P can repeatedly call the tangent-linear program, using the cartesian basis vectors in $\mathbb{R}^n$ as seeds. This yields the complete Jacobian in a runtime of $\mathcal{O}(nT)$. Alternatively, by calling the adjoint program repeatedly with all cartesian basis vectors in $\mathbb{R}^m$, the Jacobian can be computed with runtime $\mathcal{O}(mT)$. Reverse methods can greatly reduce the computational cost if $m \ll n$. If the sparsity pattern of the Jacobian matrix is known, it is sometimes possible to compute multiple non-overlapping rows with a single call to the tangent-linear program, and similarly, to compute multiple non-overlapping columns with a single call to the adjoint program. This process is called *seeding* in the literature [68]. Finding an optimal seeding can be reduced to a graph colouring problem [161], and is therefore NP-hard [53]. Practical algorithms have been developed to find a sufficient approximation in a reasonable time [106, 107].

2.2.2 Finite-differences and complex variables

The Finite-differences method (FD) can be used if the source code of the primal program is not available. It is a forward method, and can compute an approximation to the Jacobian-vector-product as

$$(\nabla P) \cdot \dot{\vec{\alpha}} \approx \frac{P(\alpha + h\dot{\vec{\alpha}}) - P(\vec{\alpha})}{h}. \tag{2.13}$$

As with other forward methods, the computation of the full Jacobian matrix is infeasible with FD if the number of inputs is large. Another disadvantage is that a step size h has to be selected, where small step sizes increase the roundoff error and large step sizes cause a truncation error.

Some languages like Fortran offer the use of complex instead of real floating point numbers. For a function P that only uses real variables, one can create a derivative procedure P_c by replacing all real datatypes with complex datatypes. The Jacobian-vector-product can then be approximated as

$$(\nabla P) \cdot \dot{\vec{\alpha}} \approx \frac{1}{h} \mathrm{Im}\left(P_c(\vec{\alpha} + h\mathbf{i}\dot{\vec{\alpha}})\right). \tag{2.14}$$

Each evaluation is expensive, since the computations have to be carried out using complex variables. The method does not work if the primal computation already uses complex variables. If the step size h is chosen correctly, the derivatives are computed to machine precision. An overview of this method is shown in [109].

2.2.3 Algorithmic differentiation

Algorithmic differentiation (AD) is the process of differentiating computer programs following a prescribed recipe. Usually, this is at least partly automated through an AD tool. AD approaches broadly fall into the categories *operator overloading* and *source transformation*. Both can in principle be used in forward and reverse mode.

2.2.3.1 Source transformation

Source transformation AD works by interpreting the primal source code, and generating the corresponding tangent-linear or adjoint source code [117]. The process happens entirely at compile time, and can benefit from a global analysis of the program [142] that can help to remove unneeded computations that have no effect on the final derivative results. Some of this analysis is further explained in Section 2.2.3.3.

Source transformation in forward mode has a long history [173]. The reverse-mode is harder to implement and understand, with developments for simple programs without control flow [170] and for general programs with loops and branches [150] following later. Tools that use source-transformation have been developed for a variety of languages, including Fortran [56, 74, 162], C [16, 54, 74], Julia [134], Java [147], and MATLAB [15].

In forward mode, derivatives are traced through the program in the same direction as the original computation. A primal floating-point variable v is augmented with a tangent-linear variable $\dot{v}$. Every operation acting on the primal variable is augmented by a linearised operation that multiplies the tangent-linear variable with the partial derivative of the operation, evaluated at the current values of the primal inputs to that operation. All control flow, such as branches, loops, goto statements, or function calls, can remain the same as in the primal program.

In reverse mode, v is augmented with an adjoint variable $\bar{v}$, and the program is first executed normally, tracing the control flow path and intermediate variables. The partial derivatives of all operations are then multiplied with the adjoint variables of their outputs, in a process that progresses backwards from the outputs to the inputs. For code that does not contain branches, loops, function calls, or any other control flow, and does not reuse any variable, an implementation of this can work by reproducing the primal code, followed by code in which the partial derivatives of the primal operations are computed in reverse order. If variables are overwritten in the primal computation, values that will be needed to compute partial derivatives of nonlinear operations have to be stored, or recomputed from other available information. For code that contains control flow, it is necessary to store the path taken by the primal computation, such as branch decisions, or the number of loop iterations performed [121]. This can be done by adding calls to the primal code that push such information on a stack, and calls at the

	primal	tangent-linear	adjoint	
1	$w \leftarrow v_1 + v_2$	$\dot{w} \leftarrow \dot{v}_1 + \dot{v}_2$	$\bar{v}_1 \leftarrow \bar{v}_1 + \bar{w}$	$\bar{v}_2 \leftarrow \bar{v}_2 + \bar{w}$
2	$w \leftarrow v_1 - v_2$	$\dot{w} \leftarrow \dot{v}_1 - \dot{v}_2$	$\bar{v}_1 \leftarrow \bar{v}_1 + \bar{w}$	$\bar{v}_2 \leftarrow \bar{v}_2 - \bar{w}$
3	$w \leftarrow v_1 \cdot v_2$	$\dot{w} \leftarrow \dot{v}_1 \cdot v_2 + v_1 \cdot \dot{v}_2$	$\bar{v}_1 \leftarrow \bar{v}_1 + v_2 \cdot \bar{w}$	$\bar{v}_2 \leftarrow \bar{v}_2 + v_1 \cdot \bar{w}$
4	$w \leftarrow \sin(v)$	$\dot{w} \leftarrow \cos(v) \cdot \dot{v}$	$\bar{v} \leftarrow \bar{v} + \cos(v) \cdot \bar{w}$	
5	$w \leftarrow \cos(v)$	$\dot{w} \leftarrow -\sin(v) \cdot \dot{v}$	$\bar{v} \leftarrow \bar{v} - \sin(v) \cdot \bar{w}$	
6	$w \leftarrow \frac{v_1}{v_2}$	$\dot{w} \leftarrow \frac{\dot{v}_1 \cdot v_2 - v_1 \cdot \dot{v}_2}{v_2^2}$	$\bar{v}_1 \leftarrow \bar{v}_1 + \frac{\bar{w}}{v_2}$	$\bar{v}_2 \leftarrow \bar{v}_2 - v_1 \cdot \frac{\bar{w}}{v_2^2}$
7	$w \leftarrow \sqrt{(v)}$	$\dot{w} \leftarrow \frac{\dot{v}}{2\sqrt{(v)}}$	$\bar{v} \leftarrow \bar{v} + \frac{\bar{w}}{2\sqrt{(v)}}$	
8	$w \leftarrow \tan(v)$	$\dot{w} \leftarrow \dot{v} \cdot (1 + \tan(v)^2)$	$\bar{v} \leftarrow \bar{v} + (1 + \tan(v)^2) \cdot \bar{w}$	
9	$w \leftarrow \log(v)$	$\dot{w} \leftarrow \frac{\dot{v}}{v}$	$\bar{v} \leftarrow \bar{v} + \frac{\bar{w}}{v}$	
10	$w \leftarrow \mathrm{abs}(v)$	$\dot{w} \leftarrow \dot{v} \cdot \mathrm{sign}(v)$	$\bar{v} \leftarrow \bar{v} + \mathrm{sign}(v) \cdot \bar{w}$	

Table 2.1: Rules for source transformation of selected operators and intrinsic functions. The primals 3-10 are nonlinear, and their derivative computation thus requires that the primal value is available. The computations in examples 6-10 are only piecewise differentiable.

corresponding location in the adjoint code that pop this information from the stack when it is needed. This results in a *control flow reversal* during the adjoint computation [160].

Derivatives of single operations for forward and reverse mode are shown in Table 2.1. Apart from the straightforward control flow reversal and statement-wise differentiation, source-transformation AD tools employ a variety of methods to improve the efficiency of the generated derivative code, including elimination techniques on the computational graph [116], exploitation of independent computations [22, 72], checkpointing schemes [67, 169] and activity analysis [45, 74, 96, 142]. Checkpointing schemes result in a trade-off between memory usage and computational cost [33] and are discussed in more detail in Chapter 6. Activity analysis is further discussed in Section 2.2.3.3 and Chapter 4.

2.2.3.2 Operator overloading

AD can very elegantly be implemented using operator overloading, in languages that support it, including Python [166], C++ [66, 79], or C#/F#/.NET [11]. The declarations for primal floating point numbers are replaced with declarations for a datatype that stores primal and derivative values, and overloading the operators such that they still compute the primal values as normal, but at the same time, propagate the derivatives.

Reverse mode AD by operator overloading relies on operators that, as well as computing the primal function, create a tape that stores all intermediate variables and the operations performed on them. This tape is then interpreted in reverse to accumulate derivatives.

Operator overloading approaches are relatively easy to implement and relatively straightforward to use even for large code bases [2, 156], but the use of the taping mechanism inhibits most compiler optimisations and has a memory footprint that makes this method expensive to use on conventional computers and infeasible to use on accelerators. For these reasons, operator overloading approaches are not further discussed in this work.

2.2.3.3 Activity analysis

A user of AD is typically interested in the derivatives of a subset of the output variables with respect to a subset of the input variables of a program. These subsets are commonly referred to as *dependent* variables $J \in \mathbb{R}^m$ and *independent* variables $\vec{\alpha} \in \mathbb{R}^n$, respectively. The differentiable portion of a program is a subset of the overall program, and it is assumed here that the differentiable portion is implemented in a top procedure P_0 and all other procedures in the call tree that are dominated by P_0, that is, called directly or indirectly by P_0, where P_0 can be written as

$$[J, c_{out}] \leftarrow P_0(\vec{\alpha}, c_{in}),$$

and c_{in} and c_{out} are input and output parameters in addition to the independent and dependent variables.

In a CFD context, the dependents could be a quantity that measures the performance of some technical system, such as fuel consumption, lift force, or noise, while the independents could be design inputs such as surface point coordinates. The inputs c_{in} could be constants such as the universal gas constant or turbulence model parameters, and the outputs c_{out} could be diagnostic outputs like final solver residual. The derivative procedures only compute the derivatives of the dependent outputs with respect to the independent inputs. The tangent-linear derivative procedure $\dot{P}_0$ computes the product of the Jacobian of P_0 with the seed vector $\dot{\vec{\alpha}}$ as

$$\dot{J} := \frac{\partial P(\vec{\alpha}, c_{in})}{\partial \vec{\alpha}} \cdot \dot{\vec{\alpha}}, \quad \dot{J} \in \mathbb{R}^m,$$

while the adjoint procedure $\bar{P}_0$ computes the product of the transpose Jacobian with $\bar{J}$ as

$$\bar{\vec{\alpha}} := \left(\frac{\partial P(\vec{\alpha}, c_{in})}{\partial \vec{\alpha}} \right)^T \cdot \bar{J}, \quad \bar{\vec{\alpha}} \in \mathbb{R}^n.$$

P' can be used to refer to either the tangent-linear derivative $\dot{P}$ in forward-mode AD or the adjoint derivative $\bar{P}$ in reverse mode AD. Depending on the programming language, P_0 can be a method, function, procedure or subroutine and the inputs and outputs may be given for example as formal arguments, global variables, or class member

variables. The same variable can be used as both an input and output to P_0, and can be both independent and dependent. This is not a contradiction: In most programming languages, the same memory location, represented by the same variable name, can be used for an independent input before P_0 is called, and then overwritten with a dependent output of P_0. Even though both numbers reside in the same computer memory location (albeit at different points in time) and share the same variable name in a particular implementation of the procedure, these two are distinct mathematical objects. An AD tool must be able to handle these cases correctly.

Industrial problem sizes dictate the use of careful performance optimisation to keep the runtimes of the primal and derivative codes acceptable [113]. The amount of computations in the derivative code can be reduced by performing *activity analysis*, which is the process of identifying *active* variables in the primal code. A variable is said to be active if its current value influences a dependent variable in a differentiable way, and is influenced by an independent variable in a differentiable way. This knowledge forms the basis of many subsequent code analysis and optimisation steps. The activity of variables in a given piece of code can depend on runtime input [45]. AD tools therefore make conservative assumptions during the activity analysis to ensure the correctness of the derivative code. The sharpness of these assumptions is decisive for the efficiency of the generated derivative code.

Activity analysis can be used in source-transformation AD, based on information that is obtained from the source code of the primal program, to generate an efficient derivative program. To this end, the tool may remove all instructions that do not contribute to the derivatives, leading to significant cost savings [45, 96, 143]. The derivative code generated in adjoint mode needs to store some intermediate results of the primal computation, and understanding which of these intermediate results are actually needed for the derivative computation is crucial to keep the memory footprint of the generated program acceptable. Activity analysis is a prerequisite for all this, and is reviewed in the remainder of this section.

In the literature on activity analysis, a variable v is commonly called *varied* if v currently holds a value that was influenced by an independent variable in a differentiable way, and *useful* if v influences a dependent variable in a differentiable way. If v is currently both *varied* and *useful*, v is called *active*. A variable that is not active is called *passive* [45, 85, 96, 143]. To ensure the correctness of the derivative code, it is necessary for static activity analysis to treat all variables as active if they *might* be active for some possible run of the input code. The *actual* activity can vary at runtime, e.g. depending on user input or the current state of the program, and a given piece of source code may have different activities each time that it is executed. Therefore, an *assumed* activity is determined that is a non-strict superset of the actual activity. For the remainder of this

work, we use the word *activity* to refer to the assumed activity, as this is the property that is of practical interest, as opposed to the actual activity that we may not determine with certainty at compile time.

It is desirable to perform as many steps of the activity analysis as possible in an intra-procedural fashion, that is, separately for each procedure, to reduce the time and memory requirements of the analysis. At every call site to another procedure, the analysis depends on the precomputed *differentiation dependency* of the called procedure, operator or intrinsic function. The differentiation dependency determines the way in which an operation changes the variedness and usefulness of all affected variables. Consider a procedure $[w_1 \dots w_m] \leftarrow P(v_1 \dots v_n)$ with n inputs and m outputs. The differentiation dependency of an instruction I that calls P, denoted as $\mathbf{Diff\text{-}dep}(I)$, is defined as a set of variable pairs where $(v,w) \in \mathbf{Diff\text{-}dep}(I)$ if and only if the value of w has a differentiable dependence on v through the call to P. The differentiation dependency of a procedure is the composition of the differentiation dependencies of all contained operators, intrinsic functions and procedure calls. The differentiation dependencies can be computed in a bottom-up sweep through the call tree, so that the property of each procedure is known when a call to it is encountered during the analysis.

Following the differentiation dependency analysis, the activity analysis can be carried out in a top-down sweep through the call tree. For each procedure, a forward sweep through its flow graph is required to determine the variedness, and a reverse sweep is required to determine the usefulness. Both can then easily be combined to determine the activity. For each instruction I, the set of variables that are varied before and after the execution of I are denoted as $\mathbf{Varied}^-(I)$ and $\mathbf{Varied}^+(I)$, respectively. The set of variables that are useful before and after I are denoted as $\mathbf{Useful}^-(I)$ and $\mathbf{Useful}^+(I)$. The relationship between these sets can be expressed in the *data flow equations* shown below and in [74].

$$\begin{aligned}
\mathbf{Varied}^+(I) &= \mathbf{Varied}^-(I) \otimes \mathbf{Diff\text{-}dep}(I) \\
\mathbf{Useful}^-(I) &= \mathbf{Diff\text{-}dep}(I) \otimes \mathbf{Useful}^+(I) \\
\mathbf{Active}^-(I) &= \mathbf{Varied}^-(I) \cap \mathbf{Useful}^-(I) \\
\mathbf{Active}^+(I) &= \mathbf{Varied}^+(I) \cap \mathbf{Useful}^+(I)
\end{aligned} \tag{2.15}$$

The composition $\otimes$ is defined for an arbitrary set of variables $\mathscr{S}$ as

$$\begin{aligned}
v_2 \in \mathscr{S} \otimes \mathbf{Diff\text{-}dep}(I) &\iff \exists v_1 \in \mathscr{S} | (v_1, v_2) \in \mathbf{Diff\text{-}dep}(I) \\
v_1 \in \mathbf{Diff\text{-}dep}(I) \otimes \mathscr{S} &\iff \exists v_2 \in \mathscr{S} | (v_1, v_2) \in \mathbf{Diff\text{-}dep}(I)
\end{aligned}$$

As an example, consider the instruction $\texttt{sub} : w \leftarrow v_1 - v_2$ with inputs v_1, v_2 and output w. If v_1 or v_2 is varied, then w becomes varied. If w is useful, then v_1 and v_2 become

useful. Neither v_1 nor v_2 are modified. The differentiation dependency of the subtraction operator is therefore $\{(v_1,w),(v_2,w),(v_1,v_1),(v_2,v_2)\}$. To illustrate the forward sweep, assume that only v_1 is varied before the subtraction. This means that $\mathbf{Varied}^-(\texttt{sub}) = \{v_1\}$. It follows that $\mathbf{Varied}^+(\texttt{sub}) = \{v_1\} \otimes \{(v_1,w),(v_2,w),(v_1,v_1),(v_2,v_2)\} = \{v_1,w\}$.

An instruction I has one successor and one predecessor, with the exception of the first and last instruction of *basic blocks* such as loop bodies, or code within control flow branches or procedures. The number of predecessors can be larger than one for the first instruction of a basic block, and the number of successors can be larger than one for the last instruction of a basic block. The variedness of variables before I is given by the union of varied variables after the predecessors $pre(I)$ of I. Similarly, the usefulness of variables after the execution of I is given as the union of all useful variables before the successor instructions $suc(I)$ of I. Formally, this can be written as

$$\begin{aligned} \mathbf{Varied}^-(I) &= \bigcup_{I_- \in pre(I)} \mathbf{Varied}^+(I_-) \\ \mathbf{Useful}^+(I) &= \bigcup_{I_+ \in suc(I)} \mathbf{Useful}^-(I_+). \end{aligned} \tag{2.16}$$

The only instruction that does not have a predecessor is the first instruction I_0 of the top procedure, denoted as $I_0(P_0)$, and $\mathbf{Varied}^-(I_0(P_0))$ is the set of independent variables as defined by the user. Likewise, the only instruction without a successor is the final instruction of the top procedure $I_\infty(P_0)$, and $\mathbf{Useful}^+(I_\infty(P_0))$ is given by the user-defined set of dependent variables. For all procedures other than P_0, the variedness before the first instruction depends on the variedness before the call sites to that procedure, and the usefulness after the last instruction depends on the usefulness after the call sites. A simple approach is to use the union of the call site variedness and usefulness given by

$$\mathbf{Varied}^-(I_0(P)) = \bigcup_{I_c \in C(P)} \mathbf{Varied}^-(I_c) \tag{2.17}$$

$$\mathbf{Useful}^+(I_\infty(P)) = \bigcup_{I_c \in C(P)} \mathbf{Useful}^+(I_c), \tag{2.18}$$

where $C(P)$ is the set of all instructions that are call sites to P. This approach can lead to an over-estimation of the activity in P if the variedness and usefulness varies between call sites to P, and to poor performance of the derivative code for two reasons.

1. If a variable v_1 is not varied at the call site, but assumed to be varied before $I_0(P)$, or a variable v_2 is not useful at the call site, but assumed to be useful after $I_\infty(P)$, dummy derivative variables are created for v_1 and v_2 at the call site location. In forward mode, $\dot{v}_1$ needs to be initialised to zero to avoid incorrect derivatives inside $\dot{P}$, while $\dot{v}_2$ receives a derivative value that remains unused. In reverse mode, $\bar{v}_2$ needs to be zeroed and $\bar{v}_1$ receives a value from $\bar{P}$ that remains unused.

2. Inside $\dot{P}$ or $\bar{P}$, if variables are assumed to be active that are actually inactive, this increases the instruction count and memory footprint. For example, derivatives of operations are computed that do not actually affect or use active variables. These derivatives may depend on otherwise unneeded intermediate variables, thus requiring more code from the primal to be inserted into the derivative procedure to compute these intermediate values. Even worse, this additional code may overwrite other variables that need to be stored and retrieved or recomputed in reverse-differentiated code, leading to a cascading effect that includes more and more unneeded code.

Multi-activity AD can overcome this problem by creating multiple specialised differentiated procedures for any given primal procedure, and is the subject of Chapter 4.

2.2.3.4 Example

To illustrate source transformation and activity analysis in forward and reverse modes, a small example code is shown in Source code 2.1. It has an independent input vector $\vec{\alpha} \in \mathbb{R}^2$ and a dependent scalar output $J \in \mathbb{R}$. Additionally, it has a passive input $c_{in} \in \mathbb{R}$ and a passive output $c_{out} \in \mathbb{R}$. The code is written with no variable reuse and with only one operation per line to simplify the discussion in the so-called *single assignment format*. Independent of the programming language, the program can be transformed into a *directed acyclic graph* (DAG) as shown in Figure 2.2 to visualise the difference between forward and reverse differentiation. The differentiation in forward and reverse modes is shown in Source code 2.2 and Source code 2.3.

It can be observed that variables that do not depend on $\vec{\alpha}$ or do not have an influence on J become inactive. It may be worth noting that the activity is the same for the tangent-linear and adjoint codes.

2.2.4 Accuracy and types of errors

This section gives an overview of the kind of errors and inaccuracies that are encountered when computing derivatives with the methods that were introdeuced in the previous sections. Just like any floating point computation, differentiated programs suffer from *roundoff errors*, caused by the fact that floating point numbers have a finite range and number of significant digits, hence almost all real numbers are only approximately represented. The primal and derivative codes are often affected by roundoff errors to a similar extent, although an exception to this is presented in Chapter 7. It should be noted that floating point numbers with any precision, or indeed any numbers that can be stored on a digital computer with finite memory, suffer from the same problem, albeit to a different extent.

```
1  subroutine f( alpha, c_in, J, c_out )
2     real, dimension(2), intent(in)  :: alpha
3     real,               intent(in)  :: c_in
4     real,               intent(out) :: J, c_out
5     real, dimension(3) :: v
6
7     v(1) = sin(alpha(1))
8     v(2) = sqrt(alpha(2))
9     v(3) = v(2)+c_in
10    J = v(1) * v(3)
11    c_out = v(1) / 2.0
12 end subroutine
```

Listing 2.1: Example code in single-assignment format. It is assumed that the user declares `alpha` ($\vec{\alpha}$) as an independent input, and `J` (J) as a dependent output. The resulting derivatives are shown in Source code 2.2 and Source code 2.3.

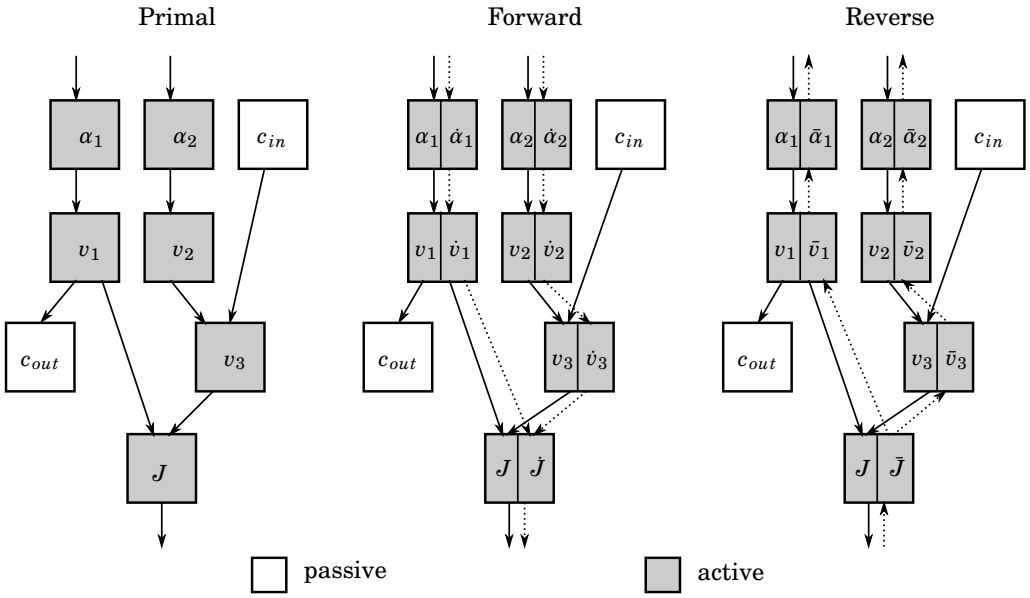


Figure 2.2: Directed acyclic graph of the single assignment code in Source code 2.1. Variables are active if and only if they lie on a path from an independent input to a dependent output. c_{in} is useful (as it affects a dependent output in a differentiable way), but not varied. c_{out} is varied (as it depends on an independent input in a differentiable way), but not useful. Every active variable has a derivative counterpart in forward and reverse computations. Solid arrows represent the data dependencies in the primal computation, such that arrows point from inputs of primal operations to outputs of the same operation. Dotted arrows likewise represent the data dependencies in the derivative computation. Note that the dependencies are flipped in reverse mode.

```
1  subroutine f_d( alpha, alpha_d c_in, J, J_d, c_out )
2     real, dimension(2), intent(in)  :: alpha, alpha_d
3     real,               intent(in)  :: c_in
4     real,               intent(out) :: J, J_d, c_out
5     real, dimension(3) :: v, v_d
6     v_d(1) = alpha_d(1)*cos(alpha(1))
7     v(1) = sin(alpha(1))
8     v_d(2) = alphad(2)/(2.0*sqrt(alpha(2)))
9     v(2) = sqrt(alpha(2))
10    v_d(3) = v_d(2)
11    v(3) = v(2) + c_in
12    J_d = v_d(1)*v(3) + v(1)*v_d(3)
13    J = v(1) * v(3)
14    c_out = v(1)/2.0
15 end subroutine
```

Listing 2.2: Tangent-linear code for Source code 2.1. Active variables have counterparts with a suffix `_d` (association by name). The derivative and primal computation is interleaved. Data dependencies are the same for primal and derivative variables.

```
1  subroutine f_b( alpha, alpha_b c_in, J, J_b, c_out )
2     real, dimension(2), intent(in)  :: alpha
3     real, dimension(2), intent(out) :: alpha_b
4     real,               intent(in)  :: c_in
5     real,               intent(out) :: J, c_out
6     real,               intent(in)  :: J_b
7     real, dimension(3) :: v, v_b
8     v(1) = sin(alpha(1))
9     v(2) = sqrt(alpha(2))
10    v(3) = v(2) + c_in
11    J = v(1) * v(3)
12    c_out = v(1) / 2.0
13
14    v_b(1) = v(3)*J_b
15    v_b(3) = v(1)*J_b
16    v_b(2) = v_b(3)
17    alpha_b(2) = v_b(2)/(2.0*sqrt(alpha(2)))
18    alpha_b(1) = cos(alpha(1))*v_b(1)
19 end subroutine
```

Listing 2.3: Adjoint code for Source code 2.1. Active variables have derivative counterparts with a suffix `_b` (association by name). The derivative computation happens after the primal computation is complete, and in reverse order. The data dependencies are inverted: If a variable is an input, then its derivative counterpart is an output, and if a variable is an output, then its derivative counterpart is an input.

CFD programs are affected by *modelling errors*, which describe the mismatch between the actual physics and the partial differential equation (PDE) chosen to represent them. In addition, they are usually affected by *discretisation errors*, which describe the mismatch between the PDE and a discretisation thereof. Because of the discretisation errors, the derivatives computed using AD only reflect the behaviour of the PDE model in the limit of an infinitely fine discretisation. However, the computed derivatives accurately reflect the derivatives of the primal program (except for roundoff errors), a property that is referred to as *consistency*.

The *continuous adjoint method* suffers, in addition to roundoff errors and discretisation errors, from a *consistency error*, that is, the computed derivatives reflect the derivatives of the primal program only in the limit of vanishing discretisation errors, which is a consequence of the rediscretisation of the differentiated physical model. It should be noted that the discretisation of the primal and adjoint systems can be designed such that consistency is preserved, a property that is used for example in the time discretisation of some adjoint CFD solvers [29, 58].

Consistency does not guarantee that the physical behaviour is captured correctly, but rather that a sufficiently small gradient-based optimisation step will actually cause the objective function *as computed by the primal program* to change in the desired direction. Furthermore, consistency makes it possible to perform meaningful verification of derivative codes, as the derivatives computed by any two consistent methods must match in the limit of vanishing roundoff errors.

The differentiation of fixed point iterations, such as those shown in equations (2.5) and (2.7), can be made more efficient by evaluating the derivative only at the final solution [31]. Such an approach is only consistent in the limit of full convergence of the iterative scheme. The same is true for the symbolic differentiation of iterative linear solvers outlined in Section 2.3.

Finally, the *finite difference* method suffers from truncation errors due to finite-sized perturbations, in addition to the roundoff and discretisation errors of other methods. It does not suffer from consistency errors.

2.3 Differentiating linear solvers

Linear solvers are an important part of many numerical algorithms, including implicit CFD solvers. Given an invertible real matrix $\vec{A}$ and a real right hand side vector $\vec{b}$, a linear solver computes the result $\vec{y}$ such that

$$\vec{y} = \vec{A}^{-1} \cdot \vec{b}.$$

In the literature one can find a variety of methods for solving linear systems. The appropriate choice of a linear solver method depends, among other things, on the structure, size, sparsity, and condition of $\vec{A}$.

Most CFD codes use *iterative linear solvers*, as they are often faster, use less memory, and are easier to parallelise than the alternative, *direct solvers*. Since scalability and small memory footprint are especially important on accelerator devices, iterative solvers are likely to remain popular. Iterative solvers create a sequence of approximate solutions $\vec{y}^1, \vec{y}^2, \ldots$, where the error (residual)

$$\epsilon = \|\vec{b} - \vec{A} \cdot \vec{y}^i\|$$

is expected to converge to zero as the iteration number i is increased. The solver is typically stopped once the residual is small enough. The number of iterations it takes to achieve this is influenced by the initial guess $\vec{y}_0$, where a poorly chosen initial guess can slow down the solver or even prevent a solution from being found.

One may be tempted to apply AD to a program that contains an iterative linear solver. For the more general concept of implicit functions implemented by fixed-point loops, such an approach is known to potentially cause problems [32], as the primal solution and the derivatives may take a different number of iterations to converge and therefore require separate stopping criteria. For many commonly used linear solvers, little is known about the convergence properties [10, 137], and even less is known about the convergence of derivatives of such solvers. Previous work has found that the derivatives of certain linear solvers computed with AD have a noisy behaviour [112]. It is shown in Section 7.4 that even if some formal convergence guarantees can be made for real arithmetic, roundoff errors can be a particular issue when applying AD to linear solvers.

For these reasons, as well as to improve performance and avoid issues with differentiating external linear solver libraries, linear solvers are commonly excluded from the AD process and differentiated by hand. This is helped by the fact that the solution of a linear system is easy to differentiate symbolically [59]. For a linear solver call

$$\vec{y} \leftarrow \texttt{solve}(\vec{A}, \vec{b}, \vec{y}_0), \tag{2.19}$$

the forward derivative can be implemented as

$$\dot{\vec{y}} \leftarrow \texttt{solve}(\vec{A}, \dot{\vec{b}} - \dot{\vec{A}}\vec{y}, \hat{\vec{y}}_0), \tag{2.20}$$

while the adjoint derivative can be implemented as

$$\begin{aligned} \bar{\vec{b}} &\leftarrow \texttt{solve}(\vec{A}^T, \bar{\vec{y}}, \hat{\bar{\vec{b}}}_0) \\ \bar{\vec{A}} &\leftarrow -\bar{\vec{b}} \cdot \vec{y}^T \\ \bar{\vec{y}}_0 &\leftarrow 0. \end{aligned} \tag{2.21}$$

It is worth noting that the derivative codes require the solution of another linear system. If iterative solvers are used for the derivative systems, one also has to provide an initial guess $\hat{\vec{y}}_0$ or $\hat{\vec{b}}_0$. The system matrix for the tangent-linear code is identical to that of the primal code, whereas the system matrix of the adjoint code is the transpose of the primal system matrix. It is common to use the same iterative solvers for the primal and derivative systems, and one might assume that the primal and derivative systems will converge similarly well, as the system matrices have similar properties. Counterexamples to this assumption are shown in Chapter 7.

The adjoint derivative $\bar{\vec{y}}_0$ represents the derivative of the program output with respect to the initial guess of the linear solver. If the solver converged, this quantity must be zero. Otherwise, a correction term must be added to $\bar{\vec{y}}_0$ to ensure consistency [1].

2.4 Implementation of primal and adjoint solvers

The work in this thesis is based on the compressible flow solver `stamps` [29]. It supports explicit or implicit time stepping as shown in equations (2.5) and (2.7). The preconditioner is either computed using the point-implicit block-Jacobi method [130] or using first-order neighbours, optionally in combination with a geometric multigrid scheme [176].

The spatial discretisation uses a second-order node-centred finite volume scheme [17, 37]. The mesh is handled as an undirected graph, where each node represents a control volume, and each edge represents the interface between two control volumes. The residual function $F(\vec{U})$ in equation (2.4) is implemented as a loop over edges in this graph, where an edge flux function is computed based on the flow state in the two nodes connected by that edge, and the value of that flux function is used to accumulate the residual at the same two nodes. This process was parallelised using the method outlined in Section 2.5. In the `stamps` source code, the parallel loop over edges is contained in a subroutine *residual*$(\vec{R}\uparrow, \vec{X}\downarrow, \vec{U}\downarrow)$, where the output vector, denoted with $\uparrow$, is computed based on the input vector, denoted with $\downarrow$. $\vec{U}$ contains the flow state and $\vec{R}$ contains the residual $F(\vec{U})$ for all mesh nodes, and $\vec{X}$ contains other active mesh properties such as node coordinates, surface normals, and cell volumes.

The adjoint solver is created using a mix of hand-differentiation and AD. The preconditioning, temporal discretisation, fixed point loops, and linear solvers are hand-differentiated, using the fact that the Runge-Kutta scheme and multigrid are self-adjoint [58] and the differentiation of the BDF2 scheme is straightforward. The cost function and the spatial discretisation, that is, *residual* and all subroutines called by it, are differentiated using the Tapenade AD tool [74], and parallelised using the method discussed in Chapter 5. A related method was described by Giering and Kaminski in [54].

Since the solver finds a valid flow state $\vec{U}$ such that $F(\vec{U}) \approx 0$ in a fixed point iteration, the primal solver only needs to store the fully converged solution for each physical time step, as the convergence history is not needed to compute the adjoint derivatives. Following the procedure presented in [32], it is also unnecessary to compute any derivatives with respect to the geometric quantities $\vec{X}$ in each adjoint iteration. Instead, AD is used to create two different versions of the adjoint residual subroutine, a process that is helped greatly by the multi-activity differentiation shown in Chapter 4. Overall, the flow and adjoint solver can be described by the pseudo-code in Algorithm 1.

