TECHNISCHE UNIVERSITÄT DRESDEN

FAKULTÄT INFORMATIK

Advanced Concepts for Automatic Differentiation based on Operator Overloading

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

eingereicht von

Dipl.-Inf. Andreas Kowarz
geboren am 26. Mai 1974 in Löbau

Betreuender Hochschullehrer:	Prof. Dr. rer. nat. Wolfgang E. Nagel
Gutachter:	Prof. Dr. rer. nat. Wolfgang E. Nagel, TU Dresden Univ.-Prof. Dr. rer. nat. Uwe Naumann, RWTH Aachen Prof. Ph. D. Andreas Griewank, HU Berlin
Eingereicht am:	16.11.2007
Tag der Verteidigung:	20.03.2008

To Daiwan,

who taught me the meaning of freedom.

Contents

1 Introduction

> *Derivatives play a central role in sensitivity analysis (model validation), inverse problems (data assimilation), and design optimization (simulation parameter choice). At a more elementary level they are used for solving algebraic and differential equations, curve fitting, and many other applications.*
>
> Andreas Griewank, "Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation", Number 19 in Frontiers in Applied Mathematics, SIAM, Philadelphia, 2000.

As summarized by the quote above, derivative information is used in many applications from science and technology. In fact, a considerable part of the improvements achieved in many areas in the last decades was only possible due to the availability of derivatives. Providing derivatives for a given function sometimes turns out to be challenging. By applying the technique of symbolic differentiation, explicit derivative formulas can be obtained either by hand or by use of specialized software. However, this approach is only applicable if the formula of the function is known and the differentiation is "handleable". Whereas an accurate definition of the term "handleable" is not easily given, its practical meaning is obvious: The effort of providing an explicit derivative formula of a function must not exceed the available recourses. This is fulfilled only for a limited set of functions that feature a quite simple structure.

Automatic differentiation is a technique that has been developed and improved in the last decades. It allows to compute numerical derivative values within machine accuracy for a given function of basically unlimited complexity. Thus, unlike finite differences, no truncation errors must be taken into account. When calculating first order directional derivatives, i.e., Jacobian-vector products, the computational effort is comparable to that of finite differences. This way, e.g., columns of the Jacobian can be computed efficiently. However, if a vector-Jacobian product is to be computed, e.g., a specific row of the Jacobian, the computational effort is proportional to the number of entries in the row when applying finite differences. The same task can be performed much more efficiently by use of automatic differentiation. In particular, the computational effort is then independent of the rows dimension. As many applications mentioned in the above quote rely on this type of information, automatic differentiation more and more attains significant importance for their numerical handling.

Taking advantage of the mathematical properties of automatic differentiation requires the application of a suitable tool. Many different tools have been implemented so far. This way, automatic differentiation has been made available to several programming languages and mathematical software packages. Independent of the provided functionality, the underlying approach belongs to one of the two implementation strategies. Source-to-source transformation is performed by specialized compilers. This results in the generation of source code for the derivative function. Using standard compiler technology, function and derivative code are object to the binary creation and benefit from the applied compiler optimization. The second approach of providing automatic differentiation is given by the use of the operator overloading capabilities of many modern programming languages. By appropriate overloading of operators and intrinsic functions, an internal function representation can be generated that can later be reevaluated in the derivation process. Whereas the overloading approach allows to apply automatic differentiation while exploiting the full language standard, it suffers from the interpretative overhead belonging to the internal function representation.

In practice, the choice of the applied approach is determined by a fact of more pragmatic nature: Source-to-source transformation is mainly available for the FORTRAN programming language. This is due to

the comparable expensive development and maintenance of the necessary tools. So far, the restricted language features of the FORTRAN 77 dialect enable the most stable application of the source-to-source approach. Additional facilities, e.g., dynamic memory management, provided by newer dialects and other programming languages, e.g., C/C++, present a major challenge to this approach. As a result, appropriate source-to-source tools are either not available or miss critical capabilities for many programming languages of interest. Nevertheless, applications can benefit from automatic differentiation in such environments due to the existence of the operator overloading approach. Provided that the considered programming language facilitates the overloading concept, the implementation of automatic differentiation based on this feature turned out to be comparable straightforward.