Input: Initial guess $\vec{U}_0$, design vector $\vec{\alpha}$
Output: Flow trajectory $\vec{U}^1 \ldots \vec{U}^{t_{final}}$, cost J, sensitivity $\bar{\vec{\alpha}}$
metrics($\vec{\alpha}\downarrow, \vec{X}\uparrow$)
$t \leftarrow 0$
while $t < t_{final}$ **do** // unsteady primal loop
  $\vec{U}_{t+1} \leftarrow \vec{U}_t$ // initial guess from previous forward step
  $t \leftarrow t+1$
  repeat
    residual($\vec{R}\uparrow, \vec{X}\downarrow, \vec{U}^t\downarrow$) // primal residual()
    update($\vec{U}^t\uparrow, \vec{R}\downarrow$)
  until $\|\vec{R}\| \leq$ *cutoff*
  store($\vec{U}^t\downarrow$)
end
$\bar{J} \leftarrow 1$
cost_b($J\uparrow, \bar{J}\downarrow, \vec{U}^t\downarrow, \bar{\vec{U}}^t\uparrow, \vec{X}\downarrow, \bar{\vec{X}}\uparrow$)
while $t \geq 0$ **do** // unsteady adjoint loop
  load($\vec{U}^t\uparrow$)
  repeat
    update_adj($\vec{U}^t\uparrow, \bar{\vec{U}}^t\uparrow, \vec{R}\downarrow, \bar{\vec{R}}\uparrow$) // hand-coded adjoint of update()
    residual_b_u($\vec{R}\uparrow, \bar{\vec{R}}\downarrow, \vec{X}\downarrow, \vec{U}^t\downarrow, \bar{\vec{U}}^t\uparrow$) // generated by AD
  until $\vec{R} \leq$ *cutoff*
  $\bar{\vec{U}}_{t-1} \leftarrow \bar{\vec{U}}_t$ // initial guess for next reverse step
  $t \leftarrow t-1$
end
residual_b_x($\vec{R}\uparrow, \bar{\vec{R}}\downarrow, \vec{X}\downarrow, \bar{\vec{X}}\uparrow, \vec{U}^t\downarrow, \bar{\vec{U}}^t\uparrow$) // generated by AD
metrics_b($\vec{\alpha}\downarrow, \bar{\vec{\alpha}}\uparrow, \vec{X}\uparrow, \downarrow$) // generated by AD

Algorithm 1: stamps primal and adjoint flow solver. The routine *residual_b_u* is generated by Tapenade, with $\vec{R}$ as a dependent and $\vec{U}$ as the only independent variable. *residual_b_x* is generated declaring $\vec{U}$ and $\vec{X}$ as independent.

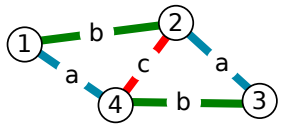


Figure 2.3: Example of a mesh with colouring (colours shown as letters on edges). Fluxes along all edges with the same colour can be computed in parallel. The nodes with numbers 2 and 4 have a vertex degree of 3 (i.e., three neighbours), meaning that at least 3 colours are needed for this mesh.

2.5 Shared-memory parallelism with OpenMP

OpenMP is a standard for shared-memory parallelisation of computer programs [35]. Parallel regions are declared in the program source code using *pragmas*, and are executed in parallel by multiple *threads*, where the maximum number of threads that can be efficiently executed at the same time on a given computer is usually a function of the number of processor cores. The pragmas also declare variables as `shared` between threads, or as `private` to each thread. The standard defines, among other things, pragmas for reduction operations, vectorisation, or execution on accelerator devices.

OpenMP pragmas should change the execution speed of the program, but not its result. As a consequence, a program that correctly uses OpenMP pragmas computes the same result whether it is compiled with or without enabled OpenMP compiler support. The main obstacle in ensuring correctness of shared-memory-parallel programs lies in the avoidance of *race conditions*, a programming error that can occur if a thread modifies the value of a shared variable that is also used or changed by another thread. OpenMP offers methods to ensure that the outcome of such operations is well defined. A technique that is specific to avoiding race conditions in unstructured CFD codes is reviewed in this section. Some other OpenMP constructs for this purpose are shown in Appendix A.

As described in Section 2.4, the residual computation in `stamps` can be regarded as a loop over edges of a graph, wherein information is exchanged between nodes connected by an edge. The connectivity that defines this graph is considered to be stored in $edg2nde(e)$ as a mapping from an edge e to the nodes l, r that are connected by e.

for $e \leftarrow 1 \ldots n_{edges}$ **do**
  $l, r \leftarrow edg2nde(i)$;
  $\vec{R}_l \xleftarrow{+} f_l(\vec{U}_l, \vec{U}_r)$;
  $\vec{R}_r \xleftarrow{+} f_r(\vec{U}_l, \vec{U}_r)$;
end

Algorithm 2: Interface-based residual computation 1D. Note that U here refers to all information about the flow stored in the graph nodes, which may include spatial gradients.

Although this is not always obvious from the source code, the residual computation is equivalent to a sum-reduction where the contributions of iterations are gathered in $\vec{R}$.

If the residual computation is parallelised, race conditions can occur if the flux for two edges that have a common node are computed simultaneously. Using an OpenMP atomic or critical section pragma would ensure that only one thread at a time can increment the output vector. However, this would require other threads to wait and hence slows down the execution. Alternatively, one could use an OpenMP reduction pragma, which results in the creation of a private copy that each thread initialises to zero, increments by its own local contributions, and finally merges with other threads to a global result vector. The memory footprint of this approach is not acceptable for a CFD code, especially on a many-core machines, as it would require hundreds of copies of the residual vector. Moreover, the performance overhead of the reduction pragma is not negligible, as each thread contributes only to some elements, but needs to initialise the entire local copy to zero, and the final merging will treat those vectors as dense vectors despite the number of nonzero elements per thread only being $\mathcal{O}(n_{edges}/n_{threads})$.

For this reason, it is preferable to avoid write conflicts in unstructured CFD codes by using a colouring on the edges [70] or colouring with mini-partitions [60]. This approach has been described for other application areas as well, such as mesh smoothing [63]. The colouring is used in this work to form groups of iterations (distinguished by "colour") such that within each group no index in the output array is accessed more than once, and therefore all computations within one group can be performed in parallel.

The following example may clarify the colouring approach. The residual computation for the coloured graph in Figure 2.3 can be performed simultaneously for edges that share the same colour, as they do not write to the same elements in the shared residual vector:

$$\begin{aligned}\vec{R} = &\begin{bmatrix} f_l(\vec{U}_1,\vec{U}_4) \\ 0 \\ 0 \\ f_r(\vec{U}_1,\vec{U}_4) \end{bmatrix} + \begin{bmatrix} 0 \\ f_l(\vec{U}_2,\vec{U}_3) \\ f_r(\vec{U}_2,\vec{U}_3) \\ 0 \end{bmatrix} \\ + &\begin{bmatrix} f_l(\vec{U}_1,\vec{U}_2) \\ f_r(\vec{U}_1,\vec{U}_2) \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ f_l(\vec{U}_3,\vec{U}_4) \\ f_r(\vec{U}_3,\vec{U}_4) \end{bmatrix} \\ + &\begin{bmatrix} 0 \\ f_l(\vec{U}_4,\vec{U}_2) \\ 0 \\ f_r(\vec{U}_4,\vec{U}_2) \end{bmatrix}\end{aligned} \tag{2.22}$$

Given a mesh with a valid colouring, the following statements are all equivalent.

- Two edges $e_1 = \{v_1, v_2\}$ and $e_2 = \{v_3, v_4\}$ have the same colour.
- v_1, v_2, v_3, v_4 are pairwise distinct.
- No index in the output array is written to by both the fluxes on e_1 and e_2.
- The flux update on e_1 and e_2 can be executed in parallel.

The edge flux loop in Algorithm 2 can thus be implemented as an outer serial loop over all colours, and an OpenMP-parallel inner loop over all edges with that colour as shown in Algorithm 3. The parallel loop requires no synchronisation except for the implicit barrier at the end of each OpenMP parallel loop. It can be expected that the synchronisation overhead is small, as the number of colours determines the number of barriers, and is typically in the order of 10 for meshes used in practice.

```
!$OMP parallel shared(Ū,R̄) private(c,e);
for c ← 1...n_colours do
    !$OMP for;
    foreach e ∈edges(c) do
        l,r ← edg2nde(e);
        R̄_l +← f_l(Ū_l,Ū_r);
        R̄_r +← f_r(Ū_r,Ū_l);
    end
end
```

Algorithm 3: Residual computation as a nested loop where each inner iteration computes the flux f along one edge e, reading $\vec{U}$ and writing $\vec{R}$ at the indices l and r that correspond to the nodes connected by that edge. Each outer iteration computes all updates for a given colour.

The `stamps` solver uses a greedy edge colouring heuristic [103] without edge preordering, which sometimes results in a higher than optimum number of colours. The maximum vertex degree of a mesh can be used as an approximate lower bound for the minimum number of required colours to assess the performance of the colouring heuristic [42], and the number of required colours was found to be close to the optimum for the test cases presented in this work.

It should be noted that the schedule for this parallelisation is determined only after the colouring of the mesh is known, at runtime of the program. Nevertheless, the implementation guarantees that there are no conflicting write accesses if a correct mesh colouring is provided. If every edge has a different colour, the parallel algorithm in Algorithm 3 degenerates to the serial algorithm shown in Algorithm 2.

CHAPTER 3

Test cases

The performance of the parallelisation strategy in Chapter 5 and the activity analysis in Chapter 4, as well as the accuracy and memory footprint of the incomplete checkpointing approach in Chapter 6, are evaluated using one of the two test cases described here.

All performance tests are run on two Intel Xeon E5-2660 CPUs clocked at 2.2 GHz with 8 cores and 20 MB L3 cache each, running Scientific Linux 6.2. The codes are compiled with `gfortran` 4.8.2 with the `-O3` flag and `scatter` OpenMP thread affinity; AD code is generated with Tapenade 3.11 and then compiled with the same flags as the primal. This setup is referred to in the remainder of this work as the CPU system. Further, performance is tested on an Intel XeonPhi 5110P Knights Corner many-integrated-core (MIC) coprocessor clocked at 1.053 GHz using native execution (i.e., the full code runs on the MIC system, not just parallel sections). The solver is compiled for the MIC using `ifort` 14.0 with `-fast -mmic` flags and `balanced` OpenMP thread affinity. There are 60 cores of which 59 are used for the experiments, and the remaining core is left to the operating system. Each core supports up to 4 threads and has 512 kB of L2 cache.

3.1 U-bend (BEND)

In this test case, flow through a U-shaped channel is simulated, and the adjoint derivatives of the pressure loss with respect to surface displacements of the channel wall are computed. The test case will be referred to as BEND in other chapters, and represents a cooling channel that is used inside turbine fan blades to prevent damage from the combustion temperatures. Adjoint derivatives are useful to optimise the channel for lower pressure loss or better heat exchange, which can increase aircraft fuel efficiency [165].

The geometry was previously described in [164]. The mesh is generated as a fully structured cartesian mesh, after which a coordinate transformation is applied to create the bend shape. Although the mesh is structured, the solver treats it as an unstructured mesh. The simplicity of the geometry allows the automatic generation of the mesh for a variety of different resolutions to investigate the solver speed. Figure 3.1 shows an example with 200,000 nodes. Depending on the node numbering, the mesh colouring algorithm in `stamps` requires between 6 and 11 colours for this mesh, where the optimum number of colours that can be achieved for this mesh is 6. All volume elements are hexahedral, and all boundary faces are quadrilateral.

The flow is set up with a fixed inlet velocity of Ma 0.02565 (the Mach number is defined as a fraction of the speed of sound; under the flow conditions inside the bend this is ca. 8 m/s) and a fixed pressure outlet, leading to a Reynolds number of ca. 40000. The channel has viscous non-slip walls, implicit LES is used [51].

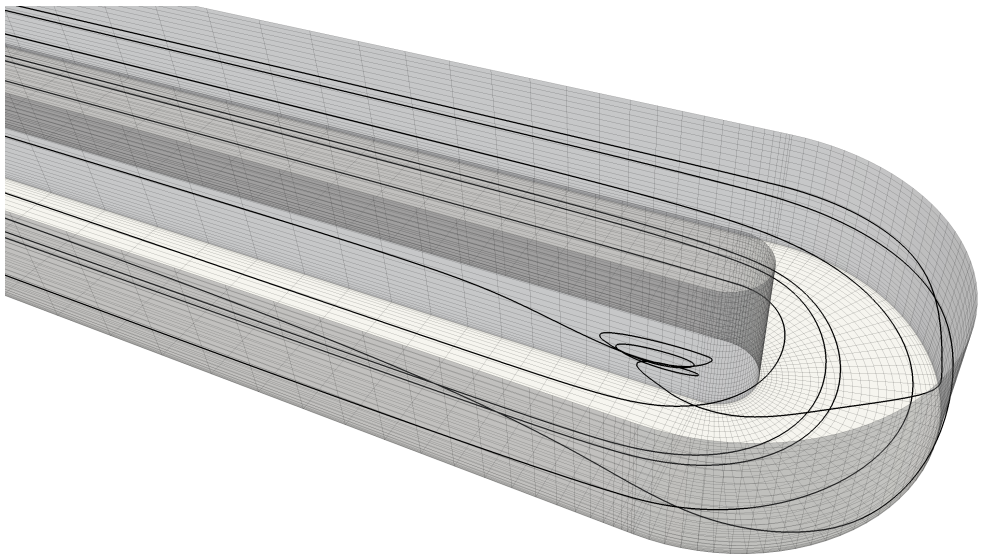


Figure 3.1: **BEND:** U-Bend model of an internal turbine blade cooling channel. Structured hexahedral mesh. Quadrilateral surface mesh and streamlines shown.

3.2 Truncated airfoil (RAE2822)

The RAE2822 airfoil is an often-used test case for aerodynamic simulation and optimisation [92]. In this work, it is discretised using 85,000 mesh nodes, see fig. 3.3. The `stamps` solver supports only three-dimensional meshes, which is overcome in this case by using a mesh with one cell thickness, and symmetry boundary conditions in the z-direction. The airfoil has viscous wall boundary conditions at the surface. The trailing edge is truncated at 95% cord length to provoke more vortex shedding, which was desired to test the accuracy of the incomplete checkpointing approach in Chapter 6.

The flow is set up with a 30° angle of attack to provoke a high amount of shedding, and a freestream velocity of 0.2 Ma. Flow is simulated using the implicit LES model [5].

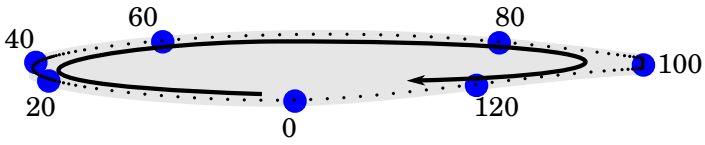


Figure 3.2: The airfoil surface is meshed with 130 nodes, numbered clockwise from the bottom centre.

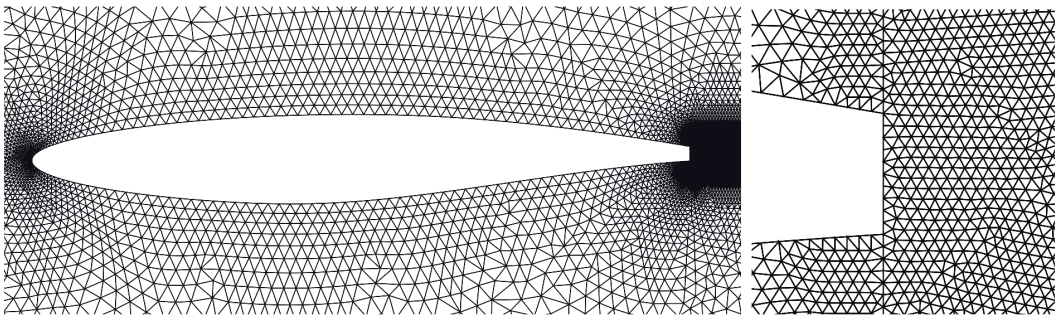


Figure 3.3: **Left:** RAE2822 airfoil with a section of the mesh. **Right:** close-up view of the blunt trailing edge.

CHAPTER 4

Multi-activity differentiation

This chapter presents a refined activity analysis strategy for source-transformation AD. As shown in Section 2.2.3.3, activity analysis is the process of determining which variables in a given input program are *active*. Active variables need a derivative counterpart variable, and operations that access active variables in memory need to be differentiated.

Each intermediate value that arises during the execution of a computer program is either active or passive. However, variables in most nontrivial programs are used several times to store different values, for example as an accumulator that receives contributions from several loop iterations. Each of these different values may be active or not, and therefore the activity of a variable may change over time. Moreover, a given piece of source code can be executed in more than one context. This occurs for example to the body of a loop, or a procedure that is called from more than one location. The activity of the variables in such a piece of code may depend on the current context of its execution.

In such situations, *multi-activity AD* may provide more efficient derivative code in forward and reverse-mode, by generating specialised derivative code for each context. Most of this chapter focuses on the situation where entire procedures are specialised for each call site. The specialisation can be used globally in the entire code, or selectively for certain call sites or routines. A global approach purely based on the detected call site activity was used in early versions of ADIFOR [14], but has been dropped from later versions without a thorough investigation of its benefits. We have implemented a refined approach that allows a user to selectively specialise the differentiation in the Tapenade AD tool, and the method shown in this chapter can be used as of version 3.11.

4.1 Example: benefits of activity analysis

This section shows how activity analysis and multi-activity differentiation can reduce the computational cost and memory footprint of adjoint computations using a small,

contrived example. The case study in Section 4.5 shows that multi-activity differentiation can also pay off in practice. Consider a primal code that computes $w \leftarrow w \cdot v^N$ as follows.

```
1   do i=1,N
2     w = v*w
3   end do
```

If differentiated using only w as an active variable, the adjoint uses the same number of operations as the primal to compute $\bar{w} \leftarrow \bar{w} \cdot v^N$ with constant storage requirement.

```
1   do i=N,1,-1
2     wb = v*wb
3   end do
```

If differentiated assuming both v and w to be active variables, the adjoint computation uses three times as many operations, and has a memory footprint of $\mathcal{O}(N)$ due to the fact that intermediate values of w need to be pushed onto a stack [34] in the primal computation, as they are needed in reverse order in the adjoint computation.

```
1   do i=1,N
2     call push(w)
3     w = v*w
4   end do
5   do i=N,1,-1
6     call pop(w)
7     xb = xb + w*wb
8     wb = v*wb
9   end do
```

If a subroutine containing this code is called from one call site with all its arguments active, and from another call site with only w active, multi-activity AD can generate both versions of the adjoint code, and use the most efficient one in each context.

4.2 Method description

Consider a procedure P in a program that is being differentiated. That procedure may be called from several call sites, at each of which a subset of the arguments to P is varied, and a subset of the arguments to P is useful. The pair containing the set of varied variables $\mathcal{V}$ and the set of useful variables $\mathcal{U}$ is called an *activity pattern*

$$\mathcal{A} = (\mathcal{V}, \mathcal{U}),$$

and each call site to P may have its own, unique activity pattern.

The data flow equations from Section 2.2.3.3 can be extended by allowing each statement to have a different variedness and usefulness for each activity pattern of the containing procedure. This is shown in Section 4.2.1. Multi-activity differentiation was implemented in this work in a way that allows user control. The reasons and implications of this are presented in Section 4.3.1. Lastly, the derivative code has to be cloned and each version adapted to its call site, which is outlined in Section 4.2.2.

4.2.1 Extended data flow equations

The variedness of the first instruction and the usefulness of the last instruction of P is connected to the activity pattern $\mathscr{A}$ of P through

$$\begin{aligned} \mathscr{A} &:= (\mathscr{V}, \mathscr{U}) \\ \mathbf{Varied}^{-}(I_0(P,\mathscr{A})) &= \mathscr{V} \\ \mathbf{Useful}^{+}(I_\infty(P,\mathscr{A})) &= \mathscr{U}. \end{aligned} \tag{4.1}$$

The data flow equations shown previously in (2.15) can be defined for a given activity pattern $\mathscr{A}$ as

$$\begin{aligned} \mathbf{Varied}^{+}(I,\mathscr{A}) &= \mathbf{Varied}^{-}(I,\mathscr{A}) \otimes \mathbf{Diff\text{-}dep}(I) \\ \mathbf{Useful}^{-}(I,\mathscr{A}) &= \mathbf{Diff\text{-}dep}(I) \otimes \mathbf{Useful}^{+}(I,\mathscr{A}) \\ \mathbf{Active}^{-}(I,\mathscr{A}) &= \mathbf{Varied}^{-}(I,\mathscr{A}) \cap \mathbf{Useful}^{-}(I,\mathscr{A}) \\ \mathbf{Active}^{+}(I,\mathscr{A}) &= \mathbf{Varied}^{+}(I,\mathscr{A}) \cap \mathbf{Useful}^{+}(I,\mathscr{A}), \end{aligned} \tag{4.2}$$

where the definition of the differentiation dependency is unchanged by the multi-activity approach, as it is a property only of the instruction I itself, and is independent of the context in which I is placed.

4.2.2 Code generation rules

Inference rules for forward and reverse differentiation have been shown in [74], following the natural semantics notation [91]. These rules are a way to formalise the transformation of primal code into derivative code. Code generation is the step that follows the analysis in Section 4.2.1, and is based on the activity patterns selected in Section 4.3.1. For every primal procedure P, the AD tool must generate a total number of $\|\mathbb{A}(P)\|$ specialised derivative procedures, one for every activity pattern $A \in \mathbb{A}(P)$ of that procedure.

As an example, the inference rule for the forward-differentiation of procedure headers is shown here. The following rule defines a code transformation for every procedure P,

where $\mathscr{A} \vdash$ denotes that the rule is executed for each activity pattern.

$$\frac{\mathit{NAMEHYP} \qquad \mathit{ARGSHYP} \qquad \mathit{INSTRHYP} \qquad \mathit{DECLHYP}}{\mathscr{A} \vdash \texttt{procedure}\ P\,\texttt{(}\mathit{ARGS}\texttt{)}\ \texttt{\{}\mathit{DECLS}\texttt{;}\ \mathit{INSTRS}\texttt{\}} \rightarrow \texttt{procedure}\ \dot{P}\,\texttt{(}\dot{\mathit{ARGS}}\texttt{)}\ \texttt{\{}\dot{\mathit{DECLS}}\texttt{;}\ \dot{\mathit{INSTRS}}\texttt{\}}} \tag{4.3}$$

This rule connects one so-called *conclusion predicate* below the fraction bar, with zero or more (here four) *hypothesis predicates* above the fraction bar given by

$$\mathit{NAMEHYP} = \mathscr{A} \vdash P \xrightarrow{\mathit{procName}} \dot{P}$$

$$\mathit{ARGSHYP} = \mathscr{A}, P, 0 \vdash \mathit{ARGS} \rightarrow \dot{\mathit{ARGS}}$$

$$\mathit{INSTRHYP} = \mathscr{A} \vdash \mathit{INSTRS} \rightarrow \dot{\mathit{INSTRS}}$$

$$\mathit{DECLHYP} = \mathscr{A} \vdash \mathit{DECLS} \rightarrow \dot{\mathit{DECLS}}$$

Each predicate represents some code transformation or rewrite, and is considered solved when it is successfully applied to some code. In predicates, an arrow separates the code before and after rewriting. To solve the conclusion predicate of a given rule, all its hypothesis predicates must be solved recursively, using other rules. With this in mind, the above inference rule can be read, or executed by a code rewriting system, as follows: for each activity pattern, if the primal code matches the pattern

$$\texttt{procedure}\ P\,\texttt{(}\mathit{ARGS}\texttt{)}\ \texttt{\{}\mathit{DECLS}\texttt{;}\ \mathit{INSTRS}\texttt{\}},$$

thus instantiating variables P, ARGS, DECLS, and INSTRS with the corresponding code pieces, and if the four hypothesis predicates can be recursively solved using other inference rules, thus instantiating variables $\dot{P}$, $\dot{\mathit{ARGS}}$, $\dot{\mathit{DECLS}}$, $\dot{\mathit{INSTRS}}$, then the conclusion predicate is solved and it produces the derivative code built as

$$\texttt{procedure}\ \dot{P}\,\texttt{(}\dot{\mathit{ARGS}}\texttt{)}\ \texttt{\{}\dot{\mathit{DECLS}}\texttt{;}\ \dot{\mathit{INSTRS}}\texttt{\}},$$

where the variables P, ARGS, DECLS, and INSTRS hold the procedure name, and its arguments, declarations, and instructions. Some utility predicates for the elementary rewrite operations need to be defined, identified by a superscript above the arrow. For instance, predicate

$$\mathscr{A} \vdash P \xrightarrow{\mathit{procName}} \dot{P}$$

means that the name P of the procedure is transformed into the name $\dot{P}$ of its differentiated version for activity pattern $\mathscr{A}$. Predicate *procName* deals with an important aspect of the multi-activity approach: without specialisation, it can act simply by appending a suffix to the procedure name, e.g. `_d` for forward and `_b` for reverse-differentiation. If however more than one specialisation is created, it is necessary to generate unique

suffixes for each activity pattern to avoid assigning the same name to several procedures, e.g. by encoding the activity in a string, or by numbering the patterns. To avoid generating excessively long procedure names, the latter approach was chosen in this work and a number is appended, starting from zero, whenever two procedures would otherwise have the same name. There is no natural way to define an order over activity patterns, hence the numbering depends on the order in which specialisations are created, which depends on implementation details of Tapenade and the primal code. The user can choose custom suffixes for differentiation heads to make the naming predictable if needed.

The rewrite predicate for the procedure arguments (the second hypothesis predicate in (4.3)) requires as a context, in addition to $\mathscr{A}$, the current procedure P and the index of the next argument. The predicate can itself be formalised in the following rewrite rules for the arguments list, whose first hypothesis predicate is a boolean condition that selects the applicable rule:

$$\frac{\mathit{isDiffArg}(\mathscr{A},P,rk) \quad ARG \xrightarrow{varName} \dot{ARG} \quad \mathscr{A},P,rk+1 \vdash ARGS \to \dot{ARGS}}{\mathscr{A},P,rk \vdash (ARG\ .\ ARGS) \to (ARG, \dot{ARG}\ .\ \dot{ARGS})}$$

$$\frac{!\mathit{isDiffArg}(\mathscr{A},P,rk) \qquad \mathscr{A},P,rk+1 \vdash ARGS \to \dot{ARGS}}{\mathscr{A},P,rk \vdash (ARG\ .\ ARGS) \to (ARG\ .\ \dot{ARGS})}$$

The predicate $\mathit{isDiffArg}(\mathscr{A},P,rk)$ is true if the rk^{th} formal argument of procedure P is active for $\mathscr{A}$, i.e. it belongs to $\mathbf{Active}^-(I_0(P),\mathscr{A})$ or to $\mathbf{Active}^+(I_\infty(P),\mathscr{A})$. In that case, the derivative argument $\dot{ARG}$ is inserted into the derivative arguments list, and the primal argument ARG is inserted in all cases. The adapted inference rules for a procedure call are shown below, together with the rules for differentiating the arguments of the call and the variable references contained in these arguments.

$$\frac{\mathit{isActiveCall}(\mathscr{A},\mathscr{A}_c) \qquad \mathscr{A}_c \vdash P \xrightarrow{procName} \dot{P} \qquad \mathscr{A},\mathscr{A}_c,P,0 \vdash ARGS \xrightarrow{actualArgs} \dot{ARGS}}{\mathscr{A} \vdash \texttt{call}\ P(ARGS) \to \texttt{call}\ \dot{P}(\dot{ARGS})} \tag{4.4}$$

$$\frac{\mathit{isDiffArg}(\mathscr{A}_c,P,rk) \qquad \mathscr{A} \vdash EXPR \xrightarrow{ref} \dot{EXPR} \qquad \mathscr{A},\mathscr{A}_c,P,rk+1 \vdash EXPRS \xrightarrow{actualArgs} \dot{EXPRS}}{\mathscr{A},\mathscr{A}_c,P,rk \vdash (EXPR\ .\ EXPRS) \xrightarrow{actualArgs} (EXPR, \dot{EXPR}\ .\ \dot{EXPRS})}$$

$$\frac{!\mathit{isDiffArg}(\mathscr{A}_c,P,rk) \qquad \mathscr{A},\mathscr{A}_c,P,rk+1 \vdash EXPRS \xrightarrow{actualArgs} \dot{EXPRS}}{\mathscr{A},\mathscr{A}_c,P,rk \vdash (EXPR\ .\ EXPRS) \xrightarrow{actualArgs} (EXPR\ .\ \dot{EXPRS})}$$

All properties of calls and of call arguments are functions of the current activity $\mathscr{A}$ of the containing procedure, or of the corresponding called activity of the called procedure. In particular, $\mathit{isActiveCall}(\mathscr{A},\mathscr{A}_c)$ is true at call site I_c if one argument of this call is in $\mathbf{Active}^-(I_c,\mathscr{A})$ or in $\mathbf{Active}^+(I_c,\mathscr{A})$. In that case, this prerequisite unifies (i.e. "sets")

$\mathscr{A}_c$ with the corresponding activity for the called procedure. It is important to note that $\mathscr{A}_c$ is an activity pattern of the *called* procedure, while $\mathscr{A}$ is an activity pattern of the *calling* procedure. For any given call site and activity pattern, it is necessary to find an activity pattern $\mathscr{A}_c \in \mathbb{A}(P)$ that is a possible match for the variedness and usefulness of the call site. Formally,

$$\mathbf{Varied}^-(I_c, \mathscr{A}) \subseteq \mathscr{A}_c.\mathcal{V}$$
$$\mathbf{Useful}^+(I_c, \mathscr{A}) \subseteq \mathscr{A}_c.\mathcal{U}.$$

If either the procedure P or the call site I_c have been marked for specialisation, there is always a perfect match, i.e.

$$\mathbf{Varied}^-(I_c, \mathscr{A}) = \mathscr{A}_c.\mathcal{V}$$
$$\mathbf{Useful}^+(I_c, \mathscr{A}) = \mathscr{A}_c.\mathcal{U}.$$

If there is no perfect match, some unnecessary computations or initialisations are made in the derivative code, which is the behaviour of AD without specialisation. One could try to find the best $\mathscr{A}_c \in \mathbb{A}(P)$ to minimise the cost that arises from superfluous derivative code, which would require a metric for said cost (in terms of memory, CPU time etc.) given by some function

$$\text{cost}(\, \mathscr{A}_c.\mathcal{V} \setminus \mathbf{Varied}^-(I_c, \mathscr{A}) \cap \mathscr{A}_c.\mathcal{V}, \quad \mathscr{A}_c.\mathcal{U} \setminus \mathbf{Useful}^+(I_c, \mathscr{A}) \cap \mathscr{A}_c.\mathcal{U} \,).$$

This is however not always possible with static analysis, as the runtime and memory cost of the primal code as well as the generated derivative code may depend on input that is only known at runtime. This is the case most of the time in practical applications (e.g. if the input defines the problem size or the desired quality of the output). The implementation that was created in Tapenade connects each call to the derivative procedure with the perfectly matching activity pattern if it exists, or an arbitrary enclosing activity pattern otherwise.

The inference rules are concluded with the remaining cases for call sites that are not covered by (4.4). Prerequisite *isLiveForDiff*($\mathscr{A}$) is true if the current call produces any result that is useful for the differentiation pattern $\mathscr{A}$:

$$\frac{!\mathit{isActiveCall}(\mathscr{A}, \mathscr{A}_c) \qquad \mathit{isLiveForDiff}(\mathscr{A})}{\mathscr{A} \vdash \texttt{call}\ P\,(ARGS) \rightarrow \texttt{call}\ P\,(ARGS)}$$

$$\frac{!\mathit{isActiveCall}(\mathscr{A}, \mathscr{A}_c) \qquad !\mathit{isLiveForDiff}(\mathscr{A})}{\mathscr{A} \vdash \texttt{call}\ P\,(ARGS) \rightarrow \{\}}$$

The inference rules for reverse differentiation can be derived in a similar fashion from the equations shown in [74].

4.3 Complexity of multi-activity AD

Locally per procedure and per activity pattern, the analysis is the same with or without multi-activity AD. Hence, the cost of using multi-activity AD is at most the product of the cost of AD without multi-activity and the maximum number of activity patterns of any given procedure. Without multi-activity differentiation, forward and reverse-mode AD can be implemented with a runtime and memory usage that is linear in the size of the primal code, as each instruction is augmented with a limited number of derivative instructions. With multi-activity specialisation, the size of the differentiated code is bounded by the product of the primal code length and the number of activity patterns, while the runtime is linear in the size of the generated derivative code.

As a consequence, the complexity of multi-activity AD is determined by the number of activity patterns $\|\mathbb{A}(P)\|$ for any given procedure P. The most obvious bound can be understood as follows: Each procedure argument of P can be varied, useful, both varied and useful, or neither varied nor useful, a total of 4 possibilities. Hence, for a procedure with n_{args} arguments the number of activity patterns $\|\mathbb{A}(P)\|$ can not be larger than

$$0 \leq \|\mathbb{A}(P)\| \leq 4^{n_{args}(P)} \tag{4.5}$$

Fortunately, there is another bound that is often tighter in practice and can be defined recursively taking into account the number of call sites in procedures that dominate the current procedure, and any additional activity patterns that are explicitly requested by the user. The explicitly requested activity patterns are referred to as *differentiation heads* and denoted by $\mathbb{D}$. An upper bound for the number of activity patterns is thus given by

$$\|\mathbb{A}(P)\| \leq \min\left(4^{n_{args}(P)},\ \|\mathbb{D}(P)\| + \sum_{I_c \in C(P)} \|\mathbb{A}(I_c)\|\right) \tag{4.6}$$

$$\leq \min\left(4^{n_{args}(P)},\ \|\mathbb{D}(P)\| + \sum_{I_c \in C(P)} \|\mathbb{A}(caller(I_c))\|\right), \tag{4.7}$$

where $caller(I_c)$ is the procedure that contains the call site I_c. (4.7) states that any given instruction can only be specialised as often as the procedure in which it is contained, which is true by construction for multi-activity AD if only entire procedures are specialised. The number of specialisations for a procedure P can be at most the sum of all specialisations of call sites to P, plus the number of differentiation heads for P. Furthermore, it is bounded exponentially in the number of arguments for P, eventually resulting in (4.6). This means that the size of the specialised derivative code is at most the size of the general derivative code multiplied by the number of call sites for all routines and the number of differentiation heads, and in particular, that a code that has

only one call site for each procedure is not specialised at all if only one differentiation head is specified by the user.

4.3.1 Strategies for multi-activity differentiation

There is a tradeoff between the speed of the resulting derivative program, which can be reduced by creating specialised differentiated procedures for as many call sites as possible, and the size of the resulting derivative program, which is increased with the number of created differentiated procedures. This section shows different strategies to choose a set of activity patterns.

A set of activity patterns $\mathbb{A}(P)$ is defined for every procedure P, and it is required that (4.1) and (4.2) hold $\forall \mathscr{A} \in \mathbb{A}(P)$. To ensure the correctness of the differentiated program, for each call site I_c to P there must be at least one activity pattern $\mathscr{A} \in \mathbb{A}(P)$ with $\mathscr{A}.\mathscr{V} \supseteq \mathbf{Varied}^-(I_c)$ and $\mathscr{A}.\mathscr{U} \supseteq \mathbf{Useful}^+(I_c)$. For a given differentiation head D, similar to (4.1), $\mathbf{Varied}^-(I_0(P), D)$ must be equal to the set of independent variables $D.\mathscr{V}$ and the set of useful variables $\mathbf{Useful}^+(I_\infty(P), D)$ must be equal to the set of dependent variables $D.\mathscr{U}$. A user can specify more than one differentiation head for P, and the set of its differentiation heads is denoted as $\mathbb{D}(P)$.