Independent of the applied approach, automatic differentiation must face an important fact: Theoretical advantage on its own does not convince users to apply the technique. Rather, it must be accompanied by considerable benefits observable in practice. Over the last years, significant effort has been invested in improving source-to-source tools. In contrast, no substantial advance has been made in the development of automatic differentiation based on operator overloading in this period. Considering its meaning for the differentiation of functions written in modern programming languages, this situation seems inappropriate.

With this thesis, new techniques are presented that considerable improve the application of operator overloading based automatic differentiation and allow higher serial performance. In addition, facilities are discussed that enable the parallel derivation of parallel user functions. The presentation of the new approaches is divided into three main chapters. In the following chapter, a short overview of the two work modes of automatic differentiation and the basic implementation strategies is given. Thereafter, the theoretical aspects of the newly developed techniques are discussed in detail. This chapter is again divided into three large sections. The first section focuses on the exploitation of the function structure for reducing the size of the internal function representation. Subsequently, an activity-tracking technique is introduced that not only allows a more compact internal representation of the function but also simplifies the application of automatic differentiation itself. The chapter is completed with the introduction of parallelization techniques and the description of the required modifications to the overloading based approach. Chapter four presents information on the validation of the developed concepts. Therefore, the initial state of the applied tool *ADOL-C* is discussed briefly and the required modifications in more detail. The main part of this chapter is then dedicated to three examples that benefit from the new techniques. Each application is described briefly and the relevant aspects of the differentiation are emphasized. Analyzing the runtime behavior for all three examples clarifies the potential of the improved approaches. A short summary of the thesis is given in the last chapter that also contains a schedule of further research activities that, to the opinion of the author, must be addressed in the future.

2 Automatic Differentiation

Many modern techniques applied in industrial processes are directly based or are the result of challenging mathematical simulations and optimizations that extensively use derivative information. Thereby, the effort for setting up general formulas for the derivative computation is tightly coupled to the complexity of the underlying functions. Deriving these general formulas by hand is often very time consuming and error-prone.

With the appearance of powerful computer hardware several solution methods to this problem were developed. All computations are based on a vector-valued function

$$F : I\!R^n \to I\!R^m, \qquad y = F(x), \tag{2.1}$$

that is given as source code in a certain programming language. In practice, many numerical algorithms are based on the knowledge of derivative values in a certain direction only, instead on the computation of the complete Jacobian F'. The directional derivative $\dot{y}$ is defined by

$$\dot{y} := F'(x)\dot{x} = \lim_{h \to 0} \frac{F(x + h * \dot{x}) - F(x)}{h}. \tag{2.2}$$

A common technique of approximating this value is known as the principle of finite differences (FD) that can be described by

$$\dot{y} = F'(x)\dot{x} \approx \frac{F(x + h * \dot{x}) - F(x)}{h} =: \tilde{y}, \tag{2.3}$$

using a small $h \in I\!R$, with

$$\dot{x} \in I\!R^n,\ \dot{y} \in I\!R^m,\ F'(x) \in I\!R^{m \times n}.$$

The computational cost ω_{FD} for computing the derivative value $\tilde{y}$ this way is dominated by the two evaluations of F for the two different arguments. It is determined for all non-trivial functions F by

$$\omega_{FD} = \frac{\text{TIME}(\tilde{y})}{\text{TIME}(y)} = 2. \tag{2.4}$$

Here, TIME(y) is the time for evaluating the function value and TIME($\tilde{y}$) is the time required to compute function and approximate derivative values. Due to the fact that the function F is evaluated at two different base points an advantageous memory behavior can be achieved. With exception of the input and output variables that have to be saved all internal storage can be reused. This way, the task of evaluating the function and the derivative calculations are comparable in terms of the memory requirement.