In addition to the differentiation heads, the set of activity patterns for P can contain one or more patterns based on the activities of call sites to P. The procedure in which a particular call site $I_c \in C(P)$ is contained is called $caller(I_c)$ and may itself have multiple activity patterns. This leads to a set of activity patterns for I_c given by

$$\mathbb{A}(I_c) := \left\{(\mathscr{V}, \mathscr{U}) : \exists \mathscr{A} \in \mathbb{A}(caller(I_c)) : \mathscr{V} = \mathbf{Varied}^-(I_c, \mathscr{A}) \wedge \mathscr{U} = \mathbf{Useful}^+(I_c, \mathscr{A})\right\}.$$

Based on the activity patterns of call sites to P, one can define the set of activity patterns for P. An extreme approach is to include the exact matching activity pattern for each call site into $\mathbb{A}$, an approach referred to here as *specialize-all*. The corresponding set of activity patterns is $\mathbb{A}_s$. The other extreme is to create only one activity pattern that encloses all callsite activities, referred to as *generalise-all* and denoted by $\mathbb{A}_g$. Formally, $\mathbb{A}_s(P)$ can be defined as

$$\mathbb{A}_s{}^{int} := \bigcup_{I_c \in C(P)} \mathbb{A}(I_c)$$
$$\mathbb{A}_s(P) := \mathbb{D} \cup \mathbb{A}_s{}^{int}$$

where $\mathbb{A}_s{}^{int}$ is the set of activity patterns that was created due to internal call sites in the code. A user may explicitly define a differentiation head for a procedure P that matches some other activity pattern for P, that is, $\mathbb{D} \cap \mathbb{A}^{int} \neq \emptyset$. This does not affect the analysis and only one instance of this pattern is contained in $\mathbb{A}$.

In contrast, the generalise-all approach yields only up to one activity pattern for each procedure in addition to the differentiation heads. To give the user more flexibility, one can implement a strategy that defaults to generalise-all, but allows user-defined specialisations, for example through additional differentiation heads supplied as arguments to the AD tool, or through pragmas in the primal code that select specific call sites for specialisation. Both options were implemented in Tapenade in the course of this work. If $C_s(P)$ is considered to hold the call sites marked for specialisation, the set of activity patterns is given by

$$\begin{aligned}
{\mathbb{A}_{gs}}^{int} &:= \bigcup_{I_c \in C_s(P)} \mathbb{A}(I_c) \\
\mathcal{V}_{gn}^{int} &:= \bigcup_{I_c \in C(P) \setminus C_s(P)} \left(\bigcup_{\mathscr{A} \in \mathbb{A}(I_c)} \mathbf{Varied}^{-}(I_c, \mathscr{A}) \right) \\
\mathcal{U}_{gn}^{int} &:= \bigcup_{I_c \in C(P) \setminus C_s(P)} \left(\bigcup_{\mathscr{A} \in \mathbb{A}(I_c)} \mathbf{Useful}^{+}(I_c, \mathscr{A}) \right) \\
\mathbb{A}_g(P) &:= \mathbb{D} \cup {\mathbb{A}_{gs}}^{int} \cup \left\{ \left(\mathcal{V}_{gn}^{int}, \mathcal{U}_{gn}^{int} \right) \right\},
\end{aligned} \tag{4.8}$$

that is, the union of the set of specialised activity patterns for all marked call instructions, and the set containing the generalised activity pattern for all other call instructions. The final set of activity patterns for each procedure is

$$\mathbb{A}(P) = \begin{cases} \mathbb{A}_s(P) & \textbf{if } P \text{ specialised} \\ \mathbb{A}_g(P) & \textbf{else} \end{cases}, \tag{4.9}$$

where a given procedure is specialised if it was marked for specialisation by the user by the means of a command-line flag or pragma in the code, or if the specialise-all approach was chosen. It follows from the above equations that regardless of the specialisation method, there is always an exact matching activity pattern for each differentiation head that was specified by the user.

Both specialisation and generalisation have their advantages. The specialisation facilitates the best-possible activity analysis, leading to more efficient derivative code. On the other hand, there is a price to pay in terms of derivative code size and runtime of the differentiation tool, see Section 4.3.

4.4 Implementation and usage

The implementation of multi-activity AD required some changes in the analysis stage of Tapenade. As shown in [74], the bottom up sweep through the call graph that determines the differentiation dependency of each procedure is followed by a top-down sweep that propagates the activities through to the leafs of the call graph, starting from the

differentiation heads. This stage is now augmented with a loop over all activity patterns for each procedure.

Whenever a call site is encountered, the usefulness and variedness of that call site is determined and add the so-obtained activity pattern to the set of activity patterns of the called procedure. If in generalisation mode, all patterns that are not equal to a differentiation root are merged into just one activity pattern. Once all dominating procedures are treated, there is a complete set of activities for every procedure.

The specialisation can be switched on by using the `-specializeActivity` command line option, followed by the procedure names to be specialised, or by "%all%" if all procedures shall be specialised. In addition, a user can activate specialisation for any procedure by placing the pragma `!$AD SPECIALIZE` in front of the procedure definition. Likewise, a user can activate specialisation for any call site by placing the pragma in front of the call site. Finally, a user can ask for multiple differentiation heads for a procedure, e.g. `-head "F[_X](U,X)/(R) F[_U](U)/(R)"` causes the creation of two differentiation heads for procedure `F`, where `_X` is appended to the name of the differentiation of `F` with `U,X` as dependents and `R` as independent, likewise `_U` appended to the name of the differentiation with only `U` as dependent.

During code generation, it has to be decided which specialised version of each procedure is linked to each call site. This is simple in specialisation mode, where there is always exactly one activity pattern of the called procedure that matches the activity of the call site. During the analysis stage, the implementation stores a pointer to this matching pattern inside the data structure that represents a call site in the internal Tapenade code representation, this information is therefore available. In generalisation mode, this pointer points at the first pattern that was found or created during the analysis stage that contains the call site activity.

4.5 Case study

The effectiveness of multi-activity AD is studied by applying Tapenade on the residual subroutine in `stamps` that is presented in Section 2.4. Runtime and memory footprint of the adjoint solver with and without the usage of multi-activity AD is evaluated on the CPU system using the RAE2822 test presented in Chapter 3.

Two setups are compared. The first, referred to as *specialised* setup, uses two differentiation heads for the residual procedure, resulting in the two differentiated procedures *residual_b_X*($\vec{R}\downarrow$,$\bar{\vec{R}}\uparrow$,$\vec{X}\uparrow$,$\bar{\vec{X}}\downarrow$,$\vec{U}\uparrow$,$\bar{\vec{U}}\downarrow$) and *residual_b_U*($\vec{R}\downarrow$,$\bar{\vec{R}}\uparrow$,$X\uparrow$,$\vec{U}\uparrow$,$\bar{\vec{U}}\downarrow$). They are used in the adjoint solver as shown in Algorithm 1. In addition, the `specializeActivity %all%` command line flag is used to create specialised procedures wherever possible. The second, called *generalised* setup, does not make use of the

	diff time	diff code lines	diff code chars	executable size
general	6.270s	22122	880514	1.8MB
special	8.394s	34728	1350785	2.2MB
change	**34%**	**57%**	**53%**	**22%**

Table 4.1: Differentiation time and resulting code size of Tapenade applied to `stamps`.

	total runtime	relative time	residual runtime	relative time	peak memory	relative memory
primal	51.15s	1	38.71	1	360.93MB	1
general	247.14s	4.83	239.56	6.2	431.68MB	1.196
special	186.5s	3.65	177.51	4.6	432.62MB	1.199
change	**-25%**		**-26%**		**0.2%**	

Table 4.2: Runtime of primal flow computation in `stamps`, as well as adjoint computation time with and without multi-activity AD.

multi-activity approach. It uses one differentiation head for the residual procedure, yielding *residual_b*($\vec{R}\downarrow, \vec{\bar{R}}\uparrow, \vec{X}\uparrow, \vec{\bar{X}}\downarrow, \vec{U}\uparrow, \vec{\bar{U}}\downarrow$). This is used instead of *residual_b_X* and *residual_b_U* throughout the code. None of the other subroutines are specialised.

The time it takes for Tapenade to complete the differentiation of the code, and the size of the resulting derivative code, is shown in Table 4.1. The specialisation results in an increase in code size of more than 50%, which manifests itself in a slightly larger executable file. The relative increase is smaller for the binary file, since it includes also non-differentiated procedures and statically linked libraries. The runtime of Tapenade only increases by a third, which is less than the linear worst-case complexity with respect to output code size predicted in Section 4.3.

The total runtime measured with `gprof` and peak memory usage measured by reading the reported high water mark from `/proc/PID/status` for the RAE2822 case on the CPU system is reported in Table 4.2. The adjoint solver runs about a quarter faster in the specialised setup than in the generalised setup. There is no considerable change in the memory usage (half of the difference is caused by the size of the executable itself). It is also worth considering the relative runtime compared to the primal solver. In the specialised setup, the adjoint solver takes 4.6 times longer per iteration, the general solver needs more than 6 times longer than the primal solver per iteration.

To understand how multi-activity AD achieves the observed speedups, some main functionalities are identified, for which timings are measured separately. To give an example: The turbulence model consists of a main procedure that calls several helper procedures. The time spent in all these is summed up and referred to under the name

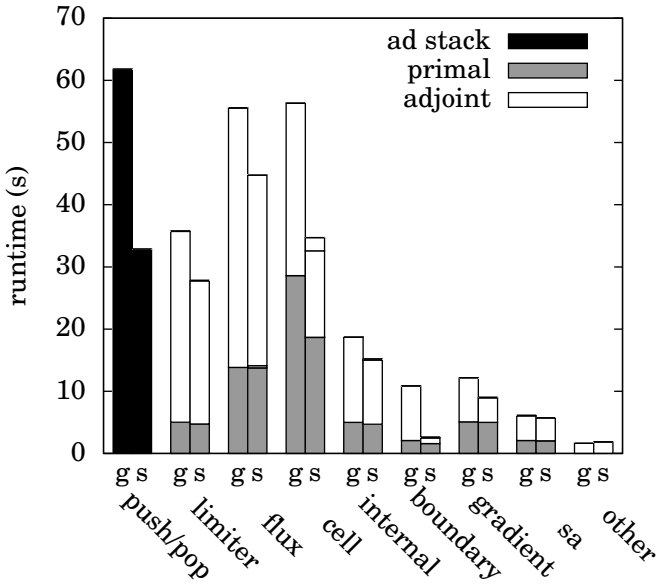


Figure 4.1: Runtimes of `stamps` functionalities for the specialised and generalised adjoint and primal. Bars marked with 'g' show timings of generalised setup, those with 's' show specialised setup. Specialisation improves the timings of all functionalities, especially of push/pop and cell gradients.

turbulence. Other functionalities that are timed separately are *internal, boundary, residual, gradient, limiter* and *cell correction*. Only the *self time* of all these functionalities is considered, i.e. time spent inside the flux calculation is already encountered for inside *flux* and therefore does not contribute to the timing of *internal* or *boundary*, from which flux is called. The call graph that is obtained by clustering all utility procedures together into these core functionalities is shown in Figure 4.2, along with the call graph of the clustered adjoint code obtained in the specialised setup.

The specialised setup has a significantly lower runtime for all these functionalities, as is shown in detail in Figure 4.1. Taken together, the primal and adjoint residual procedure and all procedures dominated by the former contribute 177s to the runtime in specialised mode and 240s in generalised mode. About 10s are spent in other procedures such as libraries, linear solvers etc. for both specialised and generalised setup. For all measured core functionalities in `stamps`, one can observe an overall reduction in runtime if multi-activity AD is used. The main contribution to this speedup is always the reduction of time spent within differentiated procedures and the push/pop mechanism that stores primal intermediate values on the stack and makes them available in reverse order during the adjoint computation.

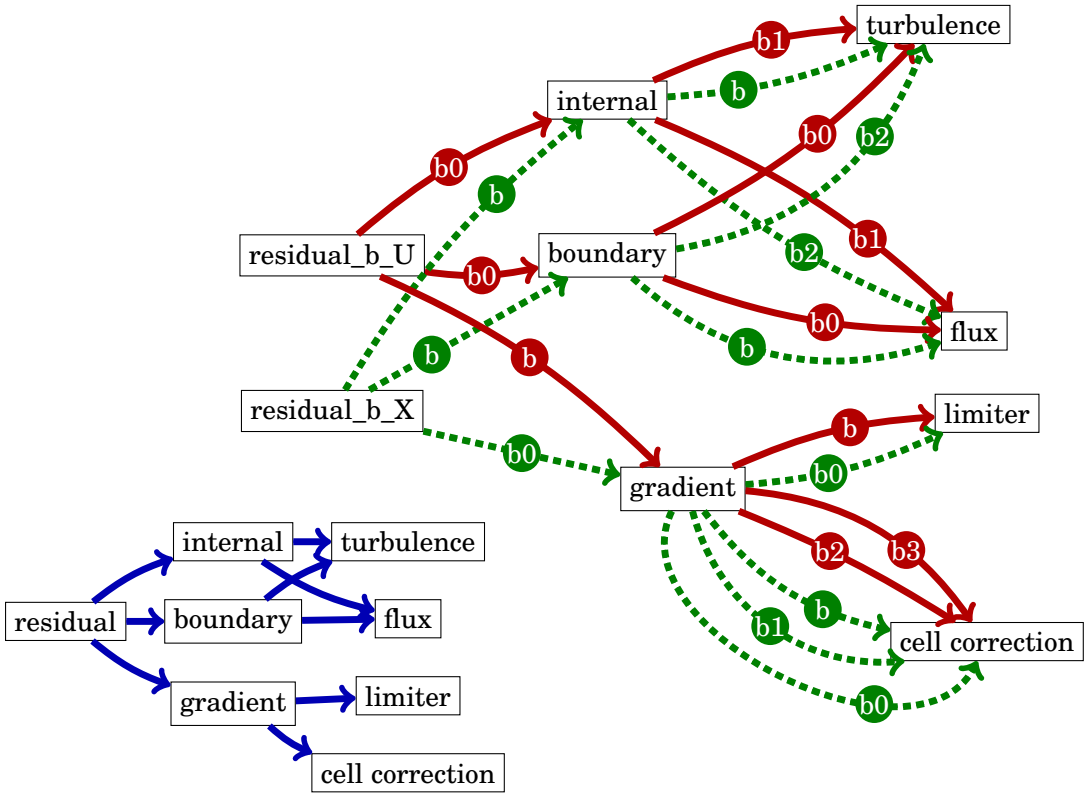


Figure 4.2: Primal (bottom left) and differentiated call graphs (top right). Each arrow in the differentiated call graph represents a specialised differentiated procedure that is named with the suffix indicated on the arrow. We can identify two main streams: Code dominated by `residual_b_U` (lines) and `residual_b_X` (dashes).

4.6 Problem case

Multi-activity differentiation may not always pay off. Obviously there can be primal programs where each routine is only called in one location, in which case the derivative code will be the same with or without using the multi-activity approach, but in these cases, the differentiation time and derivative program size will also be the same. There may also be cases where multi-activity differentiation helps to reduce the number of operations, but the reduction will be too small to be noticeable in the overall runtime. More interestingly, there are cases where multi-activity differentiation does create specialised routines, but fails to reduce the number of operations in the derivative code.

This happens if the activity of two different specialisations differs only after the last or before the first instruction, but is the same everywhere inside the subroutine. For example, consider the case where the last instruction in a subroutine is the assignment $v \leftarrow w$, where v is a useful subroutine argument at all call sites, and w is also a subroutine argument, but is only useful at some call sites. This will result in the creation of two

specialised versions of that subroutine that perform the same operations internally, except that $\bar{w}$ is set to zero at the start of the adjoint computation in one version but not the other. Without multi-activity differentiation, only one adjoint subroutine is created that does not set $\bar{w}$ to zero, and instead, the zeroing has to be performed at the call sites. In this case, multi-activity differentiation merely shifts the zeroing of $\bar{w}$ to a different location in the code. A complete working example for this is shown in Appendix C.

4.7 Specialisation on the flow graph

The multi-activity differentiation concept can be extended to not only create specialised routines, which is essentially a duplication of nodes in the call graph, but also specialised basic blocks, that is, cloning nodes in the flow graph. For example, the variedness of code following an `if`-block and the usefulness of code preceding an if-block may be different depending on which branch is executed. Similarly, the activity may depend on the number of iterations of some loop in the code. Consider as an example the following loop, in a context with independent variable v_3 and dependent variable v_1.

```
1   do i=1,n
2     v1 = v1+v2
3     v2 = v2+v3
4   end do
```

A straightforward differentiation without multi-activity must consider `v1`, `v2` and `v3` as active. However, before the loop body is executed for the first time, only `v3` is varied and `v2` becomes varied. In the second iteration, `v2` and `v3` are varied and `v1` becomes varied. Hence, the variedness changes with each iteration. Likewise, `v2` only becomes useful if at least one iteration is performed, and `v3` only becomes useful after at least two iterations. An AD tool may try to take advantage of this effect and specialise the code before and after the loop depending on the total number of iterations, and the code within the loop depending on the current iteration.

This approach appears to be less promising. While the increase in derivative code size for multi-activity on the call graph is bounded as discussed in Section 4.3, and was found to be small in practice, a multi-activity strategy on the flow graph may increase the derivative code size by a factor that is bounded only by the number of different execution paths of the program, which is exponential in the number of branches and loops in the code. Furthermore, it would greatly reduce the readability of the derivative code, as it would require loop unrolling and duplication of code sections that, unlike subroutines, are often not easily identifiable by a user.

4.8 Specialisation and encapsulation

Object oriented languages use encapsulation to hide internal implementation details and data from users of a class, and to provide a well-defined interface that remains fixed regardless of changes to internal implementation details. Source transformation AD without multi-activity differentiation conflicts with this in both forward and reverse-mode, which will be demonstrated using the following code example.

```
1 class M {
2   private double w;
3
4   public void a(double* v1) {
5     double v2 = (*v1);
6     w = w + (*v1);
7     f(v1,v2);
8   }
9   public void f(double* v1, double v2) {
10    (*v1) = (*v1)*v2*w
11  }
12 };
```

A user differentiating this code without multi-activity differentiation, with two differentiation heads `a` and `f`, using `v1` as the only independent and dependent variable, will expect a procedure with signature `f_d(v1,v1d,x2)`. However, an AD tool must generate instead the more general procedure `f_d(v1,v1d,v2,v2d)`, due to the call to `f` from within `a`. Therefore, the signature of public derivative procedures may depend on internal implementation details, and the success of the activity analysis of the class `M`.

Worse, still, is that the user may at some point wish to use `f_d` from outside of `M`. To obtain a correct derivative, `wd` needs to be set to zero before calling `f_d`. This requires that the derivative variable `wd` is accessible from outside, for example by making it public, even though the primal variable w is marked as private. As a result, an AD tool would need to make most derivative variables public, even if their primal counterparts were private, and a user calling derivative routines in `M` will need to interact with them. Furthermore, code outside of `M` may need to be updated whenever `M` is changed internally.

If source-transformation of object oriented code is to be implemented in AD tools, multi-activity differentiation can be a way to preserve the encapsulation properties of objects, and to guarantee that publicly used differentiated functions retain their signature regardless of internal implementation details that may require additional, more general differentiated versions of the same function.

CHAPTER 5

Adjoint OpenMP for stencil computations

Shared-memory parallelisation is essential for high performance computing on modern computers. Although methods for automatic parallelisation of unstructured computations exist [71, 133], manual parallelisation often yields better performance and is common in CFD applications. When applying AD to codes with carefully hand-optimised parallelism, it is desirable to replicate their parallel behaviour in the adjoint code.

Differentiation of a primal code that is inherently free of race conditions can still lead to race conditions in the adjoint code. This occurs if threads in the primal program read concurrently from the same active variable. This was observed previously [49], and the *exclusive read property* was presented that guarantees the absence of race conditions in the adjoint code. An AD tool that is unable to detect exclusive read access must safeguard the adjoint code against race conditions, which is detrimental to performance.

The detection of exclusive read access at compile time is impossible in general. A special type of code structure that satisfies the exclusive read property is identified in Section 5.2, which is of interest as it is relatively easy to detect, and is commonly responsible for a large fraction of the runtime of unstructured CFD solvers. The method presented in this chapter, referred to as SSMP for *symmetric shared memory parallelisation*, is exploiting this special case. SSMP is applied to the residual computation in `stamps` and tested on the BEND case using the CPU and MIC systems described in Chapter 3. The adjoint code generated with SSMP is shown to exhibit the same scaling characteristics as the primal code. Although the Tapenade AD tool was used, SSMP was implemented using a postprocessing script that can in principle be used with other source transformation AD tools as well, irrespective of the AD tool's support for OpenMP.

Following this, another special case that does not satisfy the exclusive read property, yet can still be reverse-differentiated while preserving parallelism, is presented in Section 5.3. Finally, rules for the differentiation of code that can not be differentiated without added safeguards against race conditions are summarised in Section 5.4.

5.1 The exclusive read property

This section formalises the condition under which write conflicts in the adjoint code arise. For this, the following property as introduced in [49] is useful. Note that an improved presentation of the results in this section is also included in [82].

Definition 1. Exclusive read. A piece of code within synchronisation barriers (such as the body of a parallel loop) is said to have exclusive read access if no two threads can read from any given memory location at the same time.

Lemma 1 states a property of the relationship between primal and adjoint memory access. This is used to show in Proof 1 that exclusive read access in the primal code ensures that the adjoint is free of race conditions in agreement with the proof in [49].

Lemma 1. Let K be a loop iteration reading from a vector $\vec{U}$ at the set of indices $M_{in}(\vec{U})$ and writing to a vector $\vec{R}$ at indices $M_{out}(\vec{R})$. Further, let $\bar{K}$ be an iteration of the corresponding adjoint loop reading from the adjoint vector $\bar{\vec{R}}$ at the set of indices $\bar{M}_{in}(\bar{\vec{R}})$ as well as the primal vector $\vec{U}$ at indices $\bar{M}_{in}(\vec{U})$ and writes to the adjoint vector $\bar{\vec{U}}$ at indices $\bar{M}_{out}(\bar{\vec{U}})$. The following holds.

- $\bar{M}_{out}(\bar{\vec{U}}) \subseteq M_{in}(\vec{U})$
- $\bar{M}_{in}(\bar{\vec{R}}) \subseteq M_{out}(\vec{R})$
- $\bar{M}_{in}(\vec{U}) \subseteq M_{in}(\vec{U})$

The lemma becomes obvious if the following is considered.

Proof 1. If K does not write to an index k_m in $\vec{R}$, that is, $k_m \notin M_{out}(\vec{R})$, then the contribution of K to $\vec{R}_{k_m}$ is zero, and thus $\frac{\partial K_{k_m}}{\partial \vec{U}_{k_n}} = 0$ for every index k_n of the input array $\vec{U}$. The Jacobian ∇K of K therefore contains only zeroes in row k_m. Likewise, if K does not read from index k_n in $\vec{U}$, namely, $k_n \notin M_{in}(\vec{U})$, the Jacobian of K contains only zeroes in column k_n. Since the adjoint of K implements

$$\bar{K} : \bar{\vec{U}} \leftarrow (\nabla K)^T \cdot \bar{\vec{R}},$$

a zero column k_n in ∇K will become a zero row k_n in $(\nabla K)^T$, and therefore the linear operator $\bar{K}$ will make no contribution to index k_n. Similarly, a zero row k_m in ∇K will mean that $\bar{K}$ does not read from index k_m.

With this in mind, we can state the following.

Lemma 2. The adjoint derivative of a loop whose body has exclusive read access can be executed in parallel and does not suffer from write data races.

This is shown in the following proof.

Proof 2. Assume that a primal that satisfies the exclusive read property has an adjoint code with race conditions. These must be caused by more than one adjoint thread writing to the same memory location in parallel. Following lemma 1, this means that the primal code must access the corresponding primal memory location from multiple threads within the same parallel region, which violates the exclusive read property.

5.2 Symmetric memory access

Symmetric memory access is a term used in this work to describe code which uses the same set of indices to read from input arrays and write to output arrays. If a parallel loop has a body with symmetric memory access, any correct parallelisation of the primal loop is also a valid parallelisation of the resulting adjoint loop. This is illustrated with an example, and afterwards formalised.

5.2.1 Memory access in unstructured CFD

The flux computation in unstructured CFD solvers as shown in Algorithm 3 is a loop where in each iteration, data is read from indices in the state vector that correspond to the two nodes adjacent to an edge, and the residual is updated at those same indices. This makes unstructured CFD solvers an example of code with symmetric memory access, whose parallelisation can be preserved during reverse differentiation using SSMP. See

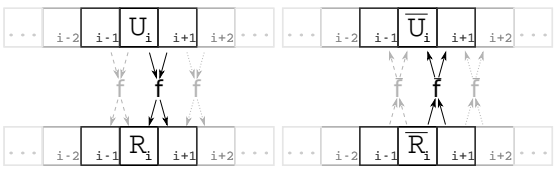


Figure 5.1: Symmetric memory access: The reversal does not change the memory access pattern. The parallelisation of the primal code can be applied verbatim to the adjoint code, as shown in Section 5.2.

Figure 5.1 for an illustration of the memory access pattern. In this section, SSMP is formalised and its correctness shown.

In the absence of user intervention, SSMP, or any other strategy to detect exclusive read access, a source-transformation AD tool applied to the computation in Algorithm 3 generates an adjoint code as shown in Algorithm 4.

for $c \leftarrow n_{colours} \dots 1$ **do**
  foreach $e \in edges(c)$ **do**
    $l, r \leftarrow edg2nde(e)$;
    $[\bar{\vec{U}}_l, \bar{\vec{U}}_r] \overset{+}{\leftarrow} \bar{f}_l(\vec{U}_l, \vec{U}_r) \cdot \bar{\vec{R}}_l$;
    $[\bar{\vec{U}}_l, \bar{\vec{U}}_r] \overset{+}{\leftarrow} \bar{f}_r(\vec{U}_l, \vec{U}_r) \cdot \bar{\vec{R}}_r$;
  end
end

Algorithm 4: Adjoint residual computation (reverse-differentiated Algorithm 3) as a loop where each iteration computes the adjoint flux $\bar{f}$ along edge e, reading $\vec{U}$ and $\bar{\vec{R}}$ and writing $\vec{U}$ at the indices l and r corresponding to the nodes connected by e.

5.2.2 Parallelisation strategy

By definition [117], the adjoint of residual function (2.4) must be an implementation of

$$\bar{\vec{U}} = (\nabla F)^T \cdot \bar{\vec{R}},$$

and is called $\bar{F}$, with

$$\bar{F}(\vec{U}, \bar{\vec{R}}) := (\nabla F(\vec{U}))^T \cdot \bar{\vec{R}}.$$

The following definition applies to the primal and adjoint residual loops in Algorithm 2 and Algorithm 4.

Definition 2. Symmetry. The memory access of a code is *symmetric* if the set of indices accessed in the input vector is equal to the set of indices accessed in the output vector, that is, $M_{in}(\vec{U}) = M_{out}(\vec{R})$. In addition, each memory location is either only read from, only incremented, or only written to during the entire parallel loop.

Loops whose iterations have symmetric memory access can be differentiated using SSMP, as stated in the following Lemma.

Lemma 3. If a parallel loop writes a result into a shared output vector without race conditions, and the loop body has symmetric memory access, then the adjoint of this loop can use a shared output vector and be executed in parallel without race conditions.

The correctness of SSMP can be shown with the following proof by contradiction.

Proof 3. Assume that there are race conditions in the generated adjoint code, caused by more than one thread writing to the same location at the same time. It then follows from Lemma 1 that the primal has concurrent read access to the input vector (thereby violating the exclusive read property).
Any code with symmetric memory access that violates the exclusive read property must also perform concurrent write access to some index of its output vector. Thus, the primal code can not be free of race conditions. That contradicts the assumption in Lemma 3. It follows that a code that is free of race conditions and has symmetric memory access also has an adjoint code that is free of race conditions.

To implement SSMP, loops with symmetric memory access must be identified. This is a manual step in this work, which requires marking such loops with a tag. A postprocessing script deploys the correct OpenMP pragmas in the generated adjoint code for all adjoint loops whose corresponding primal loops are marked as having symmetric memory access.

5.2.3 Illustrating example

If the code to implement this function is generated by AD as shown in Algorithm 4, and used on the mesh shown in Figure 2.3, $\bar{F}$ can be written as the sum of its iterations as

$$
\begin{aligned}
\bar{F} = &\begin{bmatrix} \frac{\partial f_l}{\partial \vec{U}_1} & 0 & 0 & \frac{\partial f_r}{\partial \vec{U}_1} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{\partial f_l}{\partial \vec{U}_4} & 0 & 0 & \frac{\partial f_r}{\partial \vec{U}_4} \end{bmatrix} \cdot \bar{\vec{R}} + \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & \frac{\partial f_l}{\partial \vec{U}_2} & \frac{\partial f_r}{\partial \vec{U}_2} & 0 \\ 0 & \frac{\partial f_l}{\partial \vec{U}_3} & \frac{\partial f_r}{\partial \vec{U}_3} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \cdot \bar{\vec{R}} \\
+ &\begin{bmatrix} \frac{\partial f_l}{\partial \vec{U}_1} & \frac{\partial f_r}{\partial \vec{U}_1} & 0 & 0 \\ \frac{\partial f_l}{\partial \vec{U}_2} & \frac{\partial f_r}{\partial \vec{U}_2} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \cdot \bar{\vec{R}} + \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{\partial f_l}{\partial \vec{U}_3} & \frac{\partial f_r}{\partial \vec{U}_4} \\ 0 & 0 & \frac{\partial f_l}{\partial \vec{U}_3} & \frac{\partial f_r}{\partial \vec{U}_4} \end{bmatrix} \cdot \bar{\vec{R}} \\
+ &\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & \frac{\partial f_l}{\partial \vec{U}_2} & 0 & \frac{\partial f_r}{\partial \vec{U}_4} \\ 0 & 0 & 0 & 0 \\ 0 & \frac{\partial f_l}{\partial \vec{U}_2} & 0 & \frac{\partial f_r}{\partial \vec{U}_4} \end{bmatrix} \cdot \bar{\vec{R}}
\end{aligned}
\tag{5.1}
$$

where $\bar{F}$ performs read operations in $\bar{\vec{R}}$ and write operations in $\bar{\vec{U}}$ at the same indices as the primal function F in $\vec{U}$ and $\vec{R}$. $\bar{F}$ depends on the primal variable $\vec{U}$ if f_l or f_r are

nonlinear in $\vec{U}$, in which case the evaluation of $\bar{F}$ further requires read access in $\vec{U}$. Note how the functions in equations (2.22) and (5.1) read and write using the same memory locations and can use the same colouring to avoid race conditions.

5.2.4 Implementation

A technique known as *subroutine technique* [102] or *function outlining* [100] is used to implement variable scoping, that is, to declare each variable as private or shared. To this end, each body of a parallel primal loop with symmetric memory access is placed into a dedicated subroutine. If the compiler uses function inlining during compilation, this has no effect on the final performance.

Variables that need to be shared are passed to that subroutine using call-by-reference, and the `!$OMP PARALLEL DO DEFAULT(shared)` construct is used to make all visible variables shared between all threads by default. Variables that need to be private are defined as local variables inside the subroutine into which the loop body was placed, thereby implicitly declaring them as private as per the OpenMP standard. Since the adjoint is generated by AD such that each primal subroutine argument has a corresponding adjoint argument, and each local variable has a corresponding local adjoint, it is ensured that the scope of adjoint variables is identical to that of their corresponding primal variables.

All access to AD-related libraries must be rerouted to a thread-safe implementation. For example, the push/pop mechanism used by Tapenade to store intermediate results and branch decisions needs to be modified to create a private stack for each thread. It is then necessary to ensure that the corresponding primal and adjoint iterations are executed on the same thread, and that the same scheduling and the same number of threads is used for the primal and adjoint computations.

Placing the necessary OpenMP pragmas in the adjoint code, as well as modifying the calls to the stack mechanism, is in this work performed by a python script that post-processes the adjoint code generated by Tapenade.

If the primal code performs symmetric memory access to some array and conflicting access to another array or some global variable or common block, SSMP can be used for the symmetrically accessed arrays, but conventional approaches as summarised in Section 5.4 have to be used for the latter. Ensuring the absence of concurrent read access in the primal code is not trivial, especially in the presence of aliasing. In contrast, symmetry is relatively easy to detect if it is implemented by using the same expressions as indices in the input and output arrays.

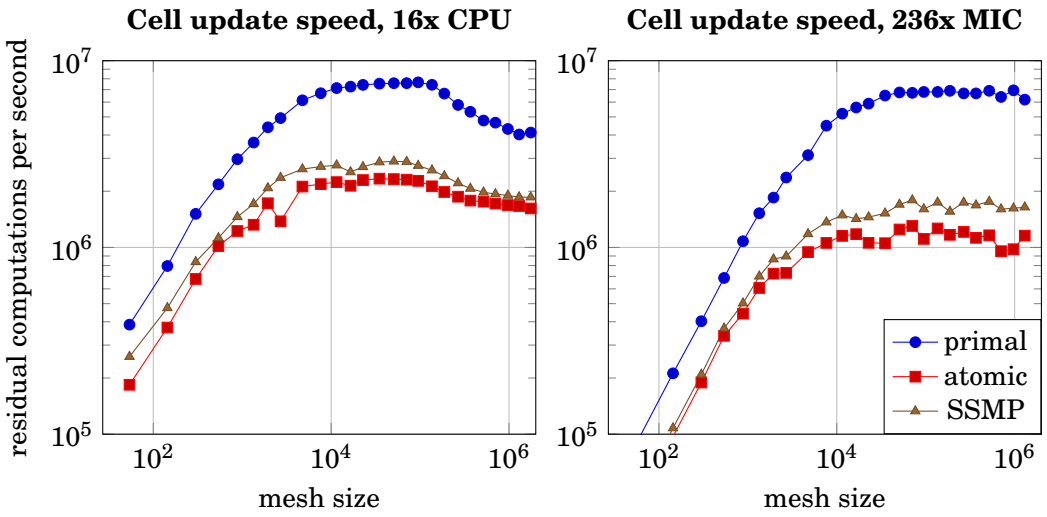


Figure 5.2: Cell residual computations per second (higher is better). BEND mesh with various mesh sizes from 27 to 1.5M cells. **16 CPU threads, left:** The performance is best for mesh sizes between 10,000 and 100,000 cells. For smaller meshes, there are not enough edges per colour, and synchronisation overheads lead to performance degradation. Larger meshes do not fit into the 20MB L3 cache. **236 MIC threads, right:** The performance is best for mesh sizes above 10,000 cells. For smaller meshes, synchronisation overheads lead to performance degradation just like on the CPU. The MIC does not have a L3 cache.

5.2.5 Test cases and results

To demonstrate its effectiveness, SSMP is applied to `stamps`, and the generated parallel adjoint solver is applied to the BEND test case on the CPU and MIC system. The timings of the adjoint solver created with SSMP are compared to that of an adjoint solver that was manually parallelised using `atomic` pragmas.

Timings are measured for the internal residual computation only, excluding all I/O, linear solvers and temporal discretisation, because they are not parallelised with SSMP. The baseline for the scalability of the primal solver on a CPU is the runtime of the serial primal solver on the CPU. For the scalability on the MIC, the baseline timing is from the serial primal solver on the MIC. Likewise, the baseline timing for the adjoint scalability is the runtime of the serial adjoint solver on the respective machine. The scaling for SSMP and atomic approaches use the same baseline.

5.2.5.1 Execution speed

The *execution speed*, which is obtained by dividing the measured program runtime by the number of iterations and the number of mesh edges, is used as a metric for the performance of the generated adjoint solver. The execution speed allows a comparison of

runtimes between different mesh sizes, regardless of the number of iterations. In the absence of caching effects and synchronisation barriers, the execution speed should be the same for all test cases.

CPU 16 threads	SSMP	Atomic	Difference
BEND	1,860,000	1,610,000	13.4%
MIC 236 threads	SSMP	Atomic	Difference
BEND	1,990,000	1,430,000	28.1%

Table 5.1: Number of edge residual updates per second (computational speed) for BEND with 600,000 nodes. The SSMP approach increases speed by $12-29.7\%$.

In reality one observes a different execution speed for different mesh sizes. On the CPU, the execution speed is significantly lower if the number of nodes in the mesh is below 10,000, since smaller meshes do not allow enough parallelism to compensate for the serial parts of the code and the overhead caused by OpenMP barriers after each colour. The execution speed for a series of BEND meshes with increasing size is shown in Figure 5.2 for the CPU and the MIC, where the speed difference for different mesh sizes is even more pronounced. A drop in execution speed can be observed on the CPU when the mesh size exceeds 100,000 nodes, which is probably caused by the fact that the state and residual vectors no longer fit into the $L3$ cache of the CPU. On the MIC there is no such effect, probably because of the lack of shared $L3$ cache.

One can observe a roughly constant factor in speed difference between the adjoint solver with atomic constructs and that with SSMP for all mesh sizes. For the BEND case with 600,000 nodes, execution speed is listed in Table 5.2. The overhead of using atomic pragmas has a larger influence on the MIC than on the CPU, probably because of the lower serial speed of the MIC and the higher cost of synchronisation barriers for a large number of threads. The effect of removing atomic pragmas is significant, with a speed improvement due to SSMP of more than 10% on the CPU and more than 25% on the MIC system in all test cases.

5.2.5.2 Scalability

The scaling is shown in Figure 5.3 for the BEND case with 600,000 nodes, and shows a significant performance penalty if atomic pragmas are used. The speedups generally agree with previous studies on OpenMP in unstructured CFD solvers [26, 27]. The adjoint code generated with SSMP consistently scales better than the adjoint solver with atomic pragmas as well as the primal solver. While it is not surprising that the adjoint code runs faster without atomic pragmas, some explanation is needed as to why the adjoint scales better than the primal.

CPU	SSMP	Atomic	Primal
BEND	9.6	8.4	8.7
MIC	SSMP	Atomic	Primal
BEND	138.2	99.4	124.2

Table 5.2: Speedup factors for primal, adjoint with SSMP, and adjoint with atomic pragmas, compared with serial execution on the same machine with the same test cases. Data shown for BEND case with 600,000 vertices, 16 CPU and 236 MIC threads.

First, it is worth noting that the speedup of the parallel adjoint is computed relative to the speed of the serial adjoint, whereas the speedup of the parallel primal is relative to the speed of the serial primal. Hence, the better scaling hides the fact that the adjoint solver is still significantly slower than the primal solver in absolute numbers. Second, a better scalability on the MIC does not indicate a better absolute runtime than on the CPU. This is the reason why absolute runtimes are discussed in Section 5.2.5.1.

The good scalability of the adjoint can be explained by the fact that the adjoint solver performs more operations than does the primal for the same number of synchronisation barriers. The higher computational cost hides the parallelisation overhead. On the MIC, the performance penalty of using atomic pragmas is more pronounced than on the CPU, which is not surprising since the overhead of synchronisation constructs grows with the number of threads. The poor speed on the MIC system is typical for first-generation XeonPhi coprocessors [41], especially on unstructured meshes. Section 5.3.4 presents a similar application on a structured mesh, where the MIC performs better. SSMP affects only the AD process and is orthogonal to code optimisation that is undertaken to speed up the code in the future on either MIC or CPU platforms.

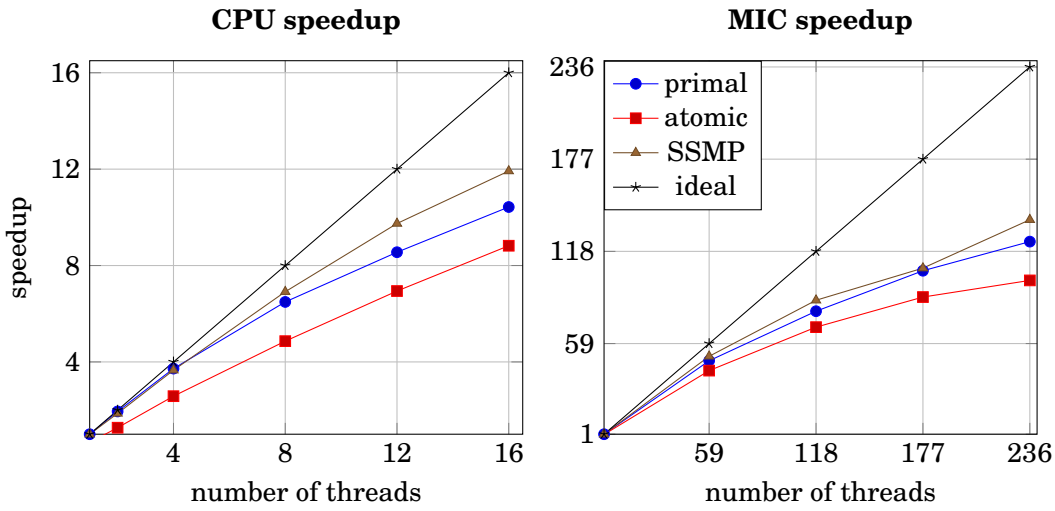


Figure 5.3: **Left:** Scaling of primal and adjoint residual for BEND with 600,000 nodes on 16 CPU threads, with maximum speedup of 9.6 (SSMP), 8.4 (atomic), and 8.7 (primal) as in Table 5.2. **Right:** Scaling on 236 MIC threads, with the maximum speedup of 138.2 (SSMP), 99.4 (atomic), and 124.2 (primal) as in Table 5.2.

5.3 Globally symmetric, locally asymmetric access

Straightforward application of AD to loops that do not satisfy the exclusive read property generally results in an adjoint code that suffers from race conditions. However, a modified reverse-differentiation strategy is presented in this section that can in some cases still yield a code that is parallelisable in the same way as the primal. This method is referred to as RSMP, for *reflexive shared-memory parallelism*. RSMP can be applied to codes that, instead of the exclusive read property, have certain other properties described in Section 5.3.2. Many structured-mesh solvers fit into this category, and the challenge in parallelising their adjoints is illustrated in Section 5.3.1.

5.3.1 Memory access in structured solvers

The spatial discretisation of some structured PDE solvers is implemented as a loop in which each iteration computes all contributions to a single index ζ in the output vector, by gathering data from all neighbouring control volumes in the mesh. In the remainder of this chapter, ζ is used to denote the control volume that is central to the current iteration, and its neighbours are assumed to be stored in a set $nde2neigh(\zeta)$, where any given element of the set of neighbours can be referred to as v. A general implementation of a structured solver that implements such a behaviour is shown in Algorithm 5.
The outer loop in Algorithm 5 can be executed in parallel, as within each outer iteration, write access is performed exclusively to one element in the output vector that

```
!$OMP parallel for shared(Ū,R̄) private(v1,v2,w,ζ,ν);
for ζ ∈ 1...n_nodes do
    R̄_i ← 0;
    foreach ν ∈ nde2neigh(ζ) do
        v1 ← Ū_ν;
        v2 ← Ū_ζ;
        f(v1 ↓, v2 ↓, w ↑);
        R̄_i ← R̄_i + w;
    end
end
```

Algorithm 5: Loop over mesh that pulls data to center node ζ from adjacent nodes *nde2neigh*(ζ). The outer loop can be executed in parallel. The inner loop is often unrolled in practice.

corresponds to that iteration. Depending on the function f, the loop structure can be used to implement a variety of CFD-related algorithms. For instance, if each element $\vec{U}_\zeta$ is considered to be a pair (v_ζ, x_ζ) containing a flow state v and a coordinate x at the center of control volume ζ, Algorithm 5 becomes a central finite difference discretisation of Burger's equation with the following definition of f:

$$f(\vec{U}_\nu, \vec{U}_\zeta) := \frac{v_\zeta \cdot (v_\nu - v_\zeta)}{2 \cdot (x_\nu - x_\zeta)} \tag{5.2}$$

By choosing a different f, one obtains instead the discretised residual of a linear diffusion equation, for example a heat equation with constant conductivity κ

$$f(\vec{U}_\nu, \vec{U}_\zeta) := \kappa \frac{v_\nu - v_\zeta}{(x_\nu - x_\zeta)^2}. \tag{5.3}$$

Yet another choice of f yields first-order spatial gradients following the Green-Gauss theorem (2.9):

$$f(\vec{U}_\nu, \vec{U}_\zeta) := \frac{S_\nu}{2 \cdot \|\Omega\|_\zeta} \cdot (v_\nu + v_\zeta) \tag{5.4}$$

Regardless of the exact choice of f, due to the overlapping read access in the primal code, the exclusive read property is violated, and hence the adjoint code (shown in Algorithm 6) can not be easily parallelised. A graphic illustration of this problem is shown in Figure 5.4.

5.3.2 Parallelisation strategy

RSMP can be applied to primal loops where each iteration satisfies the following reflexivity, separability, permutability and purity conditions.

$\bar{\vec{U}} \leftarrow 0;$
for $\zeta \in n_{nodes} \dots 1$ **do**
  foreach $v \in nde2neigh(\zeta)$ **do**
    $v_1 \leftarrow \vec{U}_v;$
    $v_2 \leftarrow \vec{U}_\zeta;$
    $\bar{w} \leftarrow \bar{\vec{R}}_\zeta;$
    $\bar{f}(v_1 \downarrow, \bar{v}_1 \uparrow, v_2 \downarrow, \bar{v}_2 \uparrow, w \uparrow, \bar{w} \downarrow);$
    $\vec{R}_\zeta \leftarrow \vec{R}_\zeta + w;$
    $\bar{\vec{U}}_v \leftarrow \bar{\vec{U}}_v + \bar{v}_1;$
    $\bar{\vec{U}}_\zeta \leftarrow \bar{\vec{U}}_\zeta + \bar{v}_2;$
  end
end

Algorithm 6: Adjoint of Algorithm 5, which must be executed in serial, because of the highlighted adjoint statements that cause concurrent write access.

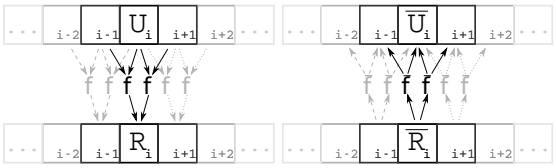


Figure 5.4: Reversal of memory access during reverse-differentiation: If a variable is read from in the primal code, then its corresponding adjoint variable is written to. The overlapping read access in the primal causes overlapping write access in the adjoint.

Definition 3. Reflexivity. Let a loop compute a function $\vec{R} \leftarrow F(\vec{U})$, and let each iteration in that loop write to an index ζ in the output vector $\vec{R}$ and read from a set $nde2neigh(\zeta)$ of indices in the input vector $\vec{U}$. The memory access of that loop is called *reflexive* if for every $v \in nde2neigh(\zeta)$, there exists an iteration that writes to $\vec{R}_v$ and reads from $\vec{U}_\zeta$ such that

$$v \in nde2neigh(\zeta) \Leftrightarrow \zeta \in nde2neigh(v),$$

where $\Leftrightarrow$ is used to express equivalence in this context. In other words, this means that the connectivity expressed in *nde2neigh* is an undirected graph: If v is a neighbour of ζ, then ζ is also a neighbour of v.

Loops whose iterations have reflexive memory access are symmetric as a whole as per Definition 2, but each iteration is not symmetric. The following holds.

Lemma 4. A loop that implements F as in Definition 3 has a Jacobian matrix ∇F with symmetric sparsity pattern. Its adjoint can be constructed such that each iteration reads from and writes to the same indices as the corresponding primal iteration, and the adjoint loop can use the same parallel schedule as the primal.

The reason for this can be explained as follows. For an iteration that writes to the center index $\vec{R}_\zeta$ and reads from all neighbours $\vec{U}_\nu$ in $\nu \in nde2neigh$, one finds that $(\nabla F)_{\zeta,\nu} \neq 0$. The reflexivity condition states that there is a corresponding iteration that reads from $\vec{U}_\zeta$ and writes to $\vec{R}_\nu$, leading to $(\nabla F)_{\nu,\zeta} \neq 0$, and hence a symmetric sparsity pattern. The tangent-linear code, which is an implementation of $(\nabla F)\cdot\dot{\vec{U}}$, can always re-use the parallelisation and read and write sets from the primal code. It computes $\dot{\vec{R}}_\zeta$ as a linear combination of all neighbour indices in the tangent-linear seed vector $\dot{\vec{U}}$ with the columns corresponding to neighbour indices in the Jacobian matrix, or formally,

$$\dot{\vec{R}}_\zeta \leftarrow (\nabla F)_{\zeta,\zeta}\cdot\dot{\vec{U}}_\zeta + \sum_{\nu\in nde2neigh(\zeta)} (\nabla F)_{\zeta,\nu}\cdot\dot{\vec{U}}_\nu. \tag{5.5}$$

As the sparsity pattern of the Jacobian matrix is symmetric, it is possible to construct an adjoint code that uses the same parallelisation and loop structure as the tangent-linear code as follows. The indices that the tangent-linear code uses to read from $\dot{\vec{U}}$ are used by the adjoint code to read from $\bar{\vec{R}}$, and the indices used by the tangent-linear code to write to $\dot{\vec{R}}$ are used by the adjoint code to write to $\bar{\vec{U}}$. This results in an adjoint code that, in each iteration, computes

$$\bar{\vec{U}}_\zeta \leftarrow (\nabla F)_{\zeta,\zeta}\cdot\bar{\vec{R}}_\zeta + \sum_{\nu\in nde2neigh(\zeta)} (\nabla F)_{\nu,\zeta}\cdot\bar{\vec{R}}_\nu. \tag{5.6}$$

The adjoint computation in (5.6) requires the coefficient $(\nabla F)_{\nu,\zeta}$ of the Jacobian matrix, while the corresponding iteration in the tangent-linear code (5.5) requires $(\nabla F)_{\zeta,\nu}$. The following conditions ensure that the transpose coefficient can be computed with information that is locally available in each restructured adjoint iteration.

Definition 4. Separability, permutability. A loop body is called *separable* if it computes $\vec{R}_\zeta$ as a summation over functions of the center node and one neighbour node, or formally,

$$\vec{R}_\zeta \leftarrow \sum_{\nu\in nde2neigh} f_\nu(\vec{U}_\nu,\vec{U}_\zeta).$$

In particular, this means that derivatives of that function with respect to the input from any neighbour ν_1 do not depend on the value at any other neighbour ν_2.
A loop body is called *permutable* if it uses the same function f for all neighbours.

Multi-activity differentiation as shown in Chapter 4 can be used to differentiate f separately for each input. In forward mode, derivatives of

$$w \leftarrow f(v_1, v_2)$$

with dependent output w, where either only the first input v_1 or only the second input is declared independent, are respectively called $\dot{f}_1$ and $\dot{f}_2$ and are an implementation of

$$\begin{aligned} \left(\frac{\partial f}{\partial v_1}\right) \cdot \dot{v}_1 &= \dot{f}_1(v_1, \dot{v}_1, v_2) \\ \left(\frac{\partial f}{\partial v_2}\right) \cdot \dot{v}_2 &= \dot{f}_2(v_1, v_2, \dot{v}_2). \end{aligned} \tag{5.7}$$

Similarly, reverse-differentiation yields the specialised procedures $\bar{f}_1$ and $\bar{f}_2$ given by

$$\begin{aligned} \left(\frac{\partial f}{\partial v_1}\right)^T \cdot \bar{w} &= \bar{f}_1(v_1, v_2, \bar{w}) \\ \left(\frac{\partial f}{\partial v_2}\right)^T \cdot \bar{w} &= \bar{f}_2(v_1, v_2, \bar{w}). \end{aligned} \tag{5.8}$$

These are needed in the following Lemma.

Lemma 5. For a loop that is reflexive, separable and permutable, each component of the adjoint output result vector $\bar{\vec{U}}$ is given by

$$\bar{\vec{U}}_\zeta = \sum_{v \in nde2neigh(\zeta)} \left(\frac{\partial f(\vec{U}_v, \vec{U}_\zeta)}{\partial \vec{U}_\zeta} \cdot \bar{\vec{R}}_\zeta + \frac{\partial f(\vec{U}_\zeta, \vec{U}_v)}{\partial \vec{U}_\zeta} \cdot \bar{\vec{R}}_v \right),$$

and can be computed using specialised forward-derivatives of f as

$$\bar{\vec{U}}_\zeta = \sum_{v \in nde2neigh(\zeta)} \left(\dot{f}_2(\vec{U}_v, \vec{U}_\zeta, \bar{\vec{R}}_\zeta) + \dot{f}_1(\vec{U}_\zeta, \bar{\vec{R}}_v, \vec{U}_v) \right),$$

or alternatively, using specialised reverse-derivatives of f with

$$\bar{\vec{U}}_\zeta = \sum_{v \in nde2neigh(\zeta)} \left(\bar{f}_2(\vec{U}_v, \vec{U}_\zeta, \bar{\vec{R}}_\zeta) + \bar{f}_1(\vec{U}_\zeta, \vec{U}_v, \bar{\vec{R}}_v) \right),$$

It will become clear in the following why this yields the same result as the standard adjoint code in Algorithm 6. Every iteration ζ in the standard adjoint code directly contributes to $\bar{\vec{U}}_\zeta$ the partial derivatives

$$\bar{\vec{U}}_\zeta += \sum_{v \in nde2neigh(\zeta)} \frac{\partial f(\vec{U}_v, \vec{U}_\zeta)}{\partial \vec{U}_\zeta} \cdot \bar{\vec{R}}_\zeta.$$

In addition, the iteration in the standard adjoint code makes contributions to $\bar{\vec{U}}$ at the neighbour nodes in $nde2neigh(\zeta)$. Because of the reflexivity, any contribution to $\bar{\vec{U}}_\zeta$ must

come from these same neighbours. Due to the permutability, those contributions must be partial derivatives of f, and due to the separability, these partial derivatives can be computed using only data that is locally available in the input vector $\vec{U}$ at the indices ζ and ν. It is therefore known that $\bar{\vec{U}}_\zeta$ receives the following contributions from the neighbour iterations:

$$\bar{\vec{U}}_\zeta += \sum_{\nu \in nde2neigh(\zeta)} \frac{\partial f(\vec{U}_\zeta, \vec{U}_\nu)}{\partial \vec{U}_\zeta} \cdot \bar{\vec{R}}_\nu \tag{5.9}$$

All together, this yields the result in Lemma 5. Note that the arguments are passed to f in (5.9) in swapped order, as the centre node of the current iteration is seen as the neighbour node in the iteration that makes this contribution in the conventional AD-generated adjoint. The code in Algorithm 5 satisfies the permutability and separability conditions. If the sets of neighbours satisfies the reflexivity condition and the function f satisfies the following purity condition, RSMP can be applied, leading to Algorithm 7 or Algorithm 8.

Definition 5. Purity. A function f is *pure* if it has no side effects, that is, it produces no output other than through its output arguments or return value. It can thus be called repeatedly without changing the program output, which is required to call multiple specialised derivatives of f. The same property is required for other AD techniques such as checkpointing. In Fortran, procedures can be explicitly declared as `pure`, although this is not required for RSMP to work.

5.3.3 Illustrating example

To illustrate RSMP, it is shown on the mesh from Figure 2.3. The primal reads

$$\vec{R} = \begin{bmatrix} f(\vec{U}_2, \vec{U}_1) + f(\vec{U}_4, \vec{U}_1) \\ f(\vec{U}_1, \vec{U}_2) + f(\vec{U}_3, \vec{U}_2) + f(\vec{U}_4, \vec{U}_2) \\ f(\vec{U}_2, \vec{U}_3) + f(\vec{U}_4, \vec{U}_3) \\ f(\vec{U}_1, \vec{U}_4) + f(\vec{U}_2, \vec{U}_4) + f(\vec{U}_3, \vec{U}_4) \end{bmatrix} \tag{5.10}$$

where each iteration writes to one row in the result vector. The adjoint model is

$$\bar{\vec{U}} = \begin{bmatrix} d_1 & \frac{\partial f(\vec{U}_1, \vec{U}_2)}{\partial \vec{U}_1} & 0 & \frac{\partial f(\vec{U}_1, \vec{U}_4)}{\partial \vec{U}_1} \\ \frac{\partial f(\vec{U}_2, \vec{U}_1)}{\partial \vec{U}_2} & d_2 & \frac{\partial f(\vec{U}_2, \vec{U}_3)}{\partial \vec{U}_2} & \frac{\partial f(\vec{U}_2, \vec{U}_4)}{\partial \vec{U}_2} \\ 0 & \frac{\partial f(\vec{U}_3, \vec{U}_2)}{\partial \vec{U}_3} & d_3 & \frac{\partial f(\vec{U}_3, \vec{U}_4)}{\partial \vec{U}_3} \\ \frac{\partial f(\vec{U}_4, \vec{U}_1)}{\partial \vec{U}_4} & \frac{\partial f(\vec{U}_4, \vec{U}_2)}{\partial \vec{U}_4} & \frac{\partial f(\vec{U}_4, \vec{U}_3)}{\partial \vec{U}_4} & d_4 \end{bmatrix} \cdot \bar{\vec{R}}$$

```
!$OMP parallel for shared(Ū,Ū̇,R̄,Ṙ) private(v1,v̇1,v2,v̇2,w,ẇ,ζ,ν);
for ζ ∈ nodes in mesh do
    Ū̇ζ ← 0;
    foreach ν ∈ nde2neigh(ζ) do
        v1 ← Ūν;
        v̇1 ← Ṙν;
        v2 ← Ūζ;
        v̇2 ← Ṙζ;
        ḟ1(v1 ↓, v̇1 ↓, v2 ↓, w ↑, ẇ ↑);
        Ū̇ζ ← Ū̇ζ + ẇ;
        ḟ2(v1 ↓, v2 ↓, v̇2 ↓, w ↑, ẇ ↑);
        Ū̇ζ ← Ū̇ζ + ẇ;
        R̄ζ ← R̄ζ + w;
    end
end
```

Algorithm 7: Restructured parallel adjoint using forward derivatives.

```
!$OMP parallel for shared(Ū,Ū̇,R̄,Ṙ) private(v1,v̄1,v2,v̄2,w,w̄,ζ,ν);
for ζ ∈ nodes in mesh do
    Ū̇ζ ← 0;
    foreach ν ∈ nde2neigh(ζ) do
        v1 ← Ūν;
        v2 ← Ūζ;
        w̄ ← Ṙν;
        f̄1(v1 ↓, v̄1 ↑, v2 ↓, w ↑, w̄ ↓);
        w̄ ← Ṙζ;
        f̄2(v1 ↓, v2 ↓, v̄2 ↑, w ↑, w̄ ↓);
        Ū̇ζ ← Ū̇ζ + v̄1 + v̄2;
        R̄ζ ← R̄ζ + w;
    end
end
```

Algorithm 8: Restructured parallel adjoint using reverse derivatives.

with diagonal terms

$$d = \begin{bmatrix} \frac{\partial f(\vec{U}_2,\vec{U}_1)}{\partial \vec{U}_1} + \frac{\partial f(\vec{U}_4,\vec{U}_1)}{\partial \vec{U}_1} \\ \frac{\partial f(\vec{U}_1,\vec{U}_2)}{\partial \vec{U}_2} + \frac{\partial f(\vec{U}_3,\vec{U}_2)}{\partial \vec{U}_2} + \frac{\partial f(\vec{U}_4,\vec{U}_2)}{\partial \vec{U}_2} \\ \frac{\partial f(\vec{U}_2,\vec{U}_3)}{\partial \vec{U}_3} + \frac{\partial f(\vec{U}_4,\vec{U}_3)}{\partial \vec{U}_3} \\ \frac{\partial f(\vec{U}_1,\vec{U}_4)}{\partial \vec{U}_4} + \frac{\partial f(\vec{U}_2,\vec{U}_4)}{\partial \vec{U}_4} + \frac{\partial f(\vec{U}_3,\vec{U}_4)}{\partial \vec{U}_4} \end{bmatrix}$$

Every iteration in the AD implementation of the adjoint model adds the contributions of a column in the matrix to all adjoint output indices, leading to scattered write access. For example, iteration 2 is computed in three calls to $\bar{f}$, one for each neighbour of node 2, and affects every entry in $\bar{\vec{U}}$:

$$\bar{\vec{U}} = \bar{\vec{U}} + \underbrace{\begin{bmatrix} \frac{\partial f(\vec{U}_1,\vec{U}_2)}{\partial \vec{U}_1} \cdot \bar{\vec{R}}_2 \\ \frac{\partial f(\vec{U}_1,\vec{U}_2)}{\partial \vec{U}_2} \cdot \bar{\vec{R}}_2 \\ 0 \\ 0 \end{bmatrix}}_{\bar{f}(\vec{U}_1,\bar{\vec{U}}_1\uparrow,\vec{U}_2,\bar{\vec{U}}_2\uparrow,\vec{R}_2\uparrow,\bar{\vec{R}}_2)} + \underbrace{\begin{bmatrix} 0 \\ \frac{\partial f(\vec{U}_3,\vec{U}_2)}{\partial \vec{U}_2} \cdot \bar{\vec{R}}_2 \\ \frac{\partial f(\vec{U}_3,\vec{U}_2)}{\partial \vec{U}_2} \cdot \bar{\vec{R}}_2 \\ 0 \end{bmatrix}}_{\bar{f}(\vec{U}_3,\bar{\vec{U}}_3\uparrow,\vec{U}_2,\bar{\vec{U}}_2\uparrow,\vec{R}_2\uparrow,\bar{\vec{R}}_2)} + \underbrace{\begin{bmatrix} 0 \\ \frac{\partial f(\vec{U}_4,\vec{U}_2)}{\partial \vec{U}_3} \cdot \bar{\vec{R}}_2 \\ 0 \\ \frac{\partial f(\vec{U}_3,\vec{U}_2)}{\partial \vec{U}_4} \cdot \bar{\vec{R}}_2 \end{bmatrix}}_{\bar{f}(\vec{U}_4,\bar{\vec{U}}_4\uparrow,\vec{U}_2,\bar{\vec{U}}_2\uparrow,\vec{R}_2\uparrow,\bar{\vec{R}}_2)}$$

Using RSMP, the adjoint code is reorganised such that each iteration acts on a single output element and has scattered read access. For iteration 2, this results in the following contributions, which require 6 evaluations of specialised derivatives of f.

$$\begin{aligned} \bar{\vec{U}}_2 = &\left[\frac{\partial f(\vec{U}_1,\vec{U}_2)}{\partial \vec{U}_2} \cdot \bar{\vec{R}}_2 + \frac{\partial f(\vec{U}_2,\vec{U}_1)}{\partial \vec{U}_2} \cdot \bar{\vec{R}}_1\right] && \left.\begin{matrix} \dot{f}^1(\vec{U}_2,\bar{\vec{R}}_2,\vec{U}_1,\vec{R}_2\uparrow,\bar{\vec{U}}_2\uparrow)+\dot{f}^2(\vec{U}_1,\vec{U}_2,\bar{\vec{R}}_1,\vec{R}_2\uparrow,\bar{\vec{U}}_2\uparrow) \\ \text{or} \\ \bar{f}^1(\vec{U}_2,\bar{\vec{U}}_2\uparrow,\vec{U}_1,\vec{R}_2\uparrow,\bar{\vec{R}}_2)+\bar{f}^2(\vec{U}_1,\vec{U}_2,\bar{\vec{U}}_2\uparrow,\vec{R}_2\uparrow,\bar{\vec{R}}_1) \end{matrix}\right. \\ + &\left[\frac{\partial f(\vec{U}_3,\vec{U}_2)}{\partial \vec{U}_2} \cdot \bar{\vec{R}}_2 + \frac{\partial f(\vec{U}_2,\vec{U}_3)}{\partial \vec{U}_2} \cdot \bar{\vec{R}}_3\right] && \left.\begin{matrix} \dot{f}^1(\vec{U}_2,\bar{\vec{R}}_2,\vec{U}_3,\vec{R}_2\uparrow,\bar{\vec{U}}_2\uparrow)+\dot{f}^2(\vec{U}_3,\vec{U}_2,\bar{\vec{R}}_3,\vec{R}_2\uparrow,\bar{\vec{U}}_2\uparrow) \\ \text{or} \\ \bar{f}^1(\vec{U}_2,\bar{\vec{U}}_2\uparrow,\vec{U}_3,\vec{R}_2\uparrow,\bar{\vec{R}}_2)+\dot{f}^2(\vec{U}_3,\vec{U}_2,\bar{\vec{U}}_2\uparrow,\vec{R}_2\uparrow,\bar{\vec{R}}_3) \end{matrix}\right. \\ + &\left[\frac{\partial f(\vec{U}_4,\vec{U}_2)}{\partial \vec{U}_2} \cdot \bar{\vec{R}}_2 + \frac{\partial f(\vec{U}_2,\vec{U}_4)}{\partial \vec{U}_2} \cdot \bar{\vec{R}}_4\right] && \left.\begin{matrix} \dot{f}^1(\vec{U}_2,\bar{\vec{R}}_2,\vec{U}_4,\vec{R}_2\uparrow,\bar{\vec{U}}_2\uparrow)+\dot{f}^2(\vec{U}_4,\vec{U}_2,\bar{\vec{R}}_4,\vec{R}_2\uparrow,\bar{\vec{U}}_2\uparrow) \\ \text{or} \\ \bar{f}^1(\vec{U}_2,\bar{\vec{U}}_2\uparrow,\vec{U}_4,\vec{R}_2\uparrow,\bar{\vec{R}}_2)+\bar{f}^2(\vec{U}_4,\vec{U}_2,\bar{\vec{U}}_2\uparrow,\vec{R}_2\uparrow,\bar{\vec{R}}_4) \end{matrix}\right. \end{aligned}$$

5.3.4 Test cases and results

RSMP removes the need for `atomic` or `critical` pragmas, but replaces every call to $\bar{f}$ with two calls to specialised derivatives of f. The effectiveness of RSMP thus depends on the effectiveness of the multi-activity approach, and the cost of using OpenMP pragmas relative to the cost of additional function calls. To investigate whether the method pays off, it was applied to a structured-mesh Burger's equation solver. As a reference, another adjoint solver is implemented using straightforward AD combined with OpenMP `atomic` pragmas. The primal and both adjoint solvers are then applied to the BEND case with 0.9 million mesh cells on the CPU and MIC platforms. The timing results reported in Figure 5.6 indicate that the reorganisation yields a much faster code than

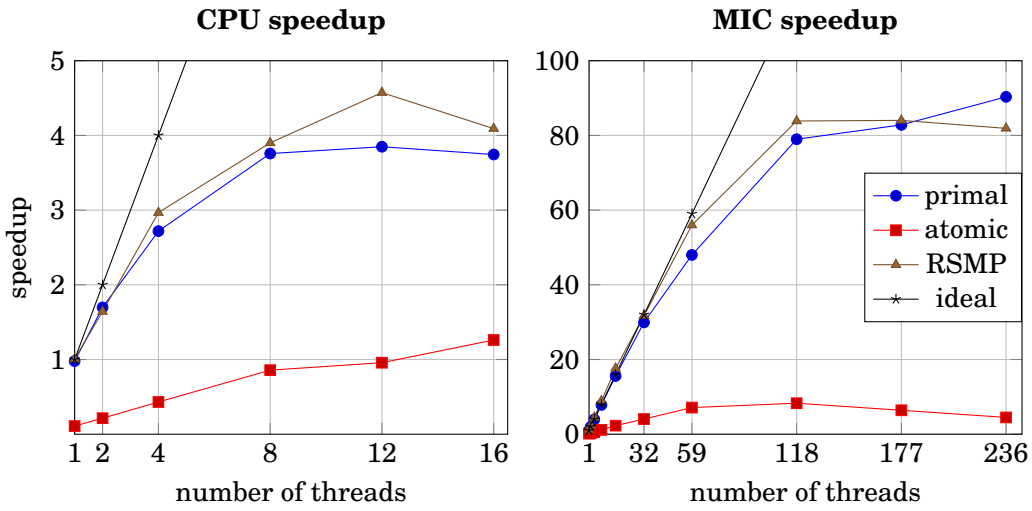


Figure 5.5: Speedup on CPU (left) and MIC (right) of primal, atomic adjoint, and RSMP adjoint code. Baseline timings are obtained by compiling the three codes with `-qopenmp-stubs` flag (ignore OpenMP pragmas, but link libraries). Atomic code uses standard AD plus manually added atomic pragmas. RSMP code uses reorganisation strategy with two specialised forward-differentiated functions. Primal code uses straightforward OpenMP parallel for loops. Scalability of primal and adjoint codes are roughly equal. Atomic pragmas make the adjoint code slower than the serial baseline if less than 12 CPU threads or 8 MIC threads are used.

the standard approach with AD and atomic pragmas, with a factor of more than 2 on the CPU and a factor of more than 5 on the MIC platform. This shows that, at least for relatively cheap functions f, the code duplication caused by repeated calls to specialised derivatives is significantly cheaper than atomic pragmas. The speedup shown in Figure 5.5 demonstrates that the adjoint solver generated using RSMP scales as well as the primal code on both CPU and MIC platforms, while the code using atomic pragmas scales poorly.

5.3.5 Applicability of RSMP

The reflexivity, separability, permutability and purity requirements can preclude the use of RSMP in practice, which is discussed in this section, along with potential remedies.

5.3.5.1 Reflexivity

Care needs to be taken when implementing boundary conditions. For instance, a one-dimensional PDE solver using a ghost cell approach to treat boundary conditions may contain a loop that performs read access from all elements of an input vector $\vec{U}_0 \ldots \vec{U}_n$, but performs write access only to the internal elements $\vec{R}_1 \ldots \vec{R}_{n-1}$. This would mean

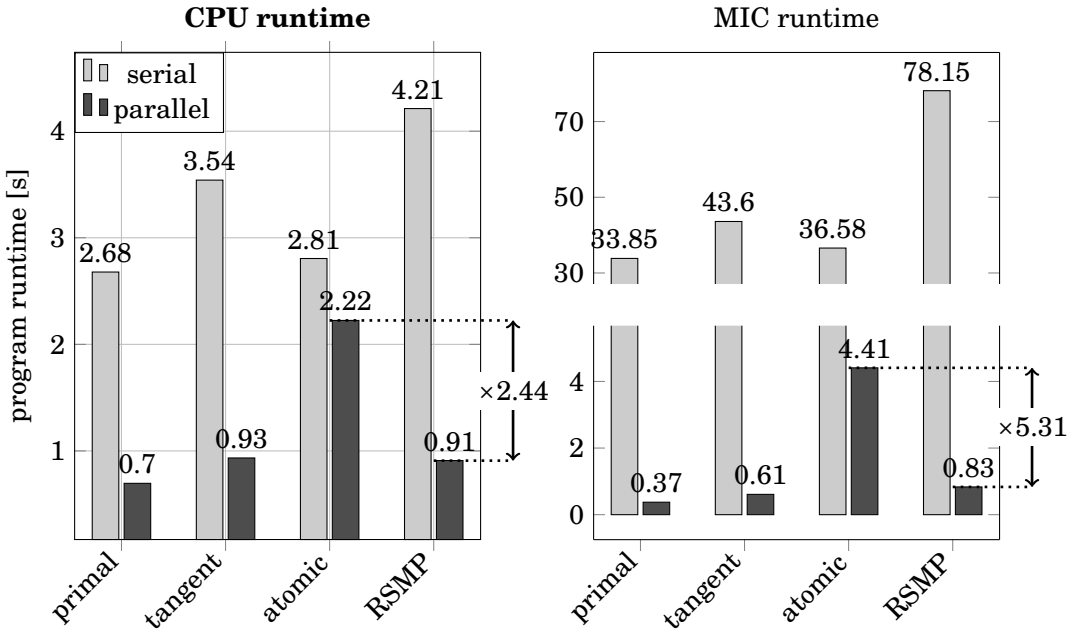


Figure 5.6: Best runtime in seconds for CPU (left) or MIC (right), 500 sweeps of Burger's equation residual computations on a U-bend mesh with 0.9 million cells. The serial runtime of the RSMP code is longer than that of the standard AD code. Despite the duplication of code, the runtime of RSMP is less than twice that of standard AD. In the RSMP case where two specialised derivative functions were generated by Tapenade, this is due to the removal of unneeded derivative code. With the reorganisation strategy, the code runs faster on the MIC system than on the CPU system, and the benefits of RSMP are much larger on the MIC than on the CPU.

that while there is an iteration that reads from $\vec{U}_0$ and writes to $\vec{R}_1$, there is no iteration that reads from $\vec{U}_1$ and writes to $\vec{R}_0$, violating the reflexivity condition. This could be fixed by rewriting the primal code such that the ghost cell also receives an update (that may be discarded in a separate, sequential loop), or by placing boundary computations in a separate loop, for which no efficient parallel adjoint is generated.