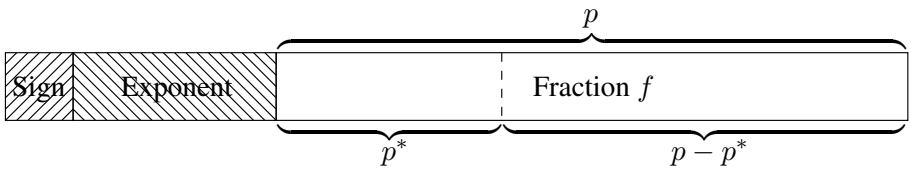


Figure 2.1: IEEE floating-point number

Derivative computations on modern computer hardware are mostly done with limited accuracy based on the IEEE-floating-point arithmetic [Ins85]. Based on this standard, the difference between $F(x + h * \dot{x})$ and $F(x)$ is determined for small values of h by the $p - p^*$ least significant bit positions of fraction f of the two corresponding floating-point numbers, see Fig. 2.1. There, p represents the number of bit

positions in the fraction f and p^* with $0 \leq p^* \leq p$ is the length of the bit sequence identical in the most significant bit positions of f. With increasing p^* due to changes in h and under assumption of the same exponents and signs the difference $F(x + h * \dot{x}) - F(x)$ is determined by a decreasing number of bit positions $p - p^*$ that is often not sufficient to represent the exact difference compared to calculations in exact arithmetic. This truncation error inhibits better approximations of the derivatives even for situation where carefully chosen smaller values of h could be encoded exactly.

Considering the computational effort, the technique of finite differences provides an efficient way to approximate a column of the Jacobian F'. However, gradient information is needed frequently, e.g., for parameter optimization. Each function according to (2.1) can be decomposed into its component functions $F_1(x_1, x2, \ldots, x_n)$ through $F_m(x_1, x2, \ldots, x_n)$ with $F_i : \mathbb{R}^n \rightarrow \mathbb{R}$, yielding

$$y = (y_1, y_2, \ldots, y_m) = (F_1(x_1, x_2, \ldots, x_n), F_2(x_1, x_2, \ldots, x_n), \ldots, F_m(x_1, x_2, \ldots, x_n)). \quad (2.5)$$

The gradient $\nabla F_i(.)$ of each component function $F_i(.)$ is contained in the ith row of the Jacobian F'. Due to the finite differences technique, gradient information needs to be constructed using columns of F'. This means, the computation of full gradients is possible only by approximating the n columns of F', i.e., the complete Jacobian. The effort for this task is dependent on the number n of arguments x.

In the following section a different technique called Automatic Differentiation is described that can be used to compute derivative values without the previously described problems.

2.1 Basics

Unlike finite differences that are used to approximate derivative values, Automatic Differentiation (AD) was developed to compute these values directly and within machine accuracy. A complete description of mostly all basic techniques known in the field of AD can be found in Andreas Griewank's book *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation* [Gri00]. Most of the mathematical symbolism used within this thesis is based on this book.

As for finite differences, a function according to (2.1) given as source code in a certain programming language is the object of the derivation procedure. The computation of derivative values for this function is based on the chain rule, i.e. for two functions $y = f(u)$ and $u = g(x)$ the derivative $y' = (f(g(x)))'$ can be determined as

$$\frac{dy}{dx} = f'(u)g'(x) = \frac{dy}{du}(g(x))\frac{du}{dx}(x). \quad (2.6)$$

To apply the chain rule to the function under consideration, the function needs to be decomposed into elemental functions $\varphi_i \in \Phi$. The corresponding sequence of elemental functions is called evaluation procedure. This is satisfied by the implementation as computer program, in general. The size of Φ may vary over the programming languages but contains two kinds of operations and functions, respectively. Prerequisite for both types is a straight-forward differentiation, i.e., the corresponding mathematical formula must be known. The difference is determined in the number of arguments. Throughout this thesis unary and binary operations/functions, respectively, are considered only. Typical representatives of binary operations are addition, subtraction, multiplication and division. Most intrinsic functions as sine, cosine, tangent, square root etc. and increment, decrement are examples of unary functions and operations, respectively.

Throughout, it is assumed that each $\varphi_i \in \Phi$ is d-times continuously differentiable on its open domain $\mathcal{D}_i$ and the evaluation procedure is well defined at all points x of its domain $\mathcal{D}$. For the result of each elemental function φ_i one defines an intermediate variable v_i with $i = 1, \ldots, l$. Furthermore, it is assumed in this section that no overwriting occurs, such that a variable v_i is the result of an elemental function exactly once. For information on the handling of overwritten intermediate variables see [Gri00].