5.3.5.2 Separability

If the separability requirement is violated, the loop body is some function

$$f(\vec{U}_\zeta, \vec{U}_{\nu_1}, \vec{U}_{\nu_2}, \ldots),$$

where the derivatives have some nonlinear interdependence, that is, for any two neighbours ν_1 and ν_2, there is a nonzero mixed derivative so that

$$\frac{\partial}{\partial \vec{U}_{\nu_1}} \left(\frac{\partial f}{\partial \vec{U}_{\nu_2}} \right) \neq 0.$$

If this is the case, an iteration ζ in the restructured adjoint code can not compute the contribution from a neighbour v without also knowing the value of $\vec{U}$ at the indices corresponding to all other neighbours of v. This requires that the code generated with RSMP has access to distance-2-neighbours in the mesh. If the necessary data-structure is not already available for other reasons, creating it just for the implementation of RSMP may increase the memory footprint of the adjoint code to an unreasonable extent. It is also unclear how such a data-structure could be derived by an AD tool during compile-time.

5.3.5.3 Permutability

Consider a PDE solver that acts on a one-dimensional mesh in which each control volume has two neighbours l and r. The loop body may be implemented as some function

$$\vec{R}_\zeta = f_l(\vec{U}_l, \vec{U}_\zeta) + f_r(\vec{U}_r, \vec{U}_\zeta).$$

When computing the adjoint result $\bar{\vec{U}}_\zeta$ in the RSMP-generated code, it is necessary to detect which function is used by each of the neighbours to read from $\vec{U}_\zeta$. For example, the primal iteration that writes to $\vec{R}_r$ receives a contribution from $\vec{U}_\zeta$ through the function f_l. A restructured adjoint code can be constructed as

$$\begin{aligned}\bar{\vec{U}}_\zeta = &\left(\dot{f}_{l,2}(\vec{U}_l, \vec{U}_\zeta, \bar{\vec{R}}_\zeta) + \dot{f}_{r,1}(\vec{U}_\zeta, \bar{\vec{R}}_l, \vec{U}_l)\right) \\ &+ \left(\dot{f}_{r,2}(\vec{U}_r, \vec{U}_\zeta, \bar{\vec{R}}_\zeta) + \dot{f}_{l,1}(\vec{U}_\zeta, \bar{\vec{R}}_r, \vec{U}_r)\right),\end{aligned}$$

which requires finding the correct permutation of functions (in this case, left and right need to be swapped). While possible in theory, such a transformation is not easy to perform in practice without higher-level knowledge. The necessary information could be gathered at runtime.

5.3.5.4 Purity

If the loop body violates the purity condition, it can not be cloned and called multiple times without changing the final result. This precludes the use of randomness, input and output, or global variables.

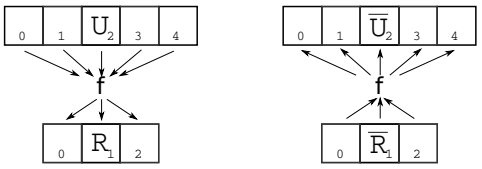


Figure 5.7: Reversal of memory access during reverse-differentiation of a code with fully non-symmetric memory access. The schedule that is used by the primal code to avoid write conflicts in the output vector $\vec{R}$ can not be used to avoid write conflicts in the adjoint output $\bar{\vec{U}}$. In such a case, OpenMP pragmas have to be used to avoid race conditions as summarised in Section 5.4.

5.4 General case: asymmetric memory access

It can be easily seen that `atomic` or `reduction` pragmas can not always be avoided. For instance, consider a parallel primal loop where each iteration concurrently reads from some active scalar variable v and writes its contribution to its own index in an output vector $\vec{R}$. Any adjoint implementation must gather the derivatives of all these loop iterations in a scalar adjoint variable $\bar{v}$, and this requires some synchronisation between threads to avoid race conditions. In some cases, it is possible to mix different parallelisation strategies. Consider a primal that has symmetric memory access for one input and one output vector, and in addition, reads from an active scalar. SSMP could be applied to parallelise the symmetric portions of the memory access, and a `reduction` pragma could be used for the adjoint of the scalar variable.

For completeness, Table 5.3 summarises the correct reverse-differentiation of parallel OpenMP loops that are neither symmetric nor reflexive. This can also be found in more detail in the literature [49, 50].

Primal variable scope	**Scope of corresponding adjoint variable**
private	private
firstprivate	reduction
lastprivate	shared (only first reverse iteration writes)
reduction	firstprivate
shared write	shared read
shared read	**atomic / critical/ reduction**

Table 5.3: Rules for adjoint OpenMP, showing the correct scope of adjoint variables (right), based on the scope of their corresponding primal variables (left). The previously presented special cases (SSMP and RSMP) affect only the treatment of shared read variables (last row, in bold).

CHAPTER 6

Incomplete checkpointing

Unsteady flow is an important feature of many technical systems. While time-dependent effects are sometimes neglected and steady-state simulations can be accurate enough for some practical purposes, it is often necessary to resolve transient flow behaviour, for example if transient effects are a crucial aspect of the system performance, such as in the simulation of wind turbine gust response, or flow in internal combustion engines. Another reason can be that the desired simulation accuracy can only be achieved when taking into account unsteady effects, for example in the simulation of turbine blades or aircraft wings (especially in high-lift configuration) [98]. Transient effects can have a large influence on the sensitivities even if a steady simulation is accurate enough to evaluate the cost function.

While unsteady simulations are becoming increasingly common in industrial applications, unsteady adjoint methods are not yet widely used. Unsteady adjoints can either be computed in frequency space [115], requiring periodic flow behaviour, or in temporal space [136], requiring the storage of the full flow history. Although unsteady adjoints have been studied, for example in the optimisation of flapping air foils [98], the memory requirement is prohibitive for most use cases, especially on accelerators such as GPUs or XeonPhi cards.

The memory requirements can be mitigated using, for example, the REVOLVE checkpointing algorithm [67]. With this approach, snapshots of the flow state are stored only at carefully chosen time steps. When needed, the primal computation can be resumed from these time steps to recompute flow states that have not been stored. While this reduces the memory footprint, the recomputations increase the computational cost. An alternative approach is to use data compression [154], which also reduces storage requirements at the cost of increased computations. If lossless compression methods are used, storage savings are limited [132].

This work investigates *coarse checkpointing*, a novel approach that, unlike general-

purpose data compression methods or checkpointing schemes, saves memory without increasing the computational cost, and allows a tradeoff between accuracy and memory footprint, without requiring significant additional computations. The method works by discarding some of the data that would otherwise be stored. During the adjoint computation, the missing data is approximated using a low-cost interpolation scheme. It is demonstrated that this approach can yield adjoint derivatives with an accuracy that is sufficient in some applications, and at the same time reduces the memory footprint to a fraction, with almost no computational cost. The method also reduces the amount of data transfers to and from memory, which is especially important if the computation is running on an accelerator, but the flow trajectory is stored on a host machine or an external storage system.

Another challenge for unsteady adjoint optimisation is that the adjoint field may never converge if the flow is chaotic, see [168], or if the cost function is not smooth enough, see [94]. It is shown in this chapter that the checkpoint coarsening can have a stabilising effect on the adjoint field for cases with complex unsteady behaviour, and may help in removing chaotic effects from the sensitivity computation.

6.1 Method description

An unsteady flow solver, such as `stamps` outlined in Algorithm 1, stores the flow state at every physical primal time step, and loads it again at the corresponding adjoint time step. This can happen either in fast volatile memory (which is mainly limited by the available space, e.g. less than 8 Gigabytes on the MIC system), or on a hard drive or solid state drive (which is mainly limited by the bandwidth of the connection between MIC and storage system and the write speed of the storage device). In either case, it is desirable to reduce the amount of data that needs to be stored or loaded. Conceptually, this can be implemented by replacing the procedures that store and load checkpoints, referred to as `store()` and `load()`, by augmented routines that perform the coarsening and reconstruction of checkpoints. These augmented procedures are described below for either spatial or temporal coarsening. It can be beneficial to combine spatial and temporal coarsening, as illustrated in the example in Section 6.1.3.

6.1.1 Temporal coarsening

One can distinguish between coarsening by *sampling* and *averaging*. Both use the same method to reconstruct the approximate flow trajectory from the stored checkpoints through interpolation, but differ in the way in which the trajectory is coarsened for storage.

if $t \in \mathcal{T}_s$ **then**
  store$(\vec{U}_t)$;
end

Algorithm 9: temporal sampling

if $t \in \mathcal{T}_s$ **then**
  store$(\sum_{i=t-c_t/2}^{t+c_t/2-1} \frac{1}{c_t}\vec{U}_i)$;
end

Algorithm 10: temporal averaging

if $t \in \mathcal{T}_s$ **then** // t was stored
  return ***load***(U_t);
else
  $\vec{U}^- \leftarrow$ ***load***(t^-);
  $\vec{U}^+ \leftarrow$ ***load***(t^+);
  return $\vec{U}^- + \frac{t-t^-}{t^+-t^-} \cdot (\vec{U}^+ - \vec{U}^-)$;
end

Algorithm 11: temporal reconstruction

For coarse sampling, the storage procedure selects certain snapshots worth storing, which are denoted by the set of stored time steps $\mathcal{T}_s$ which are a subset of all time steps $\mathcal{T}$. In the simplest incarnation of this method, $\mathcal{T}_s$ contains every c_t-th time step for some fixed temporal coarsening ratio c_t, see Algorithm 9. In some cases it may be beneficial to vary the checkpoint density over time, e.g. to capture a particular phenomenon with a higher accuracy. All checkpoints that are not in $\mathcal{T}_s$ are discarded. With this, one obtains a compression ratio $\|\mathcal{T}\|/\|\mathcal{T}_s\| = c_t$. For temporal averaging, instead of a particular snapshot, the average flow field over a certain time range is stored. For example, to compress the trajectory by a factor of c_t, the method computes the average over c_t time steps and stores them in a single snapshot, see Algorithm 10. In any case, checkpoints that have not been stored need to be reconstructed. If linear interpolation is used, any time step $t \in \mathcal{T}$ can be reconstructed from the closest stored time steps just after and before t, referred to as t^+ and t^-, with the linear interpolation implemented in Algorithm 11.

One could use higher order interpolation, taking into account more of the surrounding stored time steps. However, such a high-order reconstruction is still a linear combination of stored checkpoints (albeit with a smoother transition between coefficients). Because of this, flow states that are lost in the compression process can not be reconstructed regardless of the interpolation order, which results in "jumping" flow features if the gap sizes are too large. It may be beneficial for accuracy to use more advanced schemes that use feature tracking, as it is done for example in video compression. However, such methods would increase the computational cost, possibly even more so than using REVOLVE checkpointing.

6.1.2 Spatial coarsening

Spatial coarsening can be used to store a low-resolution flow state at each time step. While spatial and temporal coarsening are conceptually similar, their implementation is

not, at least for CFD solvers operating on unstructured meshes, as there is no obvious way to perform a uniform coarsening in space, and the mesh connectivity needs to be taken into account to gather the data from nearby cells.

Fortunately, CFD solvers that make use of the *multigrid* approach already have all the necessary routines in place to perform spatial coarsening and interpolation. A thorough discussion of multigrid methods is beyond the scope of this work and can be found for example in [20], but their general operation can be summarised as follows. Iterative solvers typically remove high-frequency error modes quickly, but can not efficiently remove low-frequency errors. Multigrid solvers work around this by *restricting* the solution to a coarse mesh, on which these errors modes can be removed with fewer iterations. The so-obtained update is *prolonged* back to the fine mesh, where the remaining high frequency error modes are removed. If the coarse system is itself restricted in a sufficiently deep sequence of increasingly coarse meshes, the number of required solver iterations becomes almost independent of the number of cells in the fine mesh.

Consider a fine mesh with N cells and a coarse mesh with N^c cells. The operator that restricts a state from the fine to the coarse mesh can be expressed as a matrix $\vec{M}_R \in \mathbb{R}^{N^c \times N}$. Likewise, the prolongation can be expressed as $\vec{M}_P \in \mathbb{R}^{N \times N^c}$. In `stamps`, the prolongation matrix $\vec{M}_P$ contains in each row i the interpolation coefficients for certain nearby coarse control volumes that are close to the fine control volume i. As a result, the value that is assigned to each fine control volume is a linear combination of values of nearby coarse control volumes, see Figure 6.1 for an illustration. The coefficients are constructed such that linear fields are exactly preserved by the prolongation, and all coefficients are non-negative. The restriction matrix is the transpose prolongation matrix. Although the restriction and prolongation operators were designed and implemented as part of the multigrid solver, they can be reused as a cost-effective data compression (Algorithm 12) and reconstruction (Algorithm 13), yielding a spatial coarsening ratio of

$$c_x := \frac{N}{N^c}.$$

6.1.3 Illustration: Interpolation in space and time

This section illustrates the effects of coarse sampling, temporal averaging, spatial averaging, and a mix of spatial and temporal averaging on a small example. Consider a bump that is transported with a constant velocity and is given by

$$\begin{aligned} r &= x - t + \frac{1}{2} \\ \Phi(r) &= \underbrace{e^4 \cdot e^{-\frac{1}{r-r^2}}}_{C^\infty \text{ bump}} \cdot \underbrace{\left(\frac{1}{2} + \frac{1}{3}\cos(32r-16) + \frac{1}{6}\cos(64r-32)\right)}_{\text{high frequency noise}}, \end{aligned}$$

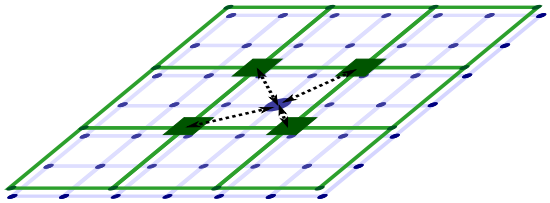


Figure 6.1: During multigrid prolongation, the value at the fine node (shown with a big dot) is computed as a linear combination of the value at the nodes that are part of the containing coarse element (shown with big squares). The weights for this linear combination depend on the distance between the fine and coarse nodes.

$U_t^c \leftarrow \vec{M}_R \cdot U_t$;
store(U_t^c);

Algorithm 12: spatial averaging

$U_t^c \leftarrow$ ***load***(t);
$U_t \leftarrow \vec{M}_P \cdot U_t^c$;
return U_t;

Algorithm 13: spatial reconstruction

which has a compact support at $[t-\frac{1}{2}, t+\frac{1}{2}]$ due to the first part of the function, and a noisy small scale structure due to the high frequency second part. This allows the observation of smoothing properties of the interpolation methods used in this work in Figure 6.2.

Over a range of $[0, 10]$ for both time t and space x, coarsening is used to reduce the storage size of the trajectory to $\frac{1}{16}th$ of the original size. A mix of temporal and spatial averaging is also used to achieve the same $\frac{1}{16}$ overall reduction, by reducing the temporal and spatial resolution each by $\frac{1}{4}$. The example shows that, as expected, coarse sampling exactly preserves the function at the selected time steps. However, the interpolation produces a poor approximation of the intermediate states. In this example, intermediate positions of the bump are not recovered, and instead the bump "jumps" from one position in space to the next, and the interpolation result contains several weak bumps at the locations where the bump was in the surrounding stored checkpoints.

The averaging methods (both in time and space) lead to a smoother transition between states. However, the methods cause artificial diffusion that removes the peak values and all high-frequency modes from the field, and the original flow field can not be exactly recovered anywhere or at any time. A mix of temporal and spatial averaging looks most promising, as the same data compression ratios can be achieved with a relatively high resolution in both time and space. In this example, coarsening time and space by a factor of 4 each results in a total compression ratio of 16, but with fewer losses in accuracy than the other tested methods.

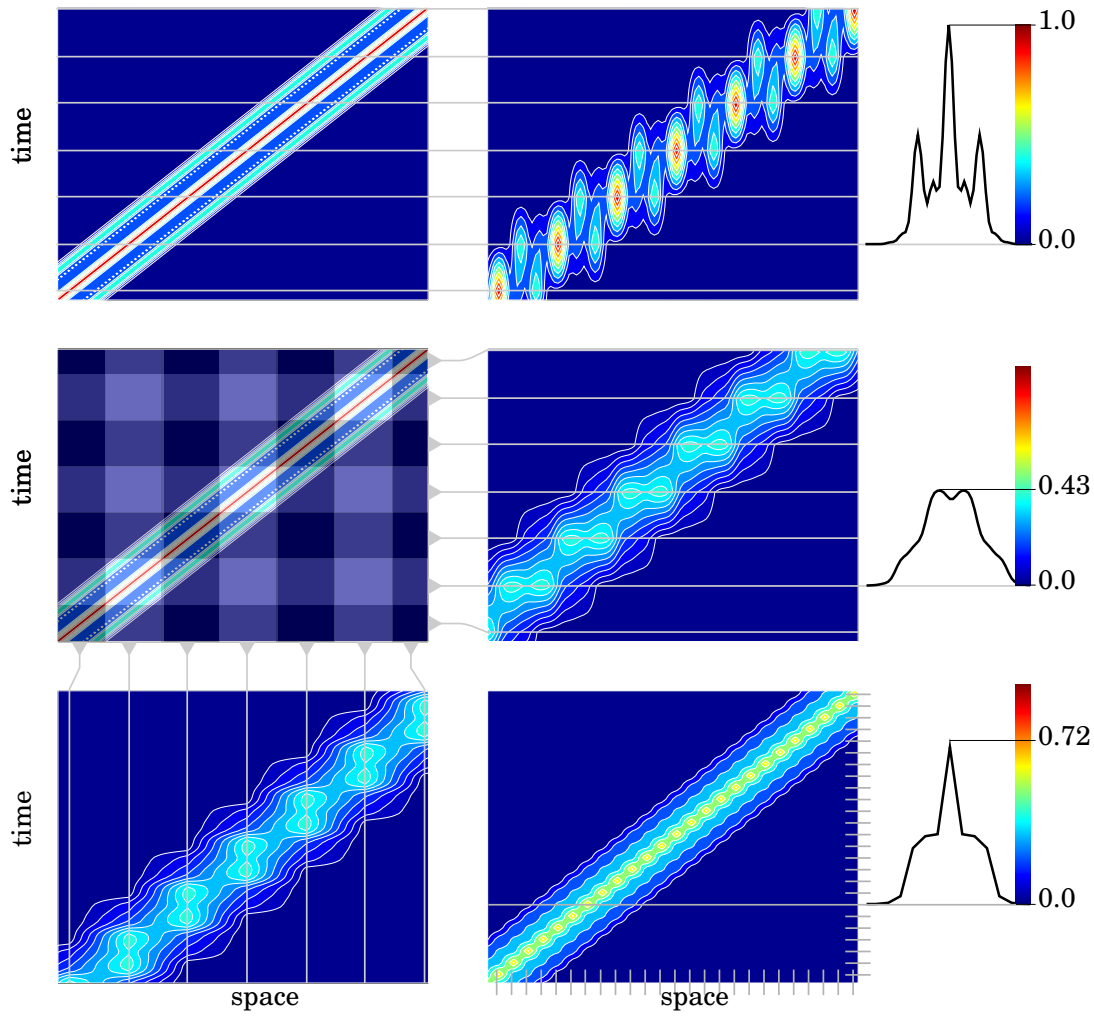


Figure 6.2: Illustration of temporal and spatial coarse checkpointing, using an irregularly shaped bump that is transported at a constant velocity through the space-time-plane.
Top left: Original bump function travelling in positive spatial direction with constant velocity, and shows up as a straight line in the space-time-plane shown here. The profile of the bump is shown on the far right of the page.
Top right: Coarse temporal sampling preserves the bump profile at selected time steps, but interpolation based on these samples does not yield a good approximation.
Middle left: Original function, horizontal stripes indicating time ranges used for temporal coarsening, and vertical stripes showing ranges used for spatial coarsening.
Middle right: Coarse temporal averaging eliminates all high frequencies from the bump and significantly reduces the peak amplitude of the bump. The reconstructed field is less noisy than that created by temporal sampling.
Bottom left: Coarse spatial averaging has a similar effect as coarse temporal averaging.
Bottom right: Combined spatial and temporal averaging can achieve a closer approximation of the original function than the other approaches tested here, while using the same amount of memory for the stored checkpoints.

6.2 Accuracy case study

The RAE2822 test case is used to demonstrate the effectiveness of checkpoint coarsening. Unsteady flow with vortex shedding is simulated with a reference time step size $\Delta t = 1\,\mathrm{ms}$ which corresponds to ca. 70 checkpoints for each shedding period. Coarsened trajectories are created from the reference primal trajectory using coarse sampling: For the setup with a coarsening ratio of $c_t = 2$, flow states at even time step numbers are discarded, and reconstructed using linear interpolation from the nearest odd time steps. By increasing the number of discarded time steps, one similarly obtains coarsening ratios of 4, 8, 16, 64 and 256. The adjoint solver operates on this reconstructed trajectory using a numerical time step size of Δt. For comparison, another set of primal results is generated using larger primal time step sizes of $2\Delta t \ldots 256\Delta t$.

The time-averaged cost function J is given by the time-dependent total aerofoil drag multiplied with a window function $\omega(t)$ that is used to make the cost function continuous in time, by ramping up the weight linearly until it reaches its peak at which it remains for 0.175 s, then ramping down to zero, as shown in Figure 6.8.

$$J = \frac{\sum_{t=1}^{t_\infty} drag(t) \cdot \omega(t)}{\sum_{t=1}^{t_\infty} \omega(t)} \tag{6.1}$$

The adjoint accuracy is investigated for three design parameters:

1. **Angle of attack:** This models a rigid body rotation of the aerofoil that allows only an adjustment of the angle of attack. Since this hides oscillatory errors in space and time, it can be used to study overall trends in the adjoint field [95].

2. **Surface node displacement:** This models surface node displacement in normal direction, which is common for shape optimisation applications [87, 175]. This allows an investigation of the spatial adjoint behaviour.

3. **Flow control:** This models a valve that is off by default, but can inject or remove tangential momentum just behind the top leading edge. The injection rate can vary in time and thus allows a study of the transient behaviour of the adjoint field [13, 69, 76]. The sensitivity with respect to this design parameter is computed for each time step by integrating the adjoint momentum over a small area around the valve location.

To compute surface sensitivity and angle of attack sensitivity, one requires a time-averaged adjoint field. The adjoint field is in this work averaged over a time frame that is twice as long as the cost averaging window given by ω.

6.2.1 Reference: Accuracy of primal coarsening

One can also reduce the memory footprint of unsteady adjoint computations by reducing the resolution of the primal simulation. For example, instead of using a temporal coarsening factor of 4, one could increase the physical time step size of the primal simulation by a factor of 4, in both cases reducing the number of stored checkpoints by and the memory footprint by the same factor. Likewise, instead of using spatial coarsening, one could run the primal simulation on a coarser mesh to reduce the size of stored checkpoints.

Previous work has found that using a steady solver for unsteady flow leads to wrong adjoint sensitivities [84, 95], even in cases where the primal results from the steady solver are reasonably accurate. This indicates that the temporal accuracy of the primal simulation has a large influence on the accuracy of adjoint derivatives.

The unsteady simulation on the RAE2822 test case confirms this effect. A snapshot of the flow field is shown in Figure 6.4. Increasing the time step size reduces the variance and the average drag and lift, and reduces the sensitivity of the drag cost function with respect to the angle of attack. Above a time step size of 16 ms, the oscillations are completely removed and the flow becomes steady. Increasing the step size further has negligible effect. The results are shown in Figure 6.3.

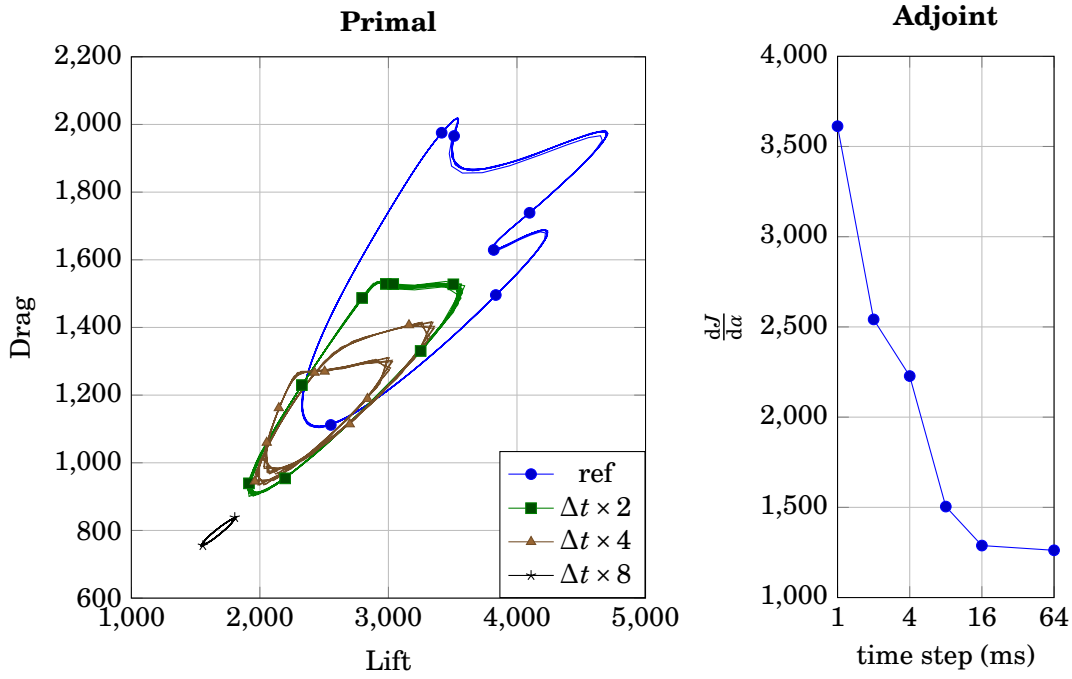


Figure 6.3: **Left:** Lift/drag for RAE2822 case with various time step sizes. Increasing the step size decreases drag, lift, and their oscillations. **Right:** Adjoint sensitivity of drag with respect to angle of attack. Increasing the time step size decreases the sensitivity.

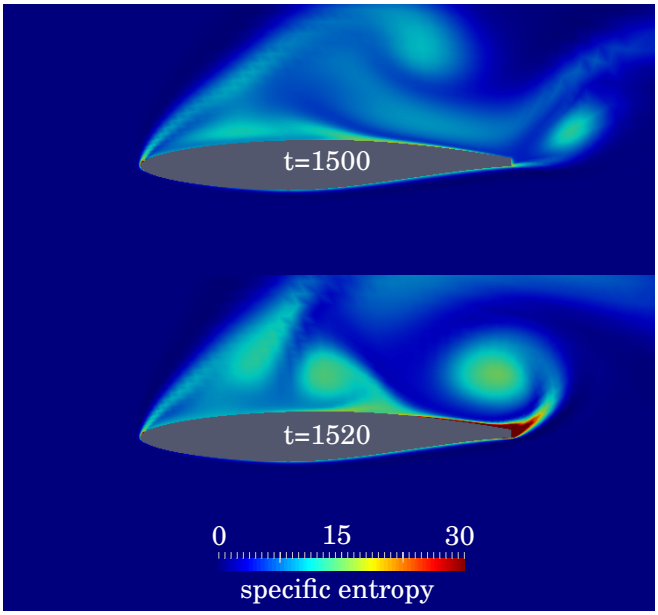


Figure 6.4: Snapshots after 1.5 s (1500 time steps) and 1.52 s (1520 time steps). Specific entropy, showing strong vortex shedding above the aerofoil with a frequency of roughly 70 time steps per period (0.07s).

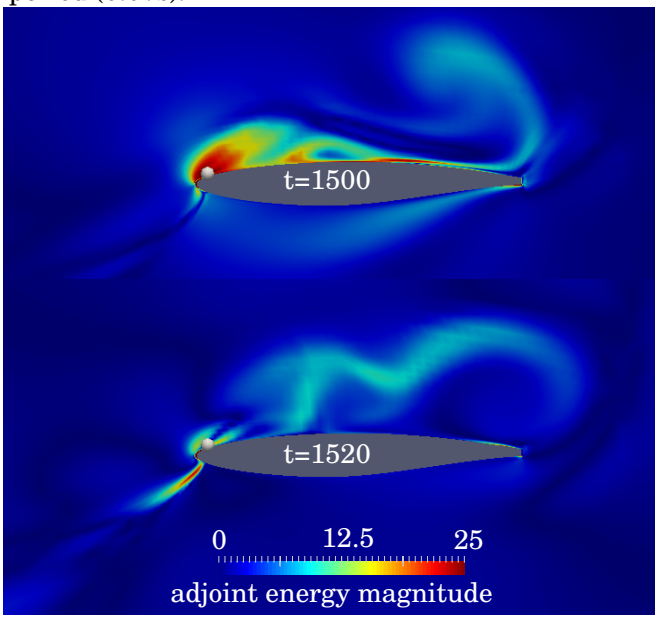


Figure 6.5: Adjoint energy magnitude field after 1.5 s and 1.52 s. A sphere above the leading edge marks the position of a flow control valve. The flow control sensitivity results are based on a design variable that is the momentum injection rate in aerofoil surface tangential direction at this location.

step/gap	$\frac{dJ}{d\alpha}$ for Δt	$\frac{dJ}{d\alpha}$ for c_t	memory reduction	relative error
1.0	**3612.203084**	**3612.203084**	0%	0
2.0	2541.615522	**3612.2**35254	50%	10^{-5}
4.0	2227.642980	**3612**.317687	75%	10^{-5}
8.0	1504.326992	**361**5.459025	88%	10^{-3}
16.0	1288.293729	**3612**.068629	94%	10^{-3}
64.0	1262.006153	**36**06.972954	98%	10^{-3}
256.0	1228.272627	**3**364.129272	>99%	10^{-1}

Table 6.1: **Second column:** $\frac{dJ}{d\alpha}$ for different primal time step sizes. **Third column:** $\frac{dJ}{d\alpha}$ for different gap sizes using temporal checkpoint coarsening. Large memory reductions are achieved with moderate relative errors.

6.2.2 Angle of attack sensitivity

The sensitivity of the aerofoil drag with respect to its angle of attack $\frac{dJ}{d\alpha}$ is computed from the time-averaged volume adjoint field, projected onto the aerofoil surface using the spring analogy model [18], which is a method to compute the relationship between the displacement of interior and surface mesh nodes. For each point on the aerofoil surface, the cross product of the sensitivity vector with the vector from the aerofoil center to that point is computed, and integrated over the surface. The result is shown in Table 6.1. One observes that the sensitivity computed with a coarsening ratio of $c_t = 2$ has a relative error of 10^{-5} compared to the reference solution, but reduces the memory footprint by 50%. With $c_t = 64$, one obtains a memory reduction above 98%, with 10^{-3} relative error.

6.2.3 Surface sensitivity

The sensitivity of the aerofoil drag with respect to normal surface node displacement $\frac{dJ}{d(\vec{x}\cdot\vec{n})}$ is also based on the time-averaged surface-projected adjoint field. The results show that the temporal accuracy of the primal simulation is crucial, as the sensitivities computed based on a coarsened primal are strongly dependent on the time step size. For primal step sizes of $8\Delta t$ and beyond, the sensitivity on the aerofoil top vanishes, which is an indicator that the unsteadiness is no longer resolved. The surface sensitivity on the aerofoil bottom is resolved correctly for large time step sizes and is the only contributor to the angle of attack sensitivity above $8 \cdot \Delta t$. Temporal checkpoint coarsening with factors up to 8 result in surface sensitivities that match the reference result to plotting accuracy, see Figure 6.6. A more detailed investigation of the errors shows that the error grows somewhat with the gap size, see Figure 6.7. The largest errors are located on the aerofoil top, in areas with more intense unsteadiness.

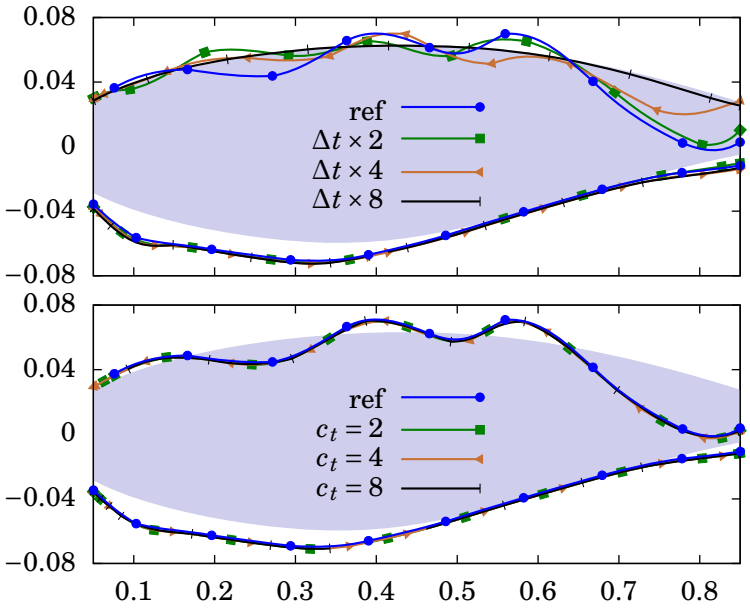


Figure 6.6: Surface drag sensitivity, superimposed on the aerofoil surface. **Top:** Results for different primal time step sizes. Reducing the temporal accuracy leads to a decrease in sensitivity. For $8\Delta t$ and above, the surface sensitivity on the aerofoil top vanishes. The aerofoil bottom shows a strong sensitivity which is resolved correctly regardless of time step size. **Bottom:** Different temporal gap sizes for incomplete checkpointing. The sensitivities match that of the reference solution to plotting accuracy.

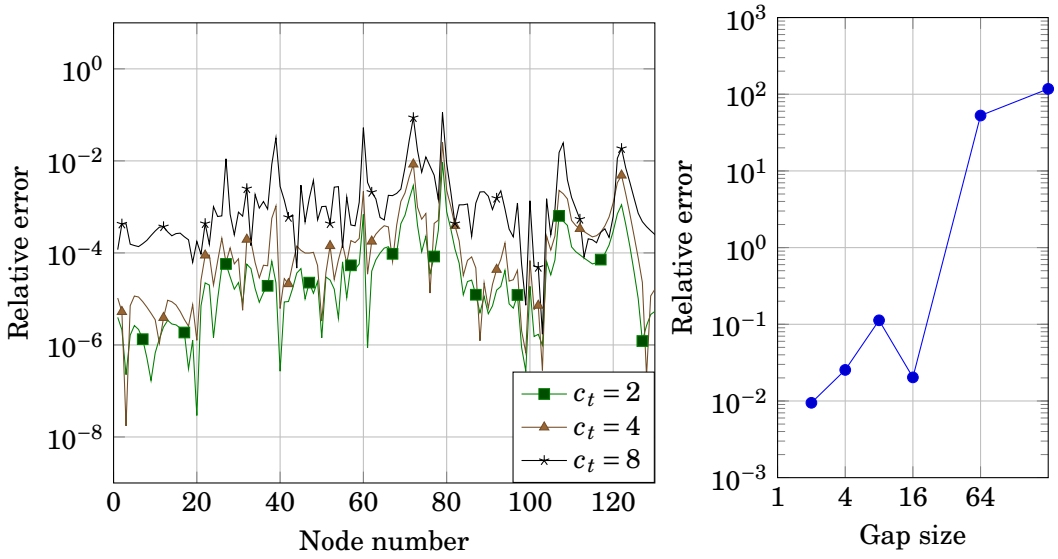


Figure 6.7: **Left:** Surface drag sensitivity error along the aerofoil surface. Errors are larger on the aerofoil top (nodes 50 to 95) than at the bottom (nodes 105 to 132 and 1 to 20). **Right:** Maximum relative error of surface sensitivity for different gap sizes. The relative error for $c_t = 2 \ldots 16$ is labeled and, surprisingly, smaller for $c_t = 16$ than for $c_t = 8$.

6.2.4 Flow control sensitivity

Active flow control can be achieved with a valve that can inject or remove momentum from the flow. The sensitivity of the average drag with respect to momentum injection $\frac{\mathrm{d}J}{\mathrm{d}I}$ shows a more detailed picture of the transient adjoint field, as it is varying in time and computed for a small spatial area at the location shown in Figure 6.5, and is therefore susceptible to high-frequency errors in both space and time, as shown in Figure 6.8.

For coarsening ratios 2 and 4, the adjoint momentum results agree well with each other, and vanish after the cost function averaging window ends. For ratios of 8 and above, the sensitivity diverges as the adjoint solution is evolved backwards in time. This growing oscillatory error cancels out in the time-averaging process that was used for the angle of attack and surface sensitivity studies, and does not seem to impact the results in these test cases.

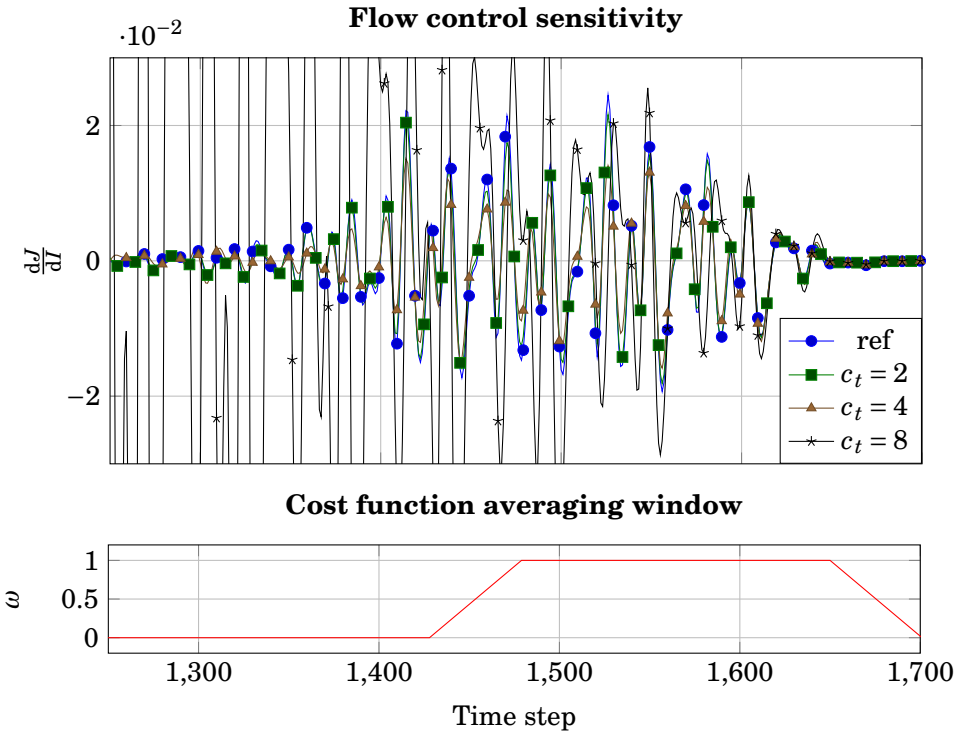


Figure 6.8: **Bottom:** cost function averaging window ω. **Top:** Sensitivity of drag with respect to momentum injection above the leading edge, plotted over time for a series of incomplete checkpointing gap sizes. Adjoints are zero beyond time step 1700, as any change happening after the window can not affect the cost function. The sensitivity peaks for a time frame that is slightly longer than the window. The setup with $c_t = 8$ diverges, all other setups match well.

6.3 Stability case study

While simple cases such as the RAE2822 are stable enough for the unsteady adjoint field to converge, chaotic [168] or nearly chaotic [141] cases can cause sensitivities to diverge as the adjoint field is evolved backwards in time. The flow in the BEND case is sufficiently unstable to trigger such behaviour, with interacting pressure waves reflected between inlet and outlet, small vortices in the bend, and larger ones downstream.

Spatial checkpoint coarsening, as illustrated for the BEND case in Figure 6.9, does not only reduce the size of the trajectory, but also removes small-scale turbulent effects. A snapshot of the flow behaviour as well as of the adjoint field with and without coarsening is shown in Figure 6.10, and the adjoint snapshot based on the coarsened trajectory is qualitatively similar to the reference, but with a smaller magnitude.

Figure 6.11 shows that 2 s of physical time, corresponding to 20,000 time steps, are needed for the unsteady solver to reach a state that is not obviously influenced by the initial condition. A C^∞-smooth averaging window [94] is used to prevent adjoint divergence. Without coarsening, the adjoint momentum is not transported out of the domain, and the field maintains a similar magnitude after the cost function becomes inactive and there is hence no remaining adjoint source term. Whether this is due to a numerical instability or a physical phenomenon is unclear. With checkpoint coarsening, the adjoint field vanishes as expected, and can be integrated to obtain surface sensitivities.

Figure 6.12 shows the surface sensitivity for the reference and coarsened case. While the results for different coarsening ratios agree with each other and with previous studies using steady flow [165], further investigation (e.g. using a finite-differencing test) is needed to confirm that the adjoint derivatives are indeed correct. The reference result computed without checkpoint coarsening shows numerical artifacts in the bend geometry caused by the non-converged adjoint field. This indicates that checkpoint coarsening can be used to obtain usable results in cases where the adjoint field diverges, in a less rigorous, but also much less expensive procedure than previously presented [94].

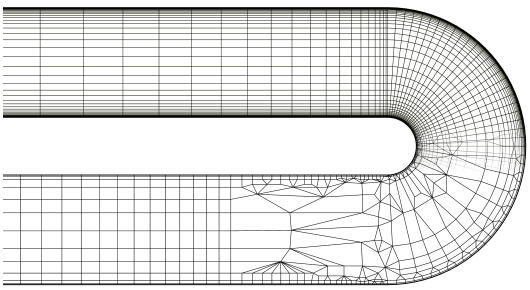


Figure 6.9: **Upper half:** Fine mesh. **Lower half:** coarsened with factor 4

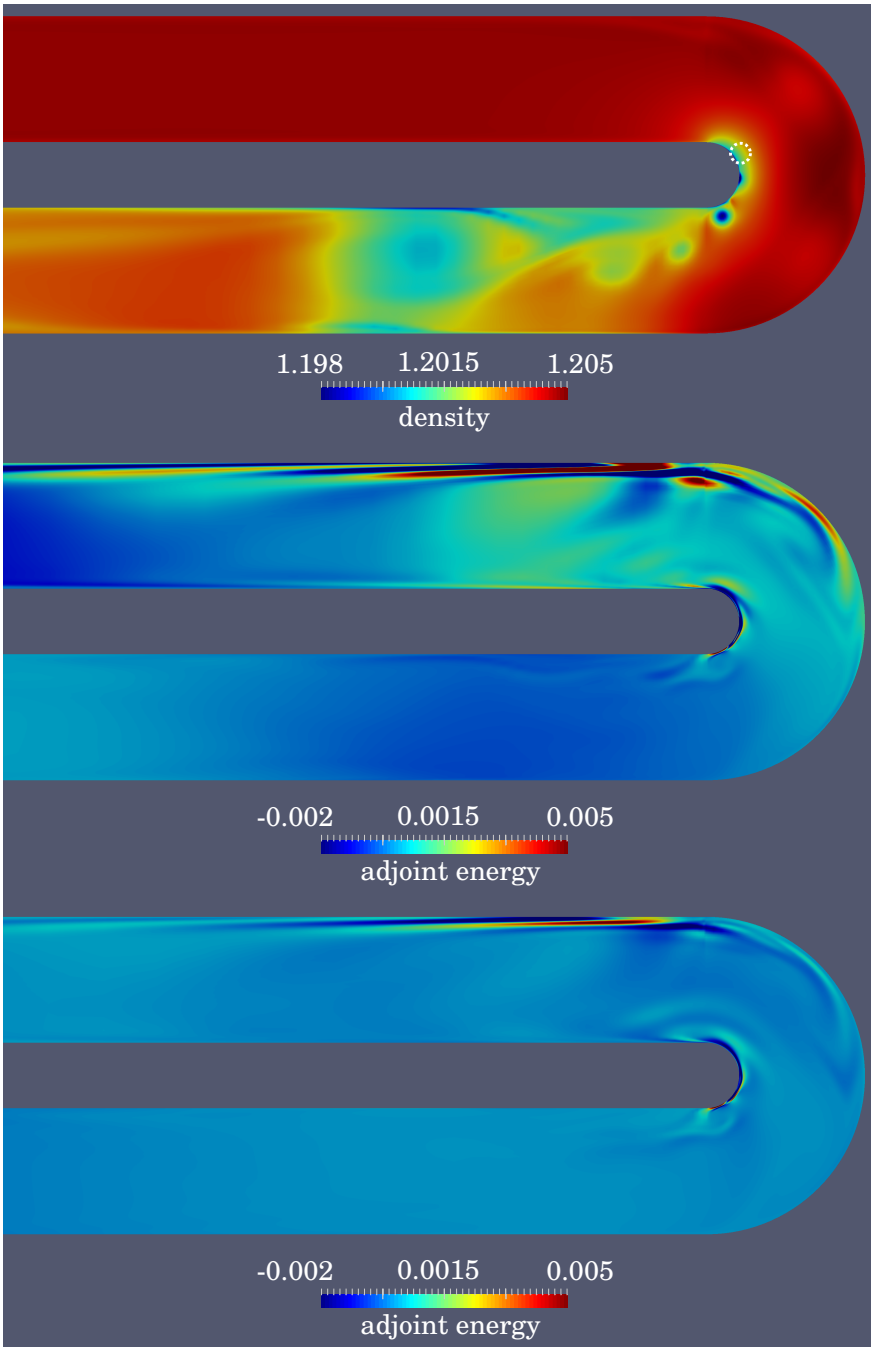


Figure 6.10: **Top:** Density at 9 s (90,000 forward time steps). Location of control flow valve shown with dotted white circle. **Center:** Adjoint energy at 9 s (10,000 reverse time steps) for reference primal. **Bottom:** Adjoint energy for $c_x = 4$ case. Qualitatively similar to the reference field, but with smaller magnitude.

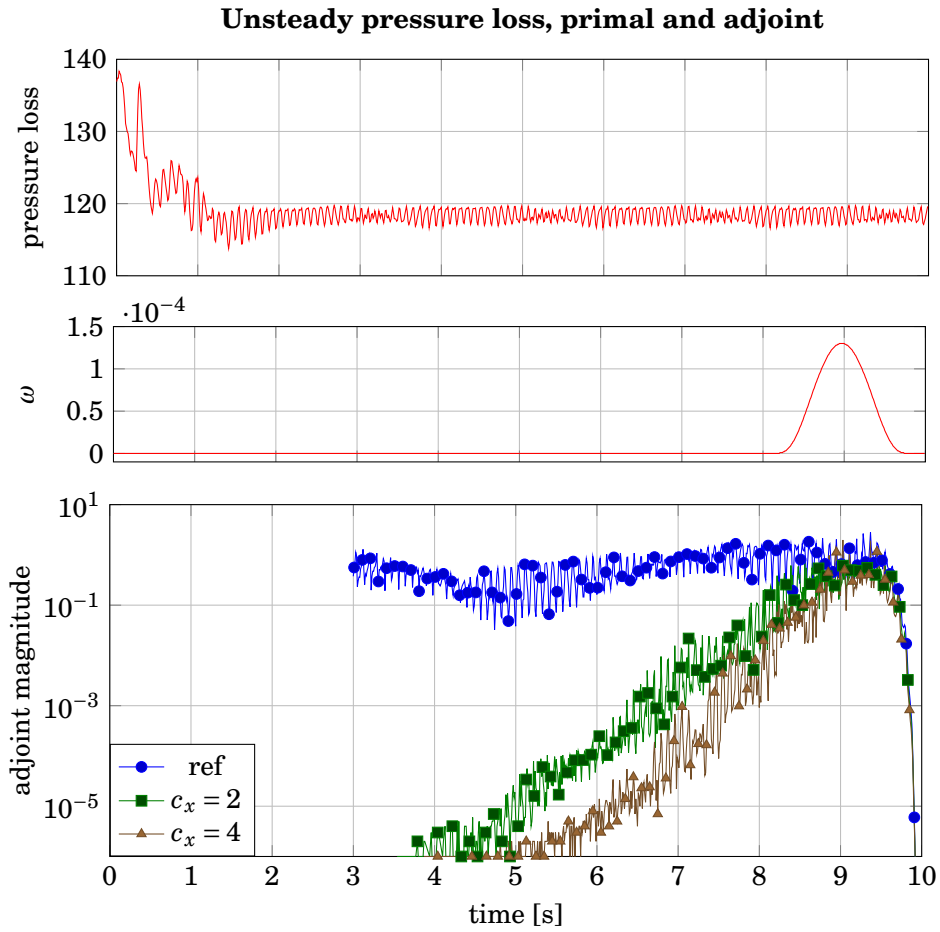


Figure 6.11: **Top:** Pressure loss history for a range of 10 s. **Center:** Cost function averaging window. **Bottom:** Adjoint momentum magnitude at the valve location shown in Figure 6.10. The reference field does not vanish. The computation was stopped at $t = 3$s.

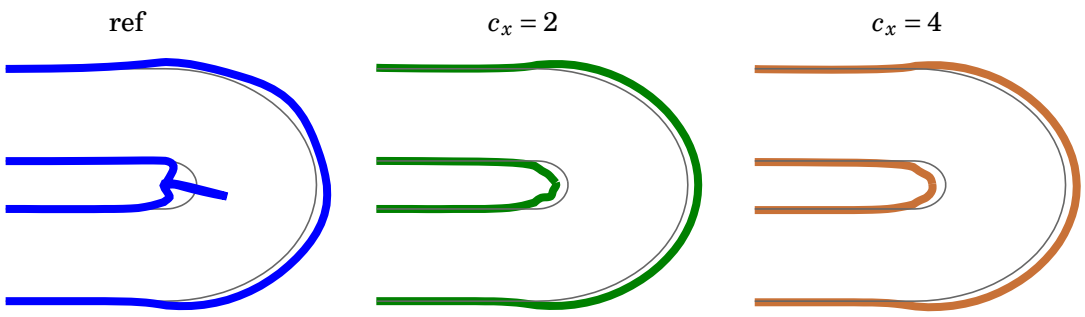


Figure 6.12: Surface sensitivity for time-averaged unsteady adjoint field, for reference, $c_x = 2$ and $c_x = 4$ test case.

CHAPTER 7

Convergence of differentiated Krylov solvers

Krylov methods are popular for solving large sparse linear systems, such as those arising in CFD solvers [93]. As discussed in Section 2.3, the implementation of adjoint solvers requires either that the linear solvers in the underlying primal are differentiated using AD, or preferably, that a hand-differentiated linear solver is implemented. Both methods yield as accurate results as the primal solver in the absence of roundoff errors and if fully converged, but hand-differentiation leads to better computational efficiency.

In practice, accuracy in the presence of roundoff, fast convergence, and reliability are crucial. Numerous methods have been developed to efficiently solve the linear systems that arise in CFD. Among them is the Conjugate Gradient method (CG), whose convergence is guaranteed under certain conditions. Section 7.4 demonstrates that these guarantees do not extend to AD-generated derivatives of CG in floating point arithmetic.

Linear systems that arise in CFD are often non-symmetric, precluding the use of CG. Popular methods that work on non-symmetric systems are the BiConjugate Gradient method (BiCG), a stabilised version of the former (BiCGSTAB), Conjugate Gradient Squared (CGS), or Generalised Minimal Residual (GMRES). These methods are known to be less reliable than CG [10, 137, 172], as confirmed by the following quotes (emphasis on method names added):

> “Few theoretical results are known about the convergence of **BiCG**. In practice [...] the convergence behavior may be quite irregular, and the method may even break down.”
>
> “This is evidenced by the often highly irregular convergence behaviour of **CGS**. [...] local corrections to the current solution may be so large that cancellation effects occur. [...] The method tends to diverge if the starting guess is close to the solution.”
>
> *Barrett et al, Templates for the solution of linear systems, p. 19, p. 23 [10]*

> **BiCGSTAB** often converges about as fast as CGS, sometimes faster and sometimes not. [If ...] the local GMRES(1) step stagnates, [...] Bi-CGSTAB will break down. This is a breakdown situation that can occur in addition to the other breakdown possibilities in the underlying BiCG algorithm.
>
> *Barrett et al, Templates for the solution of linear systems, p. 24*

An exception to this is the GMRES method, for which convergence properties are known, but do not apply if GMRES is restarted after every k iterations, as is done in GMRES(k) methods. Without restarts, memory footprint, computational cost and roundoff error grow with every iteration, which is a particular problem on accelerators.

> If no restarts are used, **GMRES** [...] will converge in no more than n steps (assuming exact arithmetic). Of course this is of no practical value when n is large; moreover, the storage and computational requirements in the absence of restarts are prohibitive.
>
> *Barrett et al, Templates for the solution of linear systems, p. 17*

Among other things, convergence speed and reliability of iterative linear solvers depend on the spectrum and condition of the system matrix, and can often be improved by using preconditioning methods [12], a topic that is beyond the scope of this work. One might hope that a method that can be successfully used to solve a primal system will work equally well on its adjoint system. After all, for a primal system

$$\vec{A}\vec{y} = \vec{b},$$

the adjoint system matrix is simply the transpose of the primal system matrix as in

$$\vec{A}^T\bar{\vec{b}} = \bar{\vec{y}}.$$

As a consequence, both system matrices have identical spectral properties, condition number, size, and number of nonzero entries. This is sometimes mentioned in the literature as a key advantage of the discrete adjoint method over continuous methods [61], and discrete adjoint CFD solvers are routinely developed under the assumption that the convergence rate for the primal and adjoint systems are identical [39, 57].

While this may be true most of the time in practice, the lack of formal convergence guarantees for most iterative solvers means that there is also no guarantee that the adjoint system can be solved as quickly or reliably as the primal system, and indeed, counterexamples can easily be found. Section 7.2 demonstrates on a matrix that arises in practical CFD applications, that most iterative linear solvers break for certain right hand side vectors. Furthermore, it shows that solvers may converge well for a given matrix and right hand side, but break down if the system matrix is transposed. Section 7.3 presents a detailed analysis of this phenomenon using a 2×2 matrix.

7.1 CG and BiCG

CG and BiCG are briefly discussed here, as they are used in the following test cases, form the basis of the related BiCGSTAB and CGS methods, and illustrate convergence and breakdown. For a system $\vec{A}\vec{y} = \vec{b}$ with $\vec{A} \in \mathbb{R}^{n\times n}$, $\vec{y}, \vec{b} \in \mathbb{R}^n$, CG as shown in Algorithm 14 is guaranteed to find the solution to the linear system in at most n steps in exact arithmetic if $\vec{A}$ is symmetric positive definite (SPD). In practice, it is often enough to use fewer iterations to find an approximation of $\vec{y}$. CG can break down due to a division by zero if the matrix is not SPD, as the denominator σ_k may become zero for nonzero p. The search direction p becomes zero only if the residual is zero and the system is solved. In floating point arithmetic, CG may break without finding the solution, or may run for more than n iterations, as seen in Section 7.4.

BiCG (shown in Algorithm 15) is designed for non-symmetric matrices, but convergence is only guaranteed for SPD matrices, for which BiCG and CG behave identically. For other matrices, BiCG may break down in exact or floating point arithmetic. There are two breakdown situations [155], known as *pivotal breakdown* (caused by $\sigma = 0$), and *breakdown in the underlying Lanczos process* (caused by $\rho = 0$). One notices that ρ_k in CG can only become zero if r_k is zero, which means that a solution was found. In BiCG, ρ_k becomes zero if r_k is zero and a solution was found, but also if $\hat{r}_k$ is zero, or if r_k and $\hat{r}_k$ are orthogonal. Section 7.3 identifies right hand sides that trigger such breakdowns.

$p_0, r_0 \leftarrow b - \vec{A} \cdot \vec{y}_0$;
for $k \leftarrow 0 \dots n-1$ **do**
  $\sigma_k \leftarrow p_k^T \cdot \vec{A} \cdot p_k$;
  $\alpha_k \leftarrow \frac{r_k^T \cdot r_k}{\sigma_k}$;
  $\vec{y}_{k+1} \leftarrow \vec{y}_k + \alpha_k \cdot p_k$;
  $r_{k+1} \leftarrow r_k - \alpha_k \cdot \vec{A} \cdot p_k$;
  $\rho_k \leftarrow r_k^T \cdot r_k$;
  $\beta_k \leftarrow \frac{r_{k+1}^T \cdot r_{k+1}}{\rho_k}$;
  $p_{k+1} \leftarrow r_{k+1} + \beta_k \cdot p_k$;
  if $\|r_{k+1}\| = 0$ **then**
    return $\vec{y}_{k+1}$;
  end
end

Algorithm 14: Conjugate Gradient solver. The highlighted statements are guaranteed not to divide by zero.

$\hat{p}_0, \hat{r}_0, p_0, r_0 \leftarrow b - \vec{A} \cdot \vec{y}_0$;
for $k \leftarrow 0, 1 \dots$ **do**
  $\sigma_k \leftarrow \hat{p}_k^T \cdot \vec{A} \cdot p_k$;
  $\alpha_k \leftarrow \frac{\hat{r}_k^T \cdot r_k}{\sigma_k}$;
  $\vec{y}_{k+1} \leftarrow \vec{y}_k + \alpha_k \cdot p_k$;
  $r_{k+1} \leftarrow r_k - \alpha_k \cdot \vec{A} \cdot p_k$;
  $\hat{r}_{k+1} \leftarrow \hat{r}_k - \alpha_k \cdot \vec{A}^T \cdot \hat{p}_k$;
  $\rho_k \leftarrow \hat{r}_k^T \cdot r_k$;
  $\beta_k \leftarrow \frac{\hat{r}_{k+1}^T \cdot r_{k+1}}{\rho_k}$;
  $p_{k+1} \leftarrow r_{k+1} + \beta_k \cdot p_k$;
  $\hat{p}_{k+1} \leftarrow \hat{r}_{k+1} + \beta_k \cdot \hat{p}_k$;
  if $\|r_{k+1}\| = 0$ **then**
    return $\vec{y}_{k+1}$;
  end
end

Algorithm 15: BiConjugate Gradient solver. The method can break down in the two highlighted statements.

7.2 Solver stability in primal and adjoint systems

This section investigates whether the convergence behaviour of iterative solver methods on some *original system* $\vec{A}\vec{y} = \vec{b}$ can reliably predict the convergence behaviour of the same solver on the *transposed system* $\vec{A}^T\hat{\vec{y}} = \vec{b}$. To make this study reproducible and at the same time relevant to real-world applications, the system matrix is chosen to be the `STEAM2` matrix, available from the Florida matrix collection [36]. It arises from the modelling of steam in an oil reservoir, discretised on a $5 \times 5 \times 6$ mesh in three dimensions with four degrees of freedom on each grid point, resulting in a total matrix size of 600×600. The matrix is real, invertible, non-symmetric, not diagonal dominant, with 13760 nonzero entries and a condition number of ca. 10^6.

For the right hand side vector $\vec{b}$, two options are explored. First, b is randomly filled with entries in the interval $[0 \ldots 1]$. Second, b is set to a Cartesian basis vector in $\mathbb{R}^{600}$, where e_i denotes the unit vector in the i-th axis direction. The iterative solver methods BiCG, CGS, BiCGSTAB and GMRES are applied to all of these systems, and the solvers are allowed to run for 500 iterations, but generally stall after less than 50 iterations. GMRES was used with a restart parameter of 30, which is suggested by [7], and used by default in the PETSc library [8].

For the randomly generated right hand side vectors, the convergence behaviour for all tested solvers does not vary significantly between different vectors, and the final residual is always similar for the original and transpose systems. GMRES, BiCG and CGS reduce the residual by 5 or 6 orders of magnitude, that is, the final residual is smaller than the initial residual by a factor of 10^5 or 10^6 respectively. In contrast, BiCGSTAB stalls after reducing the residual by only 3 orders of magnitude.

If the right hand side vector is a Cartesian basis vector, the convergence behaviour of the tested solvers becomes erratic. In Table 7.1, a convergence summary for the first 25 basis vectors is shown. One can observe that BiCG, BiCGSTAB and CGS converge for some basis vectors, but stall or break down for others. The same right hand side may work for the original system but cause a breakdown for the transpose system, or vice versa. BiCGSTAB does not break for any right hand side on the transpose system, but breaks frequently for the original system. The only solver that does not break or stall for any of the tested systems is GMRES, at least if the restart parameter is high enough. If the parameter is lowered to less than 4, GMRES stalls for all cases in which BiCG stalls or breaks.

In the context of AD, this study shows that convergence of adjoint solvers can not be taken for granted even if primal solvers converge easily, and that right hand sides with a certain structure can cause solver breakdown even for reasonably well converged system matrices that do not otherwise cause problems.

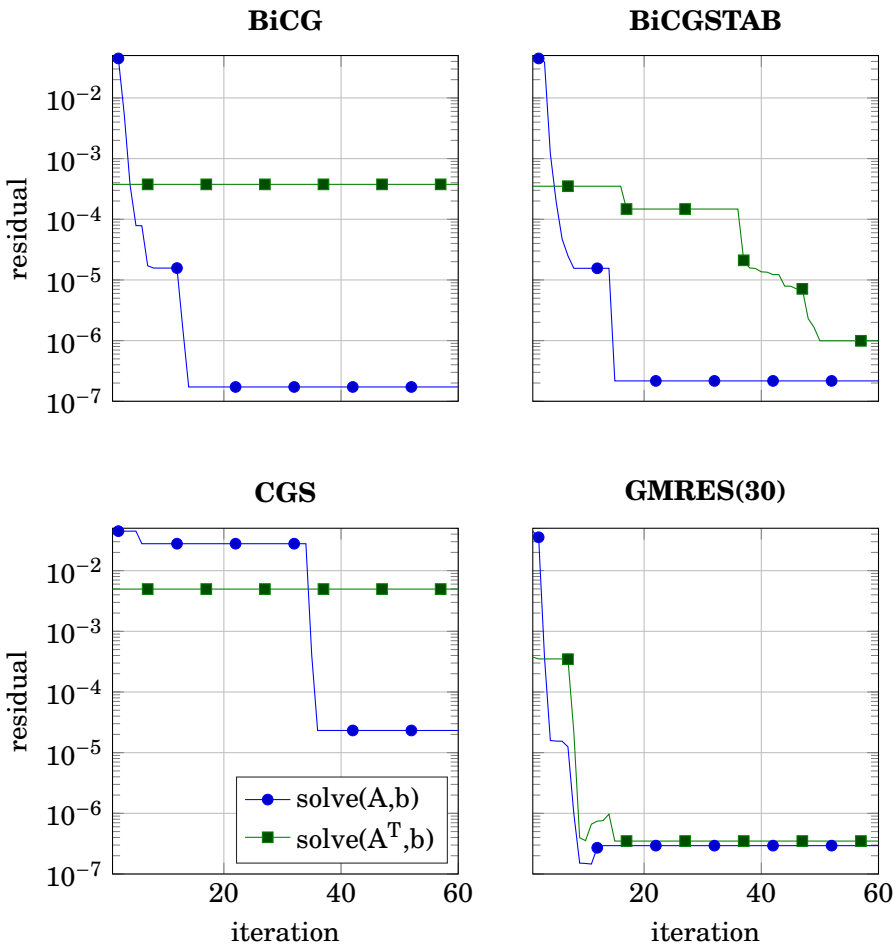


Figure 7.1: Convergence behaviour, with Cartesian basis vector e_6 as right hand side. Residual reduction differs between original and transposed systems.

BiCG	A	7	7	X	6	6	7	X	6	8	7	X	7	8	7	X	7	6	7	X	6	7	7	X	7	8
	A^T	8	3	Inf	0	7	3	10	0	7	3	10	0	7	3	10	0	7	3	Inf	0	7	3	11	0	8
BiCGSTAB	A	6	6	X	7	6	7	X	7	6	7	X	7	6	7	X	7	6	6	X	7	6	7	X	6	6
	A^T	8	6	Inf	6	7	6	10	6	7	7	10	6	7	6	10	7	7	6	Inf	7	7	6	11	6	8
CGS	A	7	6	X	0	6	5	X	0	7	7	X	3	8	6	X	1	7	6	X	0	8	4	X	0	9
	A^T	8	2	Inf	0	7	2	11	0	7	2	12	0	7	2	11	0	7	2	Inf	0	7	2	12	0	8
GMRES(30)	A	6	7	7	6	6	7	6	6	6	7	6	7	6	6	6	7	6	7	6	6	6	7	7	6	6
	A^T	8	6	13	6	7	6	10	6	7	6	10	7	7	6	10	7	7	6	13	6	7	6	11	6	8
GMRES(3)	A	6	4	0	4	6	3	0	3	6	3	0	3	6	3	0	3	6	4	0	3	6	4	0	3	6
	A^T	8	3	13	0	7	3	10	0	7	3	10	0	7	3	10	0	7	3	13	0	7	3	11	0	8

Table 7.1: Residual reduction, in orders of magnitude, where X denotes solver breakdown. The i-th column shows results for right hand side $b = e_i$. Cases that stall or break down are highlighted in gray. *Inf* indicates an output residual of exactly zero.

7.3 BiCG breakdown analysis

In the following, right hand side vectors that lead to Lanczos breakdown are identified for a 2×2 system and its transpose. Consider the following matrices

$$A := \begin{bmatrix} 1 & \frac{1}{2} \\ 0 & \frac{3}{2} \end{bmatrix} \quad A^T := \begin{bmatrix} 1 & 0 \\ \frac{1}{2} & \frac{3}{2} \end{bmatrix}.$$

The matrices are triangular, well-conditioned, diagonal-dominant and invertible. The normalised eigenvectors and eigenvalues of A are

$$v_1 := \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad v_2 := \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \qquad \lambda_1 := 1 \quad \lambda_2 := \frac{3}{2}.$$

and those of A^T are

$$v_1 := \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad v_2 := \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix} \qquad \lambda_1 := \frac{3}{2} \quad \lambda_2 := 1.$$

If a zero initial guess $\vec{y}_0 = 0$ is used, every step of BiCG is linear in the right hand side. It is therefore sufficient to study unit-length right hand side vectors defined as

$$\vec{b} = \begin{bmatrix} \cos(\alpha) \\ \sin(\alpha) \end{bmatrix}, \qquad \alpha \in [0 \ldots 360).$$

In two dimensions, the right hand side vectors can be visualised as points on the unit circle, and the solution vectors can be visualised as points on an ellipse that, when the matrix is applied to them, are projected onto the unit circle. If BiCG does not break down, it converges in two iterations. This is shown for some right hand side vectors in Figure 7.2. To find right hand sides that can cause Lanczos breakdown, one can plot ρ over the direction of the unit-length right hand side vector to identify angles where ρ becomes zero, as shown in Figure 7.3. With this, one finds that all multiples of 45° are candidates for breakdown. These angles correspond to the eigenvectors of A and A^T.

Indeed, when one uses the eigenvectors of A and A^T as right hand sides, BiCG either finds a solution in one iteration, or breaks down. In particular, when solving $A \cdot \vec{y} = b$, BiCG succeeds if b is an eigenvector of A, and breaks if b is an eigenvector of A^T. Conversely, when solving $A^T \cdot \vec{y} = b$, BiCG succeeds if b is an eigenvector of A^T, and breaks if b is an eigenvector of A. This is shown in Figure 7.4. It has been observed in previous work that BiCG remains remarkably stable and accurate near breakdown points, as long as the breakdown point is not precisely hit [155]. This was indeed found to be the case here.

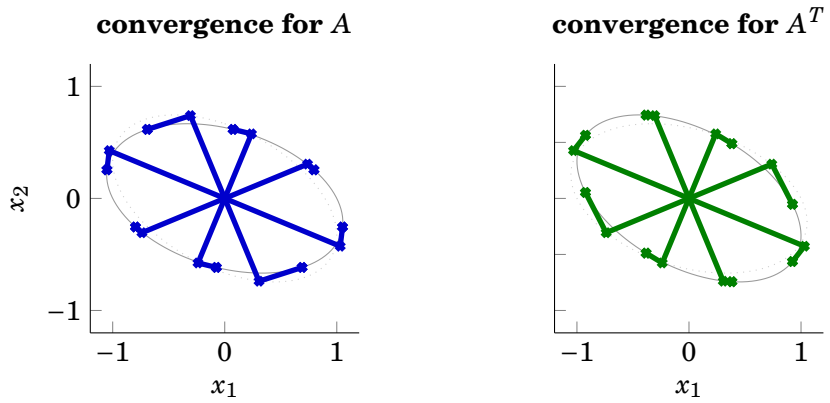


Figure 7.2: Iterations performed by BiCG to solve $A \cdot \vec{y} = b$ (left) and $A^T \cdot \vec{y} = b$ (right), starting from a zero initial guess, with unit-length right hand sides. The first iterate is always identical for both systems, while the second iteration finds the solution in an update step that is tangential to the ellipse.

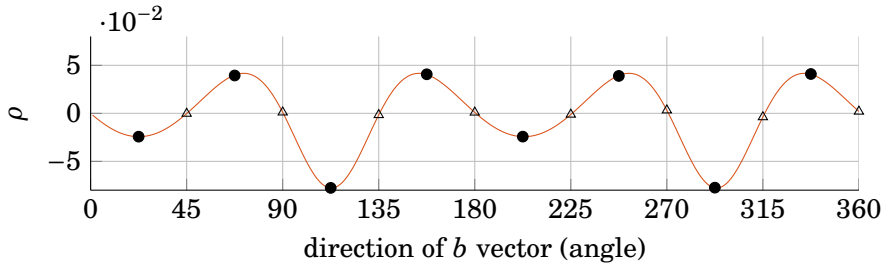


Figure 7.3: Lanzcos breakdown ($\rho = 0$) occurs if right hand side is a unit vector in direction marked with triangles. BiCG behaviour for these cases is shown in Figure 7.4. For other angles, BiCG works correctly (Figure 7.2 shows convergence for angles marked with circles).