$$
\begin{aligned}
v_{-2} &= x_1 \\
v_{-1} &= x_2 \\
v_0 &= x_3 \\
\hline
v_1 &= v_{-1}/v_{-2} \\
v_2 &= v_{-2}^2 \\
v_3 &= v_{-1}^2 \\
v_4 &= v_0^2 \\
v_5 &= v_2 + v_3 \\
v_6 &= v_5 + v_4 \\
v_7 &= \sqrt{v_5} \\
v_8 &= v_7/v_0 \\
v_9 &= \sqrt{v_6} \\
v_{10} &= \arctan(v_8) \\
v_{11} &= \arctan(v_1) \\
\hline
y_1 &= v_9 \\
y_2 &= v_{10} \\
y_3 &= v_{11}
\end{aligned}
$$

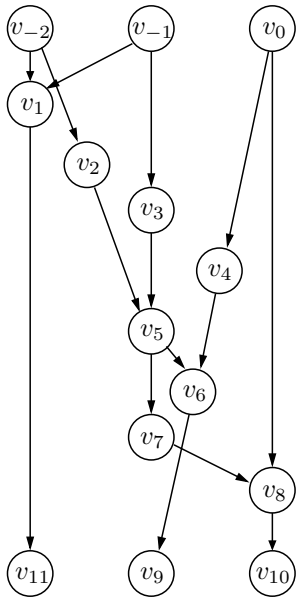


Figure 2.2: Coordinate transformation – evaluation procedure and computational graph

In addition, the input variables x_i with $i = 1 \ldots, n$, called independents in terms of AD, are assigned to the intermediate variables v_{i-n}. It is further assumed that the last m intermediate variables receive the function values that are copied to the output variables y_i, $i = 1, \ldots, m$, called dependent variables in terms of AD.

An example of an evaluation procedure is depicted in the left half of Fig. 2.2. Its origin is the well-known transformation from Cartesian to spherical coordinates, given by

$$y_1 = \sqrt{x_1^2 + x_2^2 + x_3^2}, \qquad y_2 = \arctan \frac{\sqrt{x_1^2 + x_2^2}}{x_3}, \qquad y_3 = \arctan \frac{x_2}{x_1}.$$

In Fig. 2.2, the operations between the two dashed lines implement the actual performed arithmetic. At the beginning and the end additional operations are used to copy the independent variables and to set the dependent variables, as described before. The complete arithmetic is based on the internal representation that way.

Using the variables v_i, $i = 1 - n, \ldots, l$ and the decomposed function, one can construct a computational graph as depicted in the right half of Fig. 2.2. In this graph, each variable v_i is represented by a vertex. In addition, each elemental function φ_i with $i = 1, \ldots, l$ is characterized by its result v_i and its arguments v_j with $j < i$, where a directed edge connects each argument v_j with the result v_i [Bau74]. Thus, the number of incoming edges for a vertex v_i is equivalent to the number of arguments of the corresponding elemental function φ_i. To describe this setting mathematically, the dependence relation $\prec$ is used. For two indices i and j, the term $j \prec i$ means that v_i directly depends on v_j. Using the dependence relation, an elemental function φ_i is defined by

$$v_i = \varphi_i(v_j)_{j \prec i}. \tag{2.7}$$

This way, the evaluation procedure for a decomposed function can be generalized as shown in Table 2.1 [Gri00, pp. 18].

2.1.1 Forward mode

The core idea behind the forward mode is to compute derivatives for each elemental function φ_i and combine them according to (2.6). Therefore, a path $x(t)$ in the domain $\mathbb{R}^n$ is considered that is mapped

Table 2.1: General evaluation procedure

$v_{i-n} = x_i$	$i = 1, \ldots, n$
$v_i = \varphi(v_j)_{j \prec i}$	$i = 1, \ldots, l$
$v_{m-i} = v_{l-i}$	$i = m-1, \ldots, 0$

by the function F onto a path in $\mathbb{R}^m$:

$$y(t) = F(x(t)).$$

For an arbitrary but fixed t_0, calculating the directional derivative of F at a given point $x_0 = x(t_0)$ of this path along $x(t)$ yields the formula

$$\dot{y} := \frac{dy}{dt}(t_0) = \frac{dF}{dx}(x_0)\frac{dx}{dt}(t_0).$$

With $F'(x_0) := \frac{dF}{dx}(x_0)$ and $\dot{x} := \frac{dx}{dt}(t_0)$ one has

$$\dot{y} = F'(x_0)\dot{x}.$$

Now, the arguments v_j and the results v_i of the elemental functions φ_i are time dependent and their total derivatives are determined by