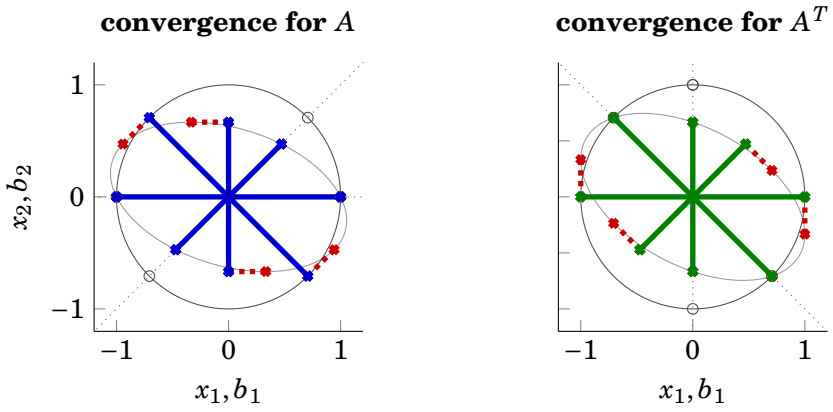


Figure 7.4: Breakdown in BiCG if eigenvectors of either A or A^T are used as right hand sides. BiCG steps after one iteration, after which it finds a solution ($r = 0$), or breaks down ($\hat{r} = 0$). Thick dotted lines show the necessary update that is not performed because of breakdown. Thin dotted lines show directions of eigenvectors.

7.4 Differentiated linear solvers and roundoff

Roundoff errors influence the behaviour of linear solvers. Previous work found that the derivatives of Krylov solvers are more severely affected by numerical noise if they are computed using AD than using finite differences [112]. This section aims to investigate roundoff errors in derivatives of Krylov solvers that are computed either using the high-level differentiation shown in Section 2.3, using AD in forward or reverse mode, and using finite differences.

The CG method is investigated here, as it is guaranteed to converge and hence after the required number of iterations, any remaining residual is caused by roundoff. Hilbert matrices [78] are used, where the n-th Hilbert matrix is

$$H^n_{i,j} = \frac{1}{i+j-1} \quad \forall \quad i,j \in \{1 \dots n\}.$$

The Hilbert matrix is SPD and therefore CG can be applied. Its condition number grows exponentially with n, hence it is a test cases that provokes large roundoff errors for small problem sizes. Its inverse is symmetric, all integer, and given explicitly by

$$(H^n_{i,j})^{-1} = (-1)^{i+j}(i+j-1)\binom{n+i-1}{n-j}\binom{n+j-1}{n-i}\binom{i+j-2}{i-1}^2.$$

Provided that it converges, applying CG is equivalent to a multiplication with the inverse matrix as in

$$\mathrm{CG}(H^n, b) := (H^n)^{-1} \cdot b,$$

and the tangent-linear and adjoint models are therefore known explicitly as

$$\bar{b} = \dot{y} = (H^n)^{-1} \cdot \dot{b} = (H^n)^{-1} \cdot \bar{y}.$$

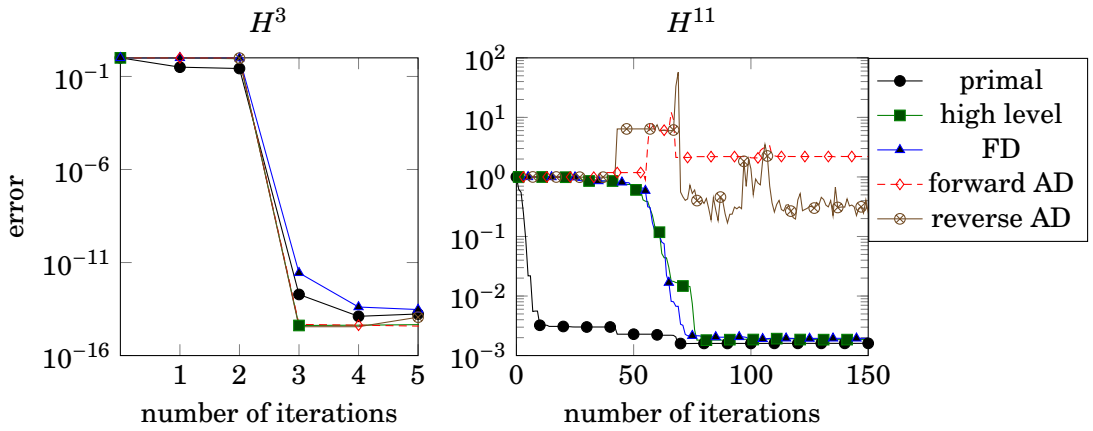


Figure 7.5: **Left:** All methods are correct within machine accuracy on H^3. **Right**: The primal reduces the residual, as do finite differences and high-level differentiation work with some delay. AD code fails due to roundoff.

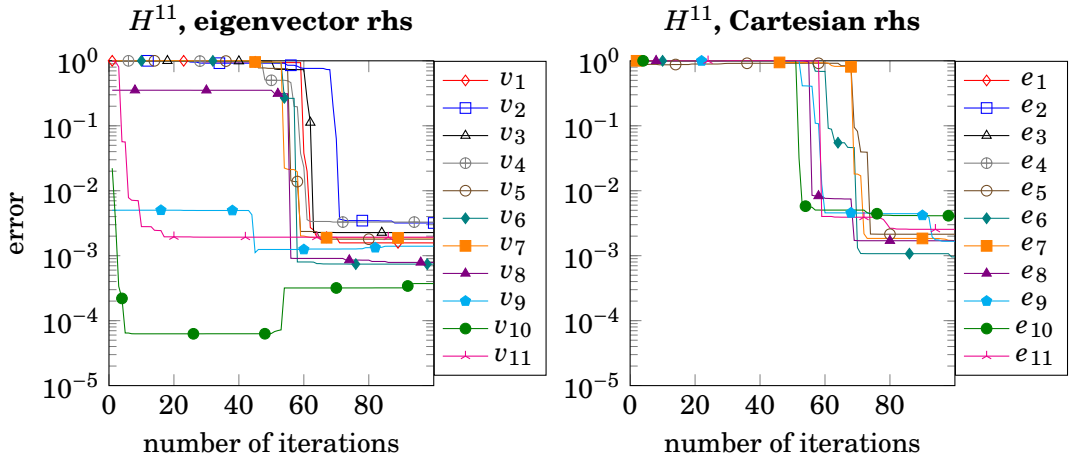


Figure 7.6: **Left:** For eigenvector right hand sides that correspond to large eigenvalues (v_9, v_{10}, v_{11}), CG starts to reduce the residual from the beginning. **Right**: CG with Cartesian basis vector right hand sides shows delayed convergence.

CG is applied to a 3×3 and a 11×11 Hilbert matrix, with the right hand side arbitrarily chosen as $b = [1,2,3\ldots]$. Next, the derivatives of CG obtained by high-level forward differentiation, Tapenade forward and reverse differentiation, and finite differences, are seeded with unit vectors to compute rows of the inverse Hilbert matrix. The results in Figure 7.5 show that for the H^3 matrix, which has a condition number of 748, the error of the primal solve and its derivatives drop close to machine precision in the 3^{rd} iteration. Perhaps surprisingly, the error is further reduced in the 4^{th} iteration, which would result in a division by zero in exact arithmetic. For H^{11} (condition number 10^{15}), primal CG reduces the residual by more than 2 orders of magnitude, while the AD-generated code fails to produce a meaningful result. Hand-differentiation and finite differences yield the same final accuracy as the primal, but require more iterations.

Since the hand-differentiated code and finite differences use CG on the original matrix with seed vector right hand sides, the delayed convergence indicates an influence of the right hand side on the roundoff error. This can be confirmed by using eigenvector and Cartesian basis vector right hand sides, which causes a delay in convergence, as shown in Figure 7.6. The delay is particularly severe if eigenvectors are used that correspond to small eigenvalues.

To confirm that roundoff is responsible for this behaviour, a symbolic code verification tool [145] was used to prove the correctness of the CG solver and the derivatives written by hand and generated with Tapenade under the assumption of exact arithmetic. This shows that AD-generated code can suffer from roundoff to a different extent than the underlying primal code, and that the structure of right hand side vectors can influence the error of solvers that are guaranteed to converge in exact arithmetic.

7.5 Using adjoint linear solvers in practice

Following the observations on the convergence of Krylov solvers in the preceding sections, this final section aims to point out their implication on adjoint computations in practice.

Section 7.4 showed that Krylov solvers are particularly affected by roundoff errors for ill-conditioned matrices and some right hand side vectors. Since the right hand sides differ between primal and adjoint systems, their convergence behaviour may differ as well. Roundoff errors can often be reduced or at least detected using higher precision or interval arithmetic [62], which leads to an increase in computational cost, memory footprint, and data transfer between memory and processor. For many-core architectures, it seems more promising to compute certain intermediate results, such as dot-products, with algorithms that minimise roundoff at the cost of a slight increase in the number of operations [124]. Such an approach does not require larger floating point numbers, does not increase the memory footprint or data transfers between host and MIC system, and could therefore improve the precision and reliability of linear solvers without significantly increasing the computation time. Using such algorithms from within code sections that are differentiated with AD would probably require a high-level treatment similar to that shown in Section 2.3 to retain their error-reducing properties.

As seen in Section 7.3, even in the absence of floating-point errors, Krylov solvers may break, converge at different rates, or to a different final residual, depending on the right hand side or transposition of the system matrix. Right hand side vectors with a particular structure (Cartesian or Eigenvectors) seem more likely to cause problems than uniformly distributed vectors, and at the same time seem more likely to occur in derivative computations, for example because of seeding. Further research is needed to understand the implications of matrix transposition and right hand side structure on the convergence of Krylov solvers. In the meantime, one could use GMRES with high restart values, which was found to have a predictable behaviour as shown in Table 7.1, or direct solvers. If the computational cost or memory footprint of such methods is not acceptable, transformation techniques or preconditioning could be used to avoid difficult-to-solve right hand sides. Note that such an approach would lead to different preconditioners in the primal and adjoint system.

The chapter serves as a reminder that one can not blindly use the same iteration number for primal and adjoint linear systems, and it is necessary to use customised convergence criteria for both systems. AD tools could be improved to provide more assistance in this task, for example by providing building-blocks for high-level differentiation of linear solvers including callback routines that perform convergence checks. Finally, there is good news for AD developers: If adjoint results seem wrong or unphysical, or fail the dot product test, perhaps the linear solver is wrong, and the AD tool is right.

CHAPTER 8

Conclusion

This thesis has the aim of improving parallelism, computation time, and memory footprint of adjoint computations performed by code that is generated using reverse-mode algorithmic differentiation, particularly on shared-memory architectures such as modern multi-core CPUs and Intel's many-core accelerator cards. The work resulted in a novel method for activity analysis, two novel methods for the reverse-differentiation of parallel code, and a new method for checkpoint compression.

8.1 Activity analysis

The thesis investigates *multi-activity differentiation*, a refined strategy for static activity analysis. Based on a context-sensitive analysis and user input, the method employs procedure cloning and specialisation to generate more efficient derivative code. The method was implemented in the algorithmic differentiation tool Tapenade and is available from version Tapenade version 3.11.

It is demonstrated in a case study that multi-activity differentiation can provide substantial benefits in the adjoint runtime, at virtually no cost in memory footprint. The cost in terms of derivative code size and differentiation time is significant, but typically less important than the performance gains in the generated derivative code. While multi-activity differentiation is shown to improve performance in many circumstances, it produces unnecessary derivative code in other cases. A user can intervene and manually use command line options and pragmas to avoid this. In the future, this could be addressed with a more sophisticated strategy that only creates specialised procedures if this actually leads to a smaller number of operations.

The specialisation for procedures is possible because each procedure can be accessed from multiple call sites with different activities. Likewise, a code portion after an if/else block can be reached with different activities depending on the branch chosen in the

preceding block. The same holds true for code that can be reached through return, cycle, goto or similar statements. A specialisation of such code sections was investigated, but not implemented in Tapenade, as it was found to result in an excessive growth of derivative code size. If this is to be developed in the future, an intuitive method is needed so that a user of an AD tool can selectively specialise only certain control flow blocks. Alternatively, a heuristic is needed that specialises only those code blocks where specialisation is most beneficial.

If some code sections are not specialised, there can be more than one possible way to link differentiated procedures to call sites. The current behaviour of Tapenade is implementation-defined and arbitrary. This could be improved in the future to choose the most cost-effective specialisation.

Even with the improvements presented in this work, the activity analysis has to make conservative assumptions, as the precise activity is in some cases only known at runtime. In the future, AD tools could create specialised versions of certain derivative procedures, and add statements that decide at runtime which of the created code versions is most efficient and still legal to use.

8.2 Shared-memory parallelism for reverse-mode AD

The work presents two methods, referred to as *SSMP* and *RSMP*, that preserve the parallelisation of a primal computation and generate an adjoint code that uses the same parallelisation strategy. SSMP was developed to generate adjoints of unstructured OpenMP-parallel stencil operations. The method is successfully applied to a fluid dynamics solver that supports shared memory parallelism on multi-core CPUs and many-core architectures such as the Intel XeonPhi. The parallelisation of the adjoint code that is automatically generated with SSMP matches that of the primal solver. Since the computation is structured essentially in the same way as in the primal code, there was reason to believe that the generated adjoint code would be as scalable as the primal code. This fact is indeed confirmed in experiments, where the scaling of the parallel adjoint solver even exceeds that of the primal solver in most cases. Furthermore, it is demonstrated for a small number of cases that SSMP performs significantly better than other approaches described in the literature that require OpenMP `atomic` pragmas or `critical` sections.

Prerequisites are defined that have to be met by a parallel loop for SSMP to be applicable, and the correctness of the generated parallel adjoint code is proven. This is an important step towards the integration of SSMP in algorithmic differentiation tools. Since the parallelisation of the primal code is usually the result of careful manual optimisation, the possibility of generating an equally scalable adjoint code makes SSMP

attractive for a wide range of applications. In the current implementation, suitable code portions need to be identified by hand. The build process itself is automated so that changes in the primal code are correctly incorporated into the adjoint solver.

RSMP can be applied to primal parallel code for which a conventional AD approach would lead to an adjoint loop that can not be parallelised without synchronisation pragmas that impair the parallel efficiency. The loops that RSMP can be applied to are commonly found in structured-mesh PDE solvers. Adjoints generated with RSMP scale as well as the corresponding primal codes, albeit with a constant-factor performance penalty that is caused by duplicated calls to specialised derivative procedures. The overhead of these duplicate calls is shown to be outweighed by the improved scalability, particularly on accelerators. As an added benefit, if the primal code fits the definition of a stencil code, the corresponding adjoint codes produced by RSMP are also stencil codes, which facilitates the use of the many optimisation strategies for this kind of computation, e.g. [30, 75, 80, 152, 158].

In the future, it would be beneficial to integrate SSMP into an AD tool along with an automated detection of self-adjoint memory access to make this approach available to a wider audience. For many practical cases, it is sufficient to detect whether the same variables are used as indices in the input and output vectors, and whether those variables are left unchanged between the read and write operations. For RSMP, an implementation in an AD tool is less straightforward, and probably needs to rely on some user input or runtime information, at least to ensure that the reflexivity condition presented in Definition 3 holds.

8.3 Low-cost checkpoint compression

It is shown that *incomplete checkpointing* is a straightforward and effective way of reducing the memory footprint of unsteady adjoint calculations. Incomplete checkpointing can be implemented by leaving gaps in the stored flow trajectory, with an effect on the sensitivity accuracy that is acceptable for many industrial cases even for relatively large gap sizes. This makes incomplete checkpointing worth considering as an alternative to lossy checkpoint compression, as it has a low computational cost and is straightforward to implement. Incomplete checkpointing can also be used to reduce the amount of data that needs to be transferred between an accelerator and the host memory, if the trajectory is stored on the host system.

The error introduced by storing a trajectory that is coarsened in time is much smaller than the error introduced by under-resolving the physical time during the primal flow computation. It is therefore preferable to perform the primal simulation with a fine temporal resolution and to use incomplete checkpointing, compared to the alternative of

reducing the number of primal time steps to a number that fits into memory. Spatial coarsening is presented as a way not only to reduce the memory footprint, but also to improve the stability of unsteady adjoint computations for simulations of flow at the onset of chaotic behaviour. The improved stability is the result of the smoothing introduced by the interpolation, which seems to remove some chaotic features of the flow. It could be questioned whether the adjoint results based on such a smoothened flow field are still useful in the presence of chaotic flow. At least in the context of shape optimisation, it is conceivable that the chaotic effects are only present for some non-optimal shapes, and that even slightly inaccurate gradients are acceptable if they prevent a breakdown of the optimisation process.

An error estimation strategy for incomplete checkpointing could help in choosing the gap size in the future, and to adapt the storing interval during the primal simulation and the restoration order during the adjoint simulation dynamically, e.g. to capture some flow features with a higher accuracy. This would probably require a more rigorous formulation of the checkpoint coarsening and interpolation mechanism.

8.4 Reliability of linear solvers in adjoint computations

Case studies on iterative linear solvers show that errors caused by limited floating point accuracy can affect the derivatives to a greater extent than the primal solver, and that convergence of the primal system does not guarantee convergence of the adjoint system even in exact arithmetic for most Krylov solvers. It is more common to use single or mixed precision computations on accelerators, as the performance difference between double and single precision operations is more pronounced on these systems. Direct solvers or GMRES without restarts are not practical on systems with limited memory, and most direct solvers do not scale on many-core architectures. However, this work shows that the available solvers with small memory footprint and high parallel efficiency can not always be safely used on linear systems that arise in adjoint solvers, and that rapid convergence in the primal system does not imply convergence in the corresponding adjoint system.

Further research is needed to identify situations in which adjoint derivatives are significantly more susceptible to roundoff errors, solver breakdown and convergence problems than the primal. This would include research on the role of right hand side vectors in the convergence of Krylov solvers.

8.5 Perspective

While this work presents some progress towards the successful reverse-differentiation of parallel code, many challenges remain. For functions that do not have a symmetric Jacobian matrix, and hence SSMP and RSMP are not applicable, AD tools need to resort to some form of automatic parallelisation, which is known to result in poor performance in most cases. This could be fixed with some intuitive method that allows user-defined parallelism such as domain decomposition methods to be automatically built into the adjoint code.

A method related to RSMP could be used to reorder the contributions to each index in the result vector in a way that minimises roundoff errors [77]. This could be done in parallel for different indices of the result vector.

Highly optimised code for many-core processors often uses vector instructions [86], compute kernels written in other languages [38] or written with vendor-specific commands [111], or software libraries such as BLAS [97] or Intel MKL [167]. It is hard for AD tools to support all of these developments, and it might be beneficial if AD tools could be more easily adapted to new languages and software packages, for example by making them more modular or allowing some sort of high-level scripting to customise the AD process and introduce new language constructs to the AD tool.

Some problems related to activity analysis, pointer analysis, differentiation of object-oriented code, and differentiation of parallel code, could be solved more easily with runtime information, and it might be beneficial to use techniques such as runtime code adaptation [105] and just-in-time compilation [6] to produce faster derivative codes.

Finally, roundoff errors and code reliability are likely to become an even greater concern than they already are today. If the original code itself is already difficult to debug because of the combined use of message-passing, OpenMP pragmas, libraries, vector intrinsics etc., a user of an AD tool can not be expected to understand or manually debug the generated adjoint code, and more work is needed to allow at least a partially automated verification of generated derivative code.

List of Symbols

$\mathscr{A}$	activity pattern
$\mathscr{A}_c$	activity pattern of called procedure
$\mathbb{A}$	set of activity patterns for procedure
$\mathbb{A}_g$	generalised set of activity patterns
$\mathbb{D}$	set of differentiation heads for procedure
$\mathbb{A}_s$	specialised set of activity patterns
$\mathbb{A}_{gs}$	partially specialised set of activity patterns
$\mathscr{U}$	set of useful variables
$\mathscr{V}$	set of varied variables
k	cell face number
Ω	control volume
x	coordinate vector
x_ζ	cell centre coordinate
x_k	k-th face centre coordinate
x_{nk}	centre coordinate of neighbour cell at k-th face
e	mesh edge
c	edge colour
l	edge left/first node index
r	edge right/second node index
$edg2nde$	mapping from edge to pair of nodes that are connected by this edge
N	number of mesh cells
$n_{colours}$	number of mesh colours
n_{edges}	number of mesh edges
n_{nodes}	number of mesh edges
$\vec{n}$	cell face normal vector

S	control volume surface
S_k	surface of k-th control volume face
f	edge flux (contribution to F for a single edge)
F	function to compute nonlinear residual
f_l	edge flux, left/first node
f_r	edge flux, right/second node
$\vec{\mathscr{P}}$	preconditioning matrix for nonlinear system
$\vec{R}$	global discrete nonlinear residual vector
$\tilde{t}$	pseudo time step (iteration of nonlinear solver)
$\vec{U}$	global discrete state vector
$\vec{X}$	geometric mesh properties
$\vec{f}_e$	body force
ρ	density
E	energy
H	enthalpy
κ	heat conductivity
q_h	heat source
p	pressure
h	step size
t	simulated time (physical time in simulation)
$\vec{\tau}$	stress tensor
Δt	time step size (resolution of temporal discretisation)
k_t	thermal conductivity
$\vec{v}$	velocity vector
c_t	temporal coarsening ratio
c_x	spatial coarsening ratio
$\vec{M}_P$	prolongation matrix
$\vec{M}_R$	restriction matrix
t^-	next stored time step for which $t^+ < t$
t^+	next stored time step for which $t^+ > t$
$\mathscr{T}_s$	set of physical time steps selected in coarsening
$\mathscr{T}$	set of physical time steps

$\vec{\alpha}$	design vector
$\downarrow$	procedure input
J	objective function result
m	dimension of output vector
n	dimension of input vector
n_{args}	number of procedure arguments
rk	procedure argument index
$n_{threads}$	number of parallel threads
ω	cost function time averaging
$\uparrow$	procedure output
P	procedure, subroutine, or function in some program
P_0	top procedure of differentiation
c_{in}	passive input
c_{out}	passive output
T	primal code runtime (wall clock time)
I_∞	last instruction in a procedure
I	instruction in a procedure
I_0	first instruction in a procedure
I_c	callsite, instruction that calls a procedure
I_-	predecessor instruction
I_+	successor instruction
C	set of instructions that call this procedure
C_s	set of instructions that call this procedure, marked for specialisation
K	iteration in loop over edges
M	set of memory addresses/set of indices in vector
M_{in}	set of read-only memory addresses/set of indices in vector
M_{out}	set of write-only memory addresses/set of indices in vector
v	meaningless variable for use in code snippets and examples
$\mathscr{S}$	set of variables
ζ	array index for local node
k_m	array index
k_n	array index

nde2neigh	list of neighbouring nodes for given centre node
v	array index for neighbour node
w	meaningless variable for use in code snippets and examples
ϵ	linear solver tolerance, error after final iteration
$\vec{y}_0$	system initial guess
i	linear solver iteration
$\vec{A}$	system matrix
$\vec{b}$	system right hand side
$\vec{y}$	system solution

List of Figures

List of Tables

List of Algorithms

APPENDIX A

Overhead of OpenMP constructs

There are various ways of implementing reductions in OpenMP, with the colouring and atomics in Chapter 5 being just two out of many options. Different implementations come at a different cost. The overhead of certain OpenMP pragmas has been studied previously [52], but a case study is shown here that is more applicable to the situation in CFD codes, and uses a XeonPhi many-core machine.

Consider the following serial program, where a nested loop performs a summation into the array `summ`.

```
1  program main
2    use omp_lib
3    implicit none
4    integer :: i,j
5    double precision, dimension(24000) :: summ
6    double precision :: start_time
7    !----------------------------------------
8    summ(1:24000) = 0.0
9    start_time = omp_get_wtime()
10   do i=1,160000
11     do j=1,24000
12       summ(j) = summ(j)+1.0
13     end do
14   end do
15   write(*,*) "Runtime: ", omp_get_wtime()-start_time
16   ! prevent compiler from optimising everything away
17   write(*,*) maxval(summ)
18 end program
```

Compiled with the Intel Fortran compiler 16.0.2, using the flags `-O3 -fopenmp -mmic`, the program was run on an Intel XeonPhi 5110P coprocessor in native execution with 240 threads. The output of the program reflects the time spent within the nested loop in lines 10 – 14, in seconds. **Runtime: 45.6s**
The loop will now be parallelised using different strategies, and runtimes reported.

Reduction pragma

The OpenMP reduction pragma is the most natural way to parallelise the loop. By default, all variables are shared between threads. The loop counters are explicitly declared private, and a sum-reduction is declared on `summ`.

```
1 !$OMP PARALLEL PRIVATE(i,idx) REDUCTION(+:SUMM)
2 do i=1,16000
3   do j=1,24000
4     summ(j) = summ(j)+1.0
5   end do
6 end do
```

Runtime: 0.855, 53x speedup from serial
The `reduction` pragma requires that each thread will have its local copy of `summ` that must be initialised to the neutral element of the reduction operation in the beginning (e.g. 0, 1, *true* respectively for +,*,.and.), updated as specified by the statements inside the loop, and finally merged to obtain the contributions from all threads. In this example, the reduction pragma yields a good speedup, but the local copy of the result array increases the memory footprint of the program by a factor of more than 200. This is however not an issue for the above example program, which is small enough.

Parallelising inner loops

Alternatively, one could parallelise the inner loop, and let the outer loop run in serial. This does not produce race conditions, as the indices accessed by several threads do not overlap.

```
1 do i=1,16000
2   !$OMP PARALLEL DO
3   do j=1,24000
4     summ(j) = summ(j)+1.0
5   end do
6 end do
```

Runtime: 1.13s, 40x speedup from serial

While this produces the correct result, the `parallel do` pragma is encountered many times. Each time there is an overhead, caused by the fact that several threads have to be initialised.

Synchronising inner loops

Instead, the threads can be started once, and wait for all other threads to finish their iterations of the inner loop before all threads commence with the next outer iteration, as shown in the following example.

```
1 !$OMP PARALLEL PRIVATE(i,j)
2 do i=1,16000
3   !$OMP DO
4   do j=1,24000
5     summ(j) = summ(j)+1.0
6   end do
7 end do
8 !$OMP END PARALLEL
```

Runtime: 0.964s, 47x speedup from serial

Critical sections

Synchronising can also be limited to the statements that actually access shared memory regions. By declaring a `critical` section that can only be entered by one thread at a time, race conditions are avoided.

```
1 !$OMP PARALLEL DO PRIVATE(i,j)
2 do i=1,16000
3   do j=1,24000
4     !$OMP CRITICAL
5     summ(j) = summ(j)+1.0
6     !$OMP END CRITICAL
7   end do
8 end do
```

Runtime: 1158, 25.4x slowdown from serial

Critical sections require inter-thread communication and can cause threads to wait idle before the start of a critical section. In this example, almost all work is performed inside the critical section and therefore done in serial. The additional synchronisation overhead results in a parallel program that runs significantly slower than the serial program.

Other programs may still benefit from critical sections, if the cost of the computations outside these sections is significant.

Atomic operations

A less general, but less costly alternative critical sections is the use of `atomic` operations.

```
1 !$OMP PARALLEL DO PRIVATE(i,j)
2 do i=1,16000
3   do j=1,24000
4     !$OMP ATOMIC
5     summ(j) = summ(j)+1.0
6   end do
7 end do
```

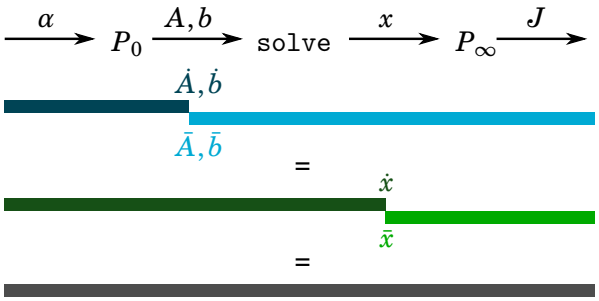
Runtime: 3.04, 15x speedup from serial

In this case, the runtime environment must ensure that the operations in line 5 happen without interference from other threads. Atomic pragmas may not be used for Fortran vector operations, and only for the following intrinsic functions and operators: `MAX`, `MIN`, `IAND`, `IOR`, `IEOR`, `+`, `*`, `-`, `/`, `.AND.`, `.OR.`, `.EQV.`, `.NEQV.`. The speed advantage of atomic operations over critical sections is not guaranteed by the standard, and depends on the compiler's ability to translate the program into machine code that does not use locks (which are used for critical sections). For older systems or more complex examples, atomic instructions may be as expensive as critical sections, as reported in [52].

APPENDIX B

The dot product test

The dot product test can be used to check the correctness of reverse-differentiated programs, assuming that correct tangent-linear derivatives are given. Applied to a linear solver and its derivatives, by definition of the tangent-linear and adjoint models, one must find that $\langle \dot{A}, \bar{A} \rangle + \langle \dot{b}, \bar{b} \rangle = \langle \dot{x}, \bar{x} \rangle$. An intuitive explanation is shown here: $\dot{A}, \dot{b}$ can be understood as the partial derivatives of P_0 multiplied with the tangent-linear seed, and $\bar{A}, \bar{b}$ are the reverse-accumulated partial derivatives through `solve` and P_∞ multiplied with the adjoint seed. Multiplying both parts yields the product of the derivatives of the entire program and both seeds. Likewise, $\dot{x}$ holds the forward-accumulated partials through P_0 and `solve`, whereas $\bar{x}$ holds the reverse-accumulated partials through P_1, which also upon multiplication yield the derivative of the entire program and both seeds.



APPENDIX C

Multi-activity differentiation without benefits

The adjoint computation for a with useful results x, y and varied inputs x, y, z requires the same number of operations with or without multi-activity differentiation.

```
1 subroutine a(n,x,y,z)
2   integer :: n
3   real, dimension(n) :: x,y
4   call f(n,x,y)
5   call f(n,y,z)
6 end
7
8 subroutine f(n,v,u)
9   integer :: n
10  real, dimension(n) :: u,v
11  u(1) = u(1)+v(1)
12  v(1) = u(1)
13 end
```

Differentiation with Tapenade without multi-activity differentiation (using command line flags `-b -head "a(x y)/(x y z)"`) results in the following.

```
1 !  Differentiation of a,
2 !  useful results: x y, varying inputs: x y z
3 SUBROUTINE A_B(n, x, xb, y, yb, z, zb)
4   ...
5   zb = 0.0 ! this instruction is removed by multi-activity
6   CALL F_B(n, y, yb, z, zb)
7   CALL F_B(n, x, xb, y, yb)
8 END SUBROUTINE A_B
9
```

```
10 !  Differentiation of f,
11 !  useful results: u v, varying inputs: u v
12 SUBROUTINE F_B(n, v, vb, u, ub)
13   ...
14   ub(1) = ub(1) + vb(1)
15   vb(1) = ub(1)
16 END SUBROUTINE F_B
```

With multi-activity differentiation (command line flags `-b -specializeActivity "%all%" -head "a(x y)/(x y z)"`), the output is the following. Note that the specialised routine `F_B0` has an additional zeroing statement, which makes the zeroing in `A_B` unnecessary, but has no beneficial effect on the number of operations performed by this program.

```
1 !  Differentiation of a,
2 !  useful results: x y, varying inputs: x y z
3 SUBROUTINE A_B(n, x, xb, y, yb, z, zb)
4   ...
5   CALL F_B0(n, y, yb, z, zb)
6   CALL F_B(n, x, xb, y, yb)
7 END SUBROUTINE A_B
8
9 !  Differentiation of f,
10 !  useful results: v, varying inputs: u v
11 SUBROUTINE F_B0(n, v, vb, u, ub)
12   ...
13   ub = 0.0 ! instead, this one is created.
14   ub(1) = ub(1) + vb(1)
15   vb(1) = ub(1)
16 END SUBROUTINE F_B0
17
18 !  Differentiation of f,
19 !  useful results: u v, varying inputs: u v
20 SUBROUTINE F_B(n, v, vb, u, ub)
21   ...
22   ub(1) = ub(1) + vb(1)
23   vb(1) = ub(1)
24 END SUBROUTINE F_B
```

APPENDIX D

Encapsulation and multi-activity: example

This is a working example for encapsulation issue discussed in Section 4.8.

```
1 module M
2   real y
3   contains
4   subroutine f(x)
5     real x
6     x = x*y
7   end subroutine
8 end module
9
10 subroutine b(x)
11   use M, only: f
12   real x
13   call f(x)
14 end subroutine
```

The derivative procedure $\dot{b}$ must initialise $\dot{y}$, even though y is not used or visible inside the primal procedure b. In an object-oriented setting, y could be declared as private, while $\dot{y}$ must be public. At the time of writing, Tapenade is unable to differentiate this code correctly without using multi-activity differentiation. Following the discovery of this problem, it displays a warning message, asking the user to change the adjoint code manually.

The following command line flags can be used to correctly differentiate the above example with Tapenade in forward mode.

```
-d -specializeActivity "%all%" -head "f(x y)\(x y) b(x)\(x)"
```

Bibliography

[1] AKBARZADEH, S., HÜCKELHEIM, J., AND MÜLLER, J.-D.
Consistent treatment of incompletely converged iterative linear solvers in reverse-mode algorithmic differentiation.
Optimization Methods and Software (submitted).

[2] ALBRING, T. A., SAGEBAUM, M., AND GAUGER, N. R.
Development of a consistent discrete adjoint solver in an evolving aerodynamic design framework.
In *16th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference* (2017/02/20 2015), American Institute of Aeronautics and Astronautics.

[3] ANDERSON, W. K., AND VENKATAKRISHNAN, V.
Aerodynamic design optimization on unstructured grids with a continuous adjoint formulation.
Computers & Fluids 28, 4 (1999), 443–480.

[4] ASOUTI, V., KONTOLEONTOS, E., TROMPOUKIS, X., AND GIANNAKOGLOU, K.
Shape optimization using the one–shot adjoint technique on graphics processing units.
In *7th GRACM International Congress on Computational Mechanics Conference* (2011), vol. 30.

[5] ASPDEN, A., NIKIFORAKIS, N., DALZIEL, S., AND BELL, J.
Analysis of implicit LES methods.
Communications in Applied Mathematics and Computational Science 3, 1 (2009), 103–126.

[6] AYCOCK, J.
A brief history of just-in-time.
ACM Computing Surveys (CSUR) 35, 2 (2003), 97–113.

[7] BAKER, A. H., JESSUP, E. R., AND KOLEV, T. V.
A simple strategy for varying the restart parameter in GMRES.

Journal of Computational and Applied Mathematics 230, 2 (2009), 751–761.

[8] BALAY, S., GROPP, W. D., MCINNES, L. C., AND SMITH, B. F.
PETSc 2.0 users manual.
Tech. rep., Argonne National Laboratory, 1996.

[9] BANK, R. E., AND CHAN, T. F.
An analysis of the composite step biconjugate gradient method.
Numerische Mathematik 66, 1 (1993), 295–319.

[10] BARRETT, R., BERRY, M. W., CHAN, T. F., DEMMEL, J., DONATO, J., DONGARRA, J., EIJKHOUT, V., POZO, R., ROMINE, C., AND VAN DER VORST, H.
Templates for the solution of linear systems: building blocks for iterative methods, vol. 43.
SIAM, 1994.