$$\dot{v}_i := \frac{dv_i}{dt} = \sum_{j \prec i} \frac{\partial \varphi_i(v_j)_{j \prec i}}{\partial v_j}\frac{dv_j}{dt} = \sum_{j \prec i} \frac{\partial \varphi_i(u_i)}{\partial v_j}\dot{v}_j = \sum_{j \prec i} c_{ij}\dot{v}_j \tag{2.8}$$

with $u_i := (v_j)_{j \prec i} \in \mathbb{R}^{n_i}$ and $c_{ij} := \frac{\partial \varphi_i(u_i)}{\partial v_j}$. Using formula (2.8), derivatives for all elemental functions $\varphi_i \in \Phi$ can be determined easily. Some examples are given in Table 2.2.

Table 2.2: Tangent operations

$v = c$	$\dot{v} = 0$
$v = u \pm w$	$\dot{v} = \dot{u} \pm \dot{w}$
$v = u * w$	$\dot{v} = \dot{u} * w + u * \dot{w}$
$v = 1/u$	$\dot{v} = (-1) * u^{-2} * \dot{u}$
$v = u^c$	$\dot{v} = c * u^{c-1} * \dot{u}$
$v = \sin(u)$	$\dot{v} = \cos(u) * \dot{u}$

Now, an algorithm for computing directional derivatives of the function F can be constructed, similar to Table 2.1. First, the intermediate variables $\dot{v}_{i-n}$ with $i = 1, \ldots, n$ need to be initialized with the component values of $\dot{x}$. Afterwards, the derivatives are computed step by step using (2.8). Finally, the components of $\dot{y}$ are extracted from the intermediate variables $\dot{v}_{l-i}$ with $i = m-1, \ldots, 0$. All together, a general procedure for the forward mode of AD, computing the function value and exact first order derivatives, can be constructed as summarized in Table 2.3. As for finite differences, a bound for the computational cost of the forward mode of AD, as given by Table 2.3, can be derived using the complexity theory of AD [Gri00, pp. 43]. In detail, one obtains

$$\omega_F = \frac{\text{TIME}(\dot{y})}{\text{TIME}(y)} \in [2, 2.5]. \tag{2.9}$$

As for finite differences the memory requirement for the forward mode of AD can be estimated. Assuming that the derivatives are computed together with the function values step by step, the necessary storage is doubled.

Table 2.3: Forward mode of AD – General procedure

$$
\begin{array}{|rcll|}
\hline
v_{i-n} & = & x_i & \left.\begin{array}{l}\\ \\ \end{array}\right\} \; i = 1, \ldots, n \\
\dot{v}_{i-n} & = & \dot{x}_i & \\
\hline
v_i & = & \varphi_i(u_i) & \left.\begin{array}{l}\\ \\ \end{array}\right\} \; i = 1, \ldots, l \\
\dot{v}_i & = & \displaystyle\sum_{j \prec i} c_{ij} * \dot{v}_j & \\
\hline
y_{m-i} & = & v_{l-i} & \left.\begin{array}{l}\\ \\ \end{array}\right\} \; i = m-1, \ldots, 0 \\
\dot{y}_{m-i} & = & \dot{v}_{l-i} & \\
\hline
\end{array}
$$

In summary, the forward mode of AD is comparable to finite differences in terms of the computational effort but in contrast to the latter offers derivatives within system accuracy at the cost of an increased memory requirement.

2.1.2 Reverse mode

Despite the fact that many problems involving derivative values can be solved efficiently using the forward mode of AD, the efficiency of other algorithm, especially for numerical optimization, depends on the computational cost for discrete adjoint information. Considering sensitivity information for the dependent variable y_i of a component Function F_i according to (2.5) with respect to all independents x_j, $j = 1, \ldots, n$, the gradient is defined by

$$\nabla F_i(x) = e_i^T F'(x),$$

where e_i is the ith Cartesian base vector. More general, one can define the vector matrix product

$$\bar{x}^T = \bar{y}^T F'(x) \qquad \text{with} \quad \bar{x} \in I\!R^n, \quad \bar{y} \in I\!R^m. \tag{2.10}$$

However, computing the whole Jacobian $F'(x)$ and then multiplying from the left by $\bar{y}$ is not efficient. To derive the reverse mode, (2.8) can be reinterpreted as mapping from $I\!R^{n+l}$ to $I\!R^{n+l}$. Using the Cartesian basis vectors in $I\!R^{n+l}$, matrices A_i, $i = 1, \ldots, l$, are defined by

$$A_i := I + e_{n+i} \left[\nabla \varphi_i(u_i) - e_{n+i}\right]^T \qquad \in I\!R^{(n+l)\times(n+l)}. \tag{2.11}$$

Given a temporal snapshot $\dot{V}_{i-1} \equiv (\dot{v}_{1-n}, \dot{v}_{2-n}, \ldots, \dot{v}_{l-1}, \dot{v}_l)$ of the intermediate variables prior to the differentiation and a snapshot $\dot{V}_i$ after the differentiation of the elemental function φ_i with $i = 1, \ldots, l$, the derivative (2.8) is given by

$$\dot{V}_i = A_i \dot{V}_{i-1}.$$

Due to the definition in (2.11) the only difference between $\dot{V}_i$ and $\dot{V}_{i-1}$ is the value of the ith element $\dot{v}_i$. Consequently, the central part of Table 2.3, concerning the evaluation and derivation of the elemental functions φ_i, can be written as matrix chain

$$\dot{V}_l = A_l A_{l-1} \ldots A_2 A_1 \dot{V}_0.$$

To obtain a matrix vector product representation for the complete procedure given in Table 2.3, two projections P and Q with

$$P \equiv [I, 0, \ldots, 0] \in I\!R^{n\times(n+l)}, \qquad Q \equiv [0, \ldots, 0, I] \in I\!R^{m\times(n+l)} \tag{2.12}$$

are used for the initialization and finalization part, respectively. Thereby, P serves to expand the vector of independents to the dimension $(n+l)$ by adding the required number of zeros. Finally, the dependent variables can be extracted from $\dot{V}_l$ using the matrix Q. Now, the sequence of matrix vector products, representing the derivative calculations of Table 2.3, can be written as

$$\dot{y} = QA_lA_{l-1}\dots A_2A_1P^T\dot{x}. \tag{2.13}$$

Together with the definition (2.2) of $\dot{y}$ this sequence of products yields a factorization of the Jacobian

$$F'(x) = QA_lA_{l-1}\dots A_2A_1P^T. \tag{2.14}$$

Substituting $F'(x)$ within equation (2.10), yields a sequence of matrix vector products that will be used for deriving a general procedure for the reverse mode of AD.

$$\begin{aligned} \bar{x}^T &= \bar{y}^TQA_lA_{l-1}\dots A_2A_1P^T \\ \bar{x} &= PA_1^TA_2^T\dots A_{l-1}^TA_l^TQ^T\bar{y} \end{aligned} \tag{2.15}$$

Compared to the matrix vector product representation of the forward mode, equation (2.15) uses the reverse sequence of products based on the transposed matrices defined in (2.11) and (2.12), respectively. Again, a temporal snapshot $\bar{V}_i \equiv (\bar{v}_l, \bar{v}_{l-1}, \dots, \bar{v}_{2-n}, \bar{v}_{1-n})$ of intermediate variables gets defined prior to the differentiation and a snapshot $\bar{V}_{i-1}$ after the differentiation of φ_i with $i = l, \dots, 1$. Then, the derivative procedure for each elemental function φ_i can be described correspondingly by

$$\bar{V}_{i-1} = A_i^T\bar{V}_i.$$

This matrix vector product yields an algorithm for updating the intermediate adjoint variables $\bar{v}_{n+l}$ through $\bar{v}_{1-n}$ for each elemental function φ_i. Due to the structure of A_i^T, one obtains

$$\bar{v}_k^{i-1} = \begin{cases} \bar{v}_k^i + \bar{v}_i * \frac{\partial}{\partial v_k^i}\varphi_i(v_j)_{j\prec i} & \text{for} \quad k = 1-n, \dots, i-1 \\ 0 & \text{for} \quad k = i \\ \bar{v}_k^i & \text{for} \quad k = i+1, \dots, n+l \end{cases} \tag{2.16}$$

Hence, the last $n + l - i$ intermediate variables remain unchanged and are ignored in practice. The first $i - 1$ variables get updated according to the dependency relation and the ith intermediate is set to zero. Having a closer look at the updating algorithm, a certain difficulty becomes obvious. The update of the intermediate variables discussed for the reverse mode so far requires the knowledge of the values of the corresponding intermediate variables that were computed during the function evaluation. Due to this dependence, the reverse mode of AD can never be applied standalone but has to follow a function evaluation.