[11] BAYDIN, A. G., PEARLMUTTER, B. A., AND RADUL, A. A.
Automatic differentiation in machine learning: a survey.
arXiv preprint arXiv:1502.05767 (2015).

[12] BENZI, M.
Preconditioning techniques for large linear systems: a survey.
Journal of computational Physics 182, 2 (2002), 418–477.

[13] BERGGREN, M.
Numerical solution of a flow-control problem: vorticity reduction by dynamic boundary action.
SIAM Journal on Scientific Computing 19, 3 (1998), 829–860.

[14] BISCHOF, C., CARLE, A., CORLISS, G., GRIEWANK, A., AND HOVLAND, P.
ADIFOR–generating derivative codes from Fortran programs.
Scientific Programming 1, 1 (1992), 11–29.

[15] BISCHOF, C. H., BÜCKER, H. M., LANG, B., RASCH, A., AND VEHRESCHILD, A.
Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs.
In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)* (Los Alamitos, CA, USA, 2002), IEEE Computer Society, pp. 65–72.

[16] BISCHOF, C. H., ROH, L., AND MAUER, A.
ADIC — An extensible automatic differentiation tool for ANSI-C.
Software–Practice and Experience 27, 12 (1997), 1427–1456.

[17] BLAZEK, J.
Computational Fluid Dynamics: Principles and Applications.
Elsevier Science, 2001.

[18] BLOM, F. J.
Considerations on the spring analogy.
International Journal for Numerical Methods in Fluids 32, 6 (2000), 647–668.

[19] BORKAR, S.
Thousand core chips: a technology perspective.
In *Proceedings of the 44th Annual Design Automation Conference* (New York, NY, USA, 2007), DAC 2007, ACM, pp. 746–749.

[20] BRIGGS, W., HENSON, V., AND MCCORMICK, S.
A Multigrid Tutorial, second ed.
Society for Industrial and Applied Mathematics, 2000.

[21] BÜCKER, H. M., LANG, B., AN MEY, D., AND BISCHOF, C. H.
Bringing together automatic differentiation and OpenMP.
In *Proceedings of the 15th International Conference on Supercomputing* (New York, NY, USA, 2001), ICS 2001, ACM, pp. 246–251.

[22] BÜCKER, H. M., LANG, B., RASCH, A., BISCHOF, C. H., AND AN MEY, D.
Explicit loop scheduling in OpenMP for parallel automatic differentiation.
In *Annual International Symposium on High Performance Computing Systems and Applications* (2002), vol. 16, pp. 121–126.

[23] BÜCKER, H. M., RASCH, A., AND WOLF, A.
A class of OpenMP applications involving nested parallelism.
In *Proceedings of the 2004 ACM Symposium on Applied Computing* (2004), ACM, pp. 220–224.

[24] BUHL, T., PEDERSEN, C. B., AND SIGMUND, O.
Stiffness design of geometrically nonlinear structures using topology optimization.
Structural and Multidisciplinary Optimization 19, 2 (2000), 93–104.

[25] BUTCHER, J. C.
The numerical analysis of ordinary differential equations: Runge-Kutta and general linear methods.
Wiley-Interscience, 1987.

[26] CAMELLI, F., LÖHNER, R., AND MESTREAU, E.
Timings of an unstructured-grid CFD code on common hardware platforms and compilers.
In *45th AIAA Aerospace Sciences Meeting and Exhibit* (2017/02/20 2007), American Institute of Aeronautics and Astronautics.

[27] CANTRELL, B., REITZ, R., RUTLAND, C., WANG, C., LIANG, L., PUDUPPAKKAM, K., AND MEEKS, E.
Performance of an OpenMP parallel flow solver for engine CFD simulations.
In *International Multidimensional Engine Modeling User's Group Meeting 2012* (2012).

[28] CAPRIOTTI, L.
Fast Greeks by algorithmic differentiation.
SSRN 14(3) (2010).

[29] CHRISTAKOPOULOS, F., JONES, D., AND MÜLLER, J.-D.
Pseudo-timestepping and verification for automatic differentiation derived CFD codes.
Computers & Fluids 46, 1 (2011), 174 – 179.

[30] CHRISTEN, M., SCHENK, O., AND BURKHART, H.
PATUS: a code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures.
In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2011).

[31] CHRISTIANSON, B.
Reverse accumulation and attractive fixed points.
Optimization Methods and Software 3, 4 (1994), 311–326.

[32] CHRISTIANSON, B.
Reverse accumulation and implicit functions.
Optimization Methods and Software 9, 4 (1998), 307–322.

[33] COLE-MULLEN, H., LYONS, A., AND UTKE, J.
Storing versus recomputation on multiple DAGs.
In *Recent Advances in Algorithmic Differentiation*, S. Forth, P. Hovland, E. Phipps, J. Utke, and A. Walther, Eds., vol. 87 of *Lecture Notes in Computational Science and Engineering*. Springer, Berlin, 2012, pp. 197–207.

[34] CORMEN, T., LEISERSON, C., RIVEST, R., AND STEIN, C.
Introduction To Algorithms.
MIT Press, 2001.

[35] DAGUM, L., AND ENON, R.
OpenMP: an industry standard API for shared-memory programming.
Computational Science & Engineering, IEEE 5, 1 (1998), 46–55.

[36] DAVIS, T. A., AND HU, Y.
The University of Florida sparse matrix collection.
ACM Trans. Math. Softw. 38, 1 (2011), 1.

[37] DISKIN, B., THOMAS, J. L., NIELSEN, E. J., NISHIKAWA, H., AND WHITE, J. A.
Comparison of node-centered and cell-centered unstructured finite-volume discretizations: Viscous fluxes.
AIAA journal 48, 7 (2010), 1326–1338.

[38] DU TOIT, J., LOTZ, J., AND NAUMANN, U.
Adjoint algorithmic differentiation of a GPU accelerated application.
Tech. rep., NAG, 2013.

[39] DWIGHT, R. P., AND BREZILLON, J.
Effect of approximations of the discrete adjoint on gradient-based optimization.
AIAA journal 44, 12 (2006), 3022–3031.

[40] ECKERT JR, J., MAUCHLY, J., GOLDSTINE, H., AND BRAINERD, J.
Description of the ENIAC and comments on electronic digital computing machines.
Tech. rep., DTIC Document, 1945.

[41] ECONOMON, T. D., PALACIOS, F., ALONSO, J. J., BANSAL, G., MUDIGERE, D., DESHPANDE, A., HEINECKE, A., AND SMELYANSKIY, M.
Towards high-performance optimizations of the unstructured open-source SU2 suite.
AIAA Paper Vol. 1949 (2015).

[42] ERDŐS, P., AND WILSON, R. J.
On the chromatic index of almost all graphs.
Journal of Combinatorial Theory, series B 23, 2-3 (1977), 255–257.

[43] ERIKSSONT, J., GULLIKSSON, M., LINDSTRÖM, P., AND WEDIN, P.-Å.
Regularization tools for training large feed-forward neural networks using automatic differentiation.
Optimization Methods and Software 10, 1 (1998), 49–69.

[44] ERLEBACHER, G., HUSSAINI, M., SPEZIALE, C., AND ZANG, T. A.
Toward the large-eddy simulation of compressible turbulent flows.
Journal of Fluid Mechanics 238 (1992), 155–185.

[45] FAGAN, M., AND CARLE, A.
Activity analysis in ADIFOR: Algorithms and effectiveness.
Tech. Rep. TR04-21, Department of Computational and Applied Mathematics, Rice University, Houston, TX, 2004.

[46] FARRELL, P. E., HAM, D. A., FUNKE, S. W., AND ROGNES, M. E.
Automated derivation of the adjoint of high-level transient finite element programs.
SIAM Journal on Scientific Computing 35, 4 (2013), C369–C393.

[47] FERZIGER, J. H., AND PERIC, M.
Computational methods for fluid dynamics.
Springer Science & Business Media, 2012.

[48] FETTWEIS, G., AND ZIMMERMANN, E.
ICT energy consumption-trends and challenges.
In *Proceedings of the 11th International Symposium on Wireless Personal Multimedia Communications* (2008), vol. 2.

[49] FÖRSTER, M.
Algorithmic Differentiation of Pragma-Defined Parallel Regions: Differentiating Computer Programs Containing OpenMP.
PhD thesis, RWTH Aachen, 2014.

[50] FÖRSTER, M., NAUMANN, U., AND UTKE, J.
Toward adjoint OpenMP.
Tech. Rep. AIB-2011-13, RWTH Aachen, Software and Tools for Computational Engineering, 2011.

[51] FRANSEN, R.
LES based aerothermal modeling of turbine blade cooling systems.
PhD thesis, Institut National Polytechnique de Toulouse-INPT, 2013.

[52] FREDRICKSON, N. R., AFSAHI, A., AND QIAN, Y.
Performance characteristics of OpenMP constructs, and application benchmarks on a large symmetric multiprocessor.
In *Proceedings of the 17th annual international conference on Supercomputing* (2003), ACM, pp. 140–149.

[53] GARY, M. R., AND JOHNSON, D. S.
Computers and Intractability: A Guide to the Theory of NP-completeness.
WH Freeman and Company, New York, 1979.

[54] GIERING, R., AND KAMINSKI, T.
Recipes for adjoint code construction.
ACM Trans. Math. Softw. 24, 4 (Dec. 1998), 437–474.

[55] GIERING, R., AND KAMINSKI, T.
Applying TAF to generate efficient derivative code of Fortran 77-95 programs.
PAMM 2, 1 (2003), 54–57.

[56] GIERING, R., KAMINSKI, T., TODLING, R., ERRICO, R., GELARO, R., AND WINSLOW, N.
Tangent linear and adjoint versions of NASA/GMAO's Fortran 90 global weather forecast model.
In *Automatic Differentiation: Applications, Theory, and Implementations*, C. H. Bischof, H. M. Bücker, and P. Hovland, Eds. Springer, 2006, pp. 275–284.

[57] GILES, M., DUTA, M., AND MUELLER, J.-D.
Adjoint code developments using the exact discrete approach.
In *15th AIAA Computational Fluid Dynamics Conference* (2001), American Institute of Aeronautics and Astronautics.

[58] GILES, M. B.
On the use of Runge-Kutta time-marching and multigrid for the solution of steady adjoint equations.
Tech. Rep. NA00/10, Oxford University Computing Laboratory, 2000.

[59] GILES, M. B.
Collected matrix derivative results for forward and reverse mode algorithmic differentiation.
In *Advances in Automatic Differentiation* (Berlin, Heidelberg, 2008), C. H. Bischof, H. M. Bücker, P. Hovland, U. Naumann, and J. Utke, Eds., Springer Berlin Heidelberg, pp. 35–44.

[60] GILES, M. B., MUDALIGE, G. R., SHARIF, Z., MARKALL, G., AND KELLY, P. H.
Performance analysis of the OP2 framework on many-core architectures.
SIGMETRICS Perform. Eval. Rev. 38, 4 (Mar. 2011), 9–15.

[61] GILES, M. B., AND PIERCE, N. A.
An introduction to the adjoint approach to design.

Flow, Turbulence and Combustion 65, 3 (2000), 393–415.

[62] GOLDBERG, D.
What every computer scientist should know about floating-point arithmetic.
ACM Computing Surveys (CSUR) 23, 1 (1991), 5–48.

[63] GORMAN, G. J., SOUTHERN, J., FARRELL, P. E., PIGGOTT, M., ROKOS, G., AND KELLY, P. H.
Hybrid OpenMP/MPI anisotropic mesh smoothing.
Procedia Computer Science 9 (2012), 1513–1522.

[64] GREENBAUM, A., AND STRAKOS, Z.
Predicting the behavior of finite precision lanczos and conjugate gradient computations.
SIAM Journal on Matrix Analysis and Applications 13, 1 (1992), 121–137.

[65] GRIEWANK, A.
On automatic differentiation.
Mathematical Programming: recent developments and applications 6, 6 (1989), 83–107.

[66] GRIEWANK, A., JUEDES, D., AND UTKE, J.
Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++.
ACM Trans. Math. Softw. 22, 2 (June 1996), 131–167.

[67] GRIEWANK, A., AND WALTHER, A.
Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation.
ACM Trans. Math. Softw. 26, 1 (Mar. 2000), 19–45.

[68] GRIEWANK, A., AND WALTHER, A.
Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, second ed.
Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008.

[69] GUNZBURGER, M.
Adjoint equation-based methods for control problems in incompressible, viscous flows.
In *Flow, Turbulence and Combustion* (2000), vol. 65, Kluwer Academic Publishers, pp. 249–272.

[70] GUO, X., LANGE, M., GORMAN, G., MITCHELL, L., AND WEILAND, M.
Developing a scalable hybrid MPI/OpenMP unstructured finite element model.
Computers & Fluids 110 (2015), 227–234.

[71] GUTIÉRREZ, E., ASENJO, R., PLATA, O., AND ZAPATA, E. L.
Automatic parallelization of irregular applications.
Parallel Computing 26, 13–14 (2000), 1709–1738.

[72] HASCOËT, L., FIDANOVA, S., AND HELD, C.
Adjoining independent computations.
In *Automatic Differentiation of Algorithms, from Simulation to Optimization* (2002), Springer New York, pp. 299–304.

[73] HASCOËT, L., NAUMANN, U., AND PASCUAL, V.
"To Be Recorded" analysis in reverse-mode automatic differentiation.
Future Generation Computer Systems 21, 8 (2004).

[74] HASCOET, L., AND PASCUAL, V.
The Tapenade automatic differentiation tool: Principles, model, and specification.
ACM Trans. Math. Softw. 39, 3 (May 2013), 20:1–20:43.

[75] HENRETTY, T., VERAS, R., FRANCHETTI, F., POUCHET, L.-N., RAMANUJAM, J., AND SADAYAPPAN, P.
A stencil compiler for short-vector SIMD architectures.
In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing* (New York, NY, USA, 2013), ICS '13, ACM, pp. 13–24.

[76] HEUVELINE, V., AND WALTHER, A.
Online checkpointing for parallel adjoint computation in PDEs: Application to goal-oriented adaptivity and flow control.
In *Euro-Par 2006 Parallel Processing*. Springer, 2006, pp. 689–699.

[77] HIGHAM, N. J.
The accuracy of floating point summation.
SIAM Journal on Scientific Computing 14, 4 (1993), 783–799.

[78] HILBERT, D.
Ein Beitrag zur Theorie des Legendre'schen Polynoms.
Acta Math. 18 (1894), 155–159.

[79] HOGAN, R. J.
Fast reverse-mode automatic differentiation using expression templates in C++.
ACM Trans. Math. Softw. 40, 4 (2014), 26.

[80] HOLEWINSKI, J., POUCHET, L.-N., AND SADAYAPPAN, P.
High-performance code generation for stencil computations on GPU architectures.
In *Proceedings of the 26th ACM international conference on Supercomputing* (2012), ACM, pp. 311–320.

[81] HOVLAND, P. D.
Automatic differentiation of parallel programs.
PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[82] HÜCKELHEIM, J., HOVLAND, P. D., STROUT, M. M., AND MÜLLER, J.-D.
Reverse-mode algorithmic differentiation of an OpenMP-parallel compressible flow solver.
International Journal for High Performance Computing Applications (in review).

[83] HÜCKELHEIM, J., AND MÜLLER, J.-D.
Checkpointing with time gaps for unsteady adjoint CFD.
In *EUROGEN - International Conference on Evolutionary and Deterministic Methods for Design, Optimization and Control with Applications to Industrial and Societal Problems* (September 2015), vol. 11.

[84] HÜCKELHEIM, J., XU, S., GUGALA, M., AND MÜLLER, J.-D.
Time-averaged steady vs. unsteady adjoint: a comparison for cases with mild unsteadiness.
In *53rd AIAA Aerospace Sciences Meeting* (2015/05/03 2015), American Institute of Aeronautics and Astronautics.

[85] HÜCKELHEIM, J. C., HASCOËT, L., AND MÜLLER, J.-D.
Algorithmic differentiation of code with multiple context-specific activities.
ACM Trans. Math. Softw. 43, 4 (Jan. 2017), 35:1–35:21.

[86] ISTVÁN Z . REGULY, ENDRE LÁSZLÓ, G. R. M., AND GILES, M. B.
Vectorizing unstructured mesh computations for many-core architectures.
In *Proceedings of Programming Models and Applications on Multicores and Manycores* (New York, NY, USA, 2007), PMAM'14, ACM, pp. 39:39–39:50.

[87] JAMESON, A.
Aerodynamic shape optimization using the adjoint method.
In *Lectures at the Von Karman Institute, Brussels* (2003).

[88] JAMESON, A., AND OU, K.
Encyclopedia of Aerospace Engineering.

John Wiley & Sons, Ltd., 2010, ch. Optimization Methods in Computational Fluid Dynamics.

[89] JAMESON, A., SHANKARAN, S., AND MARTINELLI, L.
Continuous adjoint method for unstructured grids.
AIAA Journal 46, 5 (2015/05/03 2008), 1226–1239.

[90] KÄBE, C., MARUHN, J. H., AND SACHS, E. W.
Adjoint-based Monte Carlo calibration of financial market models.
Finance and Stochastics 13, 3 (2009), 351–379.

[91] KAHN, G.
Natural semantics.
Springer, 1987.

[92] KIRN, S., ALONSO, J. J., AND JAMESON, A.
Design optimization of high-lift configurations using a viscous continuous adjoint method.
AIAA2002-0844 (2002).

[93] KNOLL, D. A., AND KEYES, D. E.
Jacobian-free Newton–Krylov methods: a survey of approaches and applications.
Journal of Computational Physics 193, 2 (2004), 357–397.

[94] KRAKOS, J. A.
Unsteady adjoint analysis for output sensitivity and mesh adaptation.
PhD thesis, Massachusetts Institute of Technology, 2012.

[95] KRAKOS, J. A., AND DARMOFAL, D. L.
Effect of small-scale output unsteadiness on adjoint-based sensitivity.
AIAA Journal 48, 11 (2015/05/03 2010), 2611–2623.

[96] KREASECK, B., RAMOS, L., EASTERDAY, S., STROUT, M., AND HOVLAND, P.
Hybrid static/dynamic activity analysis.
In *Computational Science–ICCS 2006*. Springer, 2006, pp. 582–590.

[97] LAWSON, C. L., HANSON, R. J., KINCAID, D. R., AND KROGH, F. T.
Basic linear algebra subprograms for Fortran usage.
ACM Trans. Math. Softw. 5, 3 (1979), 308–323.

[98] LEE, B. J., AND LIOU, M.-S.
Unsteady adjoint approach for design optimization of flapping airfoils.
AIAA Journal 50, 11 (2015/05/03 2012), 2460–2475.

[99] LEWIS, F. L., AND SYRMOS, V. L.
Optimal control.
John Wiley & Sons, Ltd., 1995.

[100] LIAO, C., HERNANDEZ, O., CHAPMAN, B., CHEN, W., AND ZHENG, W.
OpenUH: An optimizing, portable OpenMP compiler.
Concurrency and Computation: Practice and Experience 19, 18 (2007), 2317–2332.

[101] LIONS, J. L.
Optimal control of systems governed by partial differential equations, vol. 170.
Springer Verlag, 1971.

[102] LÖHNER, R., AND BAUM, J. D.
Handling tens of thousands of cores with industrial/legacy codes: Approaches, implementation and timings.
Computers & Fluids 85, 0 (10 2013), 53–62.

[103] LOTFI, V., AND SARIN, S.
A graph coloring algorithm for large scale scheduling problems.
Computers Operations Research 13, 1 (1986), 27 – 32.

[104] LOTZ, J., LEPPKES, K., AND NAUMANN, U.
dco/c++-derivative code by overloading in C++.
Tech. Rep. AIB-2011-06, RWTH Aachen University, 2011.

[105] LUK, C.-K., HONG, S., AND KIM, H.
Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping.
In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (2009), ACM, pp. 45–55.

[106] LYONS, A., SAFRO, I., AND UTKE, J.
Randomized heuristics for exploiting jacobian scarcity.
Optimization Methods and Software 27, 2 (2012), 311–322.

[107] LYONS, A., AND UTKE, J.
On the practical exploitation of scarsity.
In *Advances in Automatic Differentiation*, C. H. Bischof, H. M. Bücker, P. D. Hovland, U. Naumann, and J. Utke, Eds. Springer, 2008, pp. 103–114.

[108] MARTINS, J. R., ALONSO, J. J., AND REUTHER, J. J.

A coupled-adjoint sensitivity analysis method for high-fidelity aero-structural design.
Optimization and Engineering 6, 1 (2005), 33–62.

[109] MARTINS, J. R., STURDZA, P., AND ALONSO, J. J.
The complex-step derivative approximation.
ACM Trans. Math. Softw. 29, 3 (2003), 245–262.

[110] MAVRIPLIS, D. J.
Discrete adjoint-based approach for optimization problems on three-dimensional unstructured meshes.
AIAA journal 45, 4 (2007), 741–750.

[111] MITRA, G., JOHNSTON, B., RENDELL, A. P., MCCREATH, E., AND ZHOU, J.
Use of SIMD vector operations to accelerate application code performance on low-powered arm and intel platforms.
In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International* (2013), IEEE, pp. 1107–1116.

[112] MORÉ, J. J., AND WILD, S. M.
Do you trust derivatives or differences?
Journal of Computational Physics 273 (2014), 268–277.

[113] MÜLLER, J.-D., AND CUSDIN, P.
On the performance of discrete adjoint CFD codes using automatic differentiation.
International Journal for Numerical Methods in Fluids 47, 8-9 (2005), 939–945.

[114] NADARAJAH, S., AND JAMESON, A.
A comparison of the continuous and discrete adjoint approach to automatic aerodynamic optimization.
In *38th Aerospace Sciences Meeting and Exhibit* (2017/02/20 2000), American Institute of Aeronautics and Astronautics.

[115] NADARAJAH, S., AND JAMESON, A.
Optimum shape design for unsteady three-dimensional viscous flows using a nonlinear frequency-domain method.
Journal of Aircraft 44, 5 (2015/05/03 2007), 1513–1527.

[116] NAUMANN, U.
Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph.
Math. Program. 99, 3 (Apr. 2004), 399–421.

[117] NAUMANN, U.
The art of differentiating computer programs: an introduction to algorithmic differentiation, vol. 24.
SIAM, 2012.

[118] NAUMANN, U., AND DU TOIT, J.
Adjoint algorithmic differentiation tool support for typical numerical patterns in computational finance.
Tech. Rep. AIB2014-13, RWTH Aachen University, Aachen, Germany, 2014.

[119] NAUMANN, U., HASCOËT, L., HILL, C., HOVLAND, P., RIEHME, J., AND UTKE, J.
A framework for proving correctness of adjoint message-passing programs.
In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 316–321.

[120] NAUMANN, U., UTKE, J., HEIMBACH, P., HILL, C., OZYURT, D., WUNSCH, C., FAGAN, M., TALLENT, N., AND STROUT, M.
Adjoint code by source transformation with OpenAD/F.
In *ECCOMAS CFD 2006: Proceedings of the European Conference on Computational Fluid Dynamics, Egmond aan Zee, The Netherlands, September 5-8, 2006* (2006), Delft University of Technology; European Community on Computational Methods in Applied Sciences (ECCOMAS).

[121] NAUMANN, U., UTKE, J., LYONS, A., AND FAGAN, M.
Control flow reversal for adjoint code generation.
In *Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004)* (Los Alamitos, CA, USA, 2004), IEEE Computer Society, pp. 55–64.

[122] NOLAN, J. F.
Analytical differentiation on a digital computer.
PhD thesis, Massachusetts Institute of Technology, 1953.

[123] NOTAY, Y.
On the convergence rate of the conjugate gradients in presence of rounding errors.
Numerische Mathematik 65, 1 (1993), 301–317.

[124] OGITA, T., RUMP, S. M., AND OISHI, S.
Accurate sum and dot product.
SIAM Journal on Scientific Computing 26, 6 (2005), 1955–1988.

[125] OLHOFF, N.
Optimal design of vibrating rectangular plates.
International Journal of Solids and Structures 10, 1 (1974), 93–109.

[126] OTHMER, C.
A continuous adjoint formulation for the computation of topological and surface sensitivities of ducted flows.
International Journal for Numerical Methods in Fluids 58, 8 (2008), 861–877.

[127] OTHMER, C., DE VILLIERS, E., AND WELLER, H.
Implementation of a continuous adjoint for topology optimization of ducted flows.
In *18th AIAA Computational Fluid Dynamics Conference* (2017/02/20 2007), American Institute of Aeronautics and Astronautics.

[128] PALACIOS, F., ALONSO, J., DURAISAMY, K., COLONNO, M., HICKEN, J., ARANAKE, A., CAMPOS, A., COPELAND, S., ECONOMON, T., LONKAR, A., LUKACZYK, T., AND TAYLOR, T.
Stanford university unstructured (SU2): An open-source integrated computational environment for multi-physics simulation and design.
In *51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition* (2017/02/20 2013), American Institute of Aeronautics and Astronautics.

[129] PASCUAL, V., AND HASCOËT, L.
Advances in Automatic Differentiation.
Springer, 2008, ch. Tapenade for C, pp. 199–209.

[130] PIERCE, N. A., AND GILES, M. B.
Preconditioned multigrid methods for compressible flow calculations on stretched meshes.
Journal of Computational Physics 136, 2 (1997), 425–445.

[131] PLESSIX, R.-E.
A review of the adjoint-state method for computing the gradient of a functional with geophysical applications.
Geophysical Journal International 167, 2 (2006), 495–503.

[132] RATANAWORABHAN, P., KE, J., AND BURTSCHER, M.
Fast lossless compression of scientific floating-point data.
In *Proceedings of the Data Compression Conference* (Washington, DC, USA, 2006), DCC '06, IEEE Computer Society, pp. 133–142.

[133] RAVISHANKAR, M., EISENLOHR, J., POUCHET, L.-N., RAMANUJAM, J., ROUNTEV, A., AND SADAYAPPAN, P.
Code generation for parallel execution of a class of irregular loops on distributed memory systems.
In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)* (November 2012).

[134] REVELS, J., LUBIN, M., AND PAPAMARKOU, T.
Forward-mode automatic differentiation in Julia.
arXiv:1607.07892 [cs.MS] (2016).

[135] ROSS, P. E.
Why CPU frequency stalled.
IEEE Spectrum 45, 4 (2008), 72–72.

[136] RUMPFKEIL, M., AND ZINGG, D.
A general framework for the optimal control of unsteady flows with applications.
In *45th AIAA Aerospace Sciences Meeting and Exhibit* (2015/05/03 2007), American Institute of Aeronautics and Astronautics.

[137] SAAD, Y., AND VAN DER VORST, H. A.
Iterative solution of linear systems in the 20th century.
Journal of Computational and Applied Mathematics 123, 1 (2000), 1–33.

[138] SCHANEN, M., FÖRSTER, M., LOTZ, J., LEPPKES, K., AND NAUMANN, U.
Adjoining hybrid parallel code.
In *Proceedings of the Eighth International Conference on Engineering Computational Technology* (2012), B. H. V. Topping, Ed., vol. 100, Kippen, Stirlingshire, p. 18.

[139] SCHANEN, M., NAUMANN, U., HASCOËT, L., AND UTKE, J.
Interpretative adjoints for numerical simulation codes using MPI.
Procedia Computer Science 1, 1 (2010), 1825–1833.

[140] SCHMIDL, D., CRAMER, T., WIENKE, S., TERBOVEN, C., AND MÜLLER, M. S.
Assessing the performance of OpenMP programs on the Intel Xeon Phi.
In *Euro-Par 2013 Parallel Processing*. Springer, 2013, pp. 547–558.

[141] SCHNEIDER, T. M., ECKHARDT, B., AND YORKE, J. A.
Turbulence transition and the edge of chaos in pipe flow.
Phys. Rev. Lett. 99 (Jul 2007), 034502.

[142] SHIN, J., AND HOVLAND, P. D.
Comparison of two activity analyses for automatic differentiation: Context-sensitive flow-insensitive vs. context-insensitive flow-sensitive.
In *Proceedings of the 2007 ACM Symposium on Applied Computing* (New York, NY, USA, 2007), SAC '07, ACM, pp. 1323–1329.

[143] SHIN, J., MALUSARE, P., AND HOVLAND, P. D.
Design and implementation of a context-sensitive, flow-sensitive activity analysis algorithm for automatic differentiation.
In *5th International Conference on Automatic Differentiation (AD 2008)* (Bonn, Germany, 2007), vol. 64, Lect. Notes Comput. Sci. Eng., Lect. Notes Comput. Sci. Eng., pp. 115–125.

[144] SHU, C.-W.
High-order finite difference and finite volume WENO schemes and discontinuous Galerkin methods for CFD.
International Journal of Computational Fluid Dynamics 17, 2 (2003), 107–118.

[145] SIEGEL, S. F., DWYER, M. B., GOPALAKRISHNAN, G., LUO, Z., RAKAMARIC, Z., THAKUR, R., ZHENG, M., AND ZIRKEL, T. K.
CIVL: The concurrency intermediate verification language.
Tech. Rep. UD-CIS-2014/001, Department of Computer and Information Sciences, University of Delaware, 2014.

[146] SIGMUND, O.
Topology optimization in multiphysics problems.
In *Proc of 7th AIAA/USAF/NASA/ISSMO Symp on Multidisciplinary Analysis and Optimization* (1998), vol. 1, pp. 1492–1500.

[147] SLUSANSCHI, E.
Algorithmic differentiation of Java programs.
PhD thesis, Department of Computer Science, RWTH Aachen University, 2008.

[148] SPALART, P., AND ALLMARAS, S.
A one-equation turbulence model for aerodynamic flows.
In *30th Aerospace Sciences Meeting and Exhibit* (2016/05/04 1992), American Institute of Aeronautics and Astronautics.

[149] SPALART, P. R.
Detached-Eddy simulation.
Annual Review of Fluid Mechanics 41 (2009), 181–202.

[150] SPEELPENNING, B.
Compiling fast partial derivatives of functions given by algorithms.
PhD thesis, Department of Computer Science, Illinois University, Urbana (USA), 1980.

[151] STRAKOŠ, Z., AND TICHÝ, P.
On error estimation in the conjugate gradient method and why it works in finite precision computations.
Electron. Trans. Numer. Anal 13, 56-80 (2002), 8.

[152] TANG, Y., CHOWDHURY, R. A., KUSZMAUL, B. C., LUK, C.-K., AND LEISERSON, C. E.
The pochoir stencil compiler.
In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2011), SPAA '11, ACM, pp. 117–128.

[153] THOMPSON, J. F., SONI, B. K., AND WEATHERILL, N. P.
Handbook of grid generation.
CRC press, 1998.

[154] TIM WILDEY, E. C. C., AND SHADID, J.
Adjoint based a posteriori error estimates using data compression.
In *VI International Conference on Adaptive Modeling and Simulation* (2013), C. T. J. P. Moitinho de Almeida, P. Díez and N. Parés, Eds.

[155] TONG, C., AND YE, Q.
Analysis of the finite precision bi-conjugate gradient algorithm for nonsymmetric linear systems.
Mathematics of Computation of the American Mathematical Society 69, 232 (2000), 1559–1575.

[156] TOWARA, M., AND NAUMANN, U.
A discrete adjoint model for OpenFOAM.
Procedia Computer Science 18 (2013), 429–438.

[157] TOWARA, M., SCHANEN, M., AND NAUMANN, U.
MPI-parallel discrete adjoint OpenFOAM.
Procedia Computer Science 51 (2015), 19–28.

[158] TRUONG, L., MARKLEY, C., AND FOX, A.
An extensible framework for composing stencils.
In *Workshop on Stencil Computations 2014* (2014).

[159] UTKE, J., HASCOËT, L., HEIMBACH, P., HILL, C., HOVLAND, P., AND NAUMANN, U.
Toward adjoinable MPI.
In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* (2009), IEEE, pp. 1–8.

[160] UTKE, J., LYONS, A., AND NAUMANN, U.
Efficient reversal of the interprocedural flow of control in adjoint computations.
Journal of Systems and Software 79 (2006), 1280–1294.

[161] UTKE, J., AND NAUMANN, U.
Combinatorial problems in OpenAD.
In *Combinatorial Scientific Computing*, U. Naumann and O. Schenk, Eds., Computational Science. Chapman and Hall/CRC, January 2012.
chap. 6.

[162] UTKE, J., NAUMANN, U., FAGAN, M., TALLENT, N., STROUT, M., HEIMBACH, P., HILL, C., AND WUNSCH, C.
OpenAD/F: A modular open-source tool for automatic differentiation of Fortran codes.
ACM Trans. Math. Softw. 34, 4 (July 2008), 18:1–18:36.

[163] VAN DOORMAAL, J., AND RAITHBY, G.
Enhancements of the SIMPLE method for predicting incompressible fluid flows.
Numerical heat transfer 7, 2 (1984), 147–163.

[164] VERSTRAETE, T.
The VKI U-bend optimization test case.
Tech. rep., VKI, 2015.

[165] VERSTRAETE, T., COLETTI, F., BULLE, J., VANDERWIELEN, T., AND ARTS, T.
Optimization of a U-Bend for minimal pressure loss in internal cooling channels, part I: Numerical method.
Journal of Turbomachinery 135, 5 (2013), 051015.

[166] WALTER, S. F., AND LEHMANN, L.
Algorithmic differentiation in python with algopy.
Journal of Computational Science 4, 5 (2013), 334–344.

[167] WANG, E., ZHANG, Q., SHEN, B., ZHANG, G., LU, X., WU, Q., AND WANG, Y.
Intel math kernel library.

In *High-Performance Computing on the Intel® Xeon Phi*. Springer, 2014, pp. 167–188.

[168] WANG, Q., HU, R., AND BLONIGAN, P.
Least squares shadowing sensitivity analysis of chaotic limit cycle oscillations.
Journal of Computational Physics 267, 0 (2014), 210 – 224.

[169] WANG, Q., MOIN, P., AND IACCARINO, G.
Minimal repetition dynamic checkpointing algorithm for unsteady adjoint calculation.
SIAM Journal on Scientific Computing 31, 4 (2015/05/03 2009), 2549–2567.

[170] WARNER, D. D.
A partial derivative generator.
Computing Science Technical Report No. 28, Bell Telephone Laboratories, Murray Hill, N.J., 1975.

[171] WEISER, M., AND GÖTSCHEL, S.
State trajectory compression for optimal control with parabolic PDEs.
SIAM Journal on Scientific Computing 34, 1 (2012), A161–A184.

[172] WEISS, R.
A theoretical overview of Krylov subspace methods.
Applied numerical mathematics 19, 3 (1995), 207–233.

[173] WENGERT, R. E.
A simple automatic derivative evaluation program.
Communications of the ACM 7, 8 (1964), 463–464.

[174] WILKES, M. V.
The memory gap and the future of high performance memories.
ACM SIGARCH Computer Architecture News 29, 1 (2001), 2–7.

[175] XU, S., JAHN, W., AND MÜLLER, J.-D.
CAD-based shape optimisation with CFD using a discrete adjoint.
International Journal for Numerical Methods in Fluids 74, 3 (2014), 153–168.

[176] XU, S., RADFORD, D., MEYER, M., AND MÜLLER, J.-D.
Stabilisation of discrete steady adjoint solvers.
Journal of Computational Physics 299 (2015), 175–195.

[177] ZHOU, B. Y., ALBRING, T. A., GAUGER, N. R., ECONOMON, T. D., PALACIOS, F., AND ALONSO, J. J.

A discrete adjoint framework for unsteady aerodynamic and aeroacoustic optimization.

In *16th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference* (2017/02/20 2015), American Institute of Aeronautics and Astronautics.