The remaining two matrix vector multiplications involving Q^T and P, respectively, are mainly necessary to adjust the dimension of the input vector $\bar{y}$ to the size $n + l$ and to extract the derivative solution $\bar{x}$ from the intermediate variable set, respectively, but introduce no new mathematical difficulties. Taking all dependencies into account the overall procedure of the reverse mode can be summarized as done in Table 2.4.

As for the forward mode, the complexity theory of AD provides a bound for the cost of the derivative computation [Gri00, pp. 56]

$$\omega_R = \frac{\text{TIME}(\bar{x})}{\text{TIME}(y)} \in [3, 4]. \tag{2.17}$$

Thus, complete gradients can be computed independent from the number of arguments of the given function F, using one execution of the reverse mode of AD.

Considering the storage requirements, some important observations can be made. In addition to each variable involved in the function evaluation a corresponding variable holding the derivative must be

Table 2.4: reverse mode of AD – General procedure

v_{i-n}	$=$	x_i		$i = 1, \ldots, n$
v_i	$=$	$\varphi_i(u_i)$		$i = 1, \ldots, l$
y_{m-i}	$=$	v_{l-i}		$i = m-1, \ldots, 0$
$\bar{v}_{l-i}$	$=$	$\bar{y}_{m-i}$		$i = 0, \ldots, m-1$
$\bar{v}_j$	$=$	$\bar{v}_j + \bar{v}_i * c_{ij}$	$\forall j \in \{j : j \prec i\}$	$i = l, \ldots, 1$
$\bar{v}_i$	$=$	0		
$\bar{x}_i$	$=$	$\bar{v}_{i-n}$		$i = n, \ldots, 1$

available in the reverse mode, as observed for the forward mode before. According to the update algorithm (2.16), partial derivatives for each elemental function φ_i need to be computed. This is only possible by knowing the values of their input variables. Due to the common practice of variable reusage, these values might have been overwritten meanwhile. To resolve this conflict, the value of the result variable of each elemental function is copied to a temporal storage place, immediately before it gets overwritten. During the reverse mode, the original values of overwritten variables can be restored using this information. As one result of the backup of values in execution order of the elemental functions, the temporal storage is often organized as sequentially accessed memory. A second observation concerns the size of the temporal storage. Since a value is stored for every elemental operation, the overall size of the sequentially accessed memory is a small multiple of the operation count of the underlying function F. For many practical tasks this size is more important than the memory required for storing the function and derivative variables. However, as long as the temporal storage can be kept within the main memory, the reverse mode of AD can evolve its strength nearly unaffected.

Based on the two main work modes provided by AD solutions to related problem classes, e.g. computing Jacobian matrices, can be provided. By extending and modifying the mathematical formulation, other problem categories, e.g., higher order derivatives, sparse matrices, are supported, too. For more details on these topics see [Gri00].

Despite the interesting theoretical complexity results, the applicability of Automatic Differentiation in a practical fashion is tightly coupled to the quality of its implementation and the usability of the resulting tools. In the following subsection the two basic implementation philosophies are introduced and discussed, briefly.

2.2 Basic implementation strategies

Automatic Differentiation has been implemented and used for more than 30 years. Though many obstacles have been removed during this period of time, not all issues could be solved up to now. In addition, the changing availability and popularity of computer programming languages has resulted in an heterogeneous foundation for the application of AD. A constantly growing collection of information on AD tools, publications, etc. can be found at `http://www.autodiff.org`. Historically, as well as based on the current research, most implementations belong to one of the two main strategies that are described within the following subsections.

2.2.1 Source-to-source transformation

To automatically create derivative code from a function given as source code in a certain programming language one needs a tool that is able to analyze at least this certain language and after a defined set of intermediate steps produces the intended output. For this purpose, the source code has to be scanned, data and control flow need to be revealed and optimized, and finally, a derivative code version has to be generated. Most of the inclosed work steps are well known from the field of compiler technology and related areas, i.e., debugging. However, as can be seen

> *Simply stated, a compiler is a program that reads a program written in one language – the source language – and translates it into an equivalent program in another language – the target language . . .*
>
> *. . . Target languages are equally as varied; a target language may be another programming language, or the machine language of any computer . . .*
>
> [ASU86, p. 1]

a tool that can afford a derivative augmentation cannot be treated as a traditional compiler in the sense of the quote above, as the target program is, of course, not equivalent to the source program. Since the user expects exactly this behavior of the applied tool, it seems appropriate to subsume it into the class of compilers, anyway. In practice, the process of generating derivative code this way is often called Source-to-Source Transformation using an AD-enabled compiler.

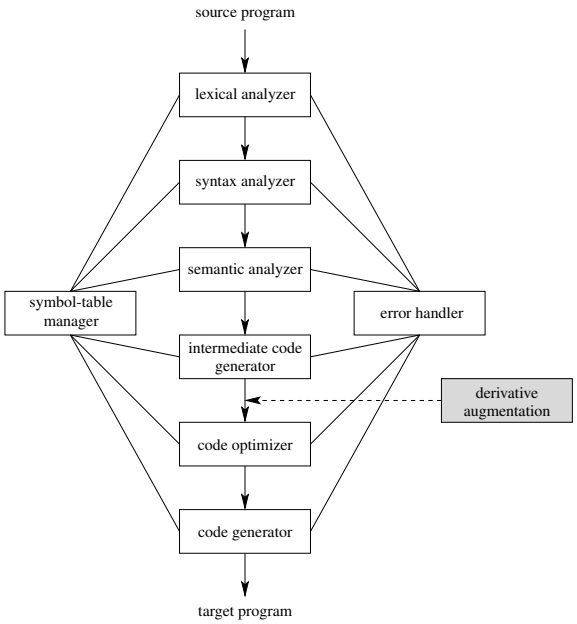


Figure 2.3: Basic phases of a standard compiler + derivative augmentation

As depicted in Figure 2.3, a standard compiler operates in six phases. In practice, some of them may be grouped together [ASU86, p. 10]. Figure 2.3 shows conceptually, where the derivative augmentation process may enter the standard phase chain. Accordingly, the derivative code generation is based on the intermediate code. This means in particular that the forgoing phases are common to both standard and AD-enabled compiler. In most cases the intermediate code is extended or in some rare cases replaced in parts [VGK04], respectively, by the derivative code through the derivative augmentation. Finally, after a code optimization the target program is created. Though the generation of machine code would be possible, current AD-enabled compilers generate source code in the programming language the source

program is written in. Then it is up to the user to combine derivative and function code in a meaningful way before applying a standard compiler to the result, to create the final executable.

Implementing Automatic Differentiation as AD-enabled compiler is as challenging as and, in most cases, as expensive as it is promising. As mentioned above, implementing a specialized compiler comprises significant effort in areas of compiler technology not directly necessary for the application of AD. The inherent complexity might be reduced by applying a sufficient level of abstraction, i.e., reusing standard compiler technology. Currently, almost all available Source-to-Source projects known to the author fully implement an own compiler. The OpenAD project [Utk04] can be seen as one exception.

On the other hand, due to this wide approach, full program information is available to the AD-part of the compiler. This includes control flow information that can be applied to the derivative code in an adequate manner. In theory, Source-to-Source transformation affords a maximal optimization, since all operations not influencing the dependent variables can be safely ignored. The latter is achieved through an activity analysis, i.e., by building the set of active variables that serves for identifying the relevant operations. Traditionally, static analysis at compile time was used but turned out to be too restrictive. Therefore, some ongoing research projects are focused on constructing an optimal set of active variables by taking dynamic runtime information into account, too, see, e.g., [KRE+06].

The availability of Source-to-Source tools is currently restricted to a subset of the existing programming languages. Historically, AD-enabled compilers were developed for the Fortran 77 programming language due to its, compared to other languages, very restricted and easier to manageable set of instructions, programmers can use in their codes. Many of the additional language features, introduced with Fortran 90/95, are now supported by many modern AD Source-to-Source tools, too. Examples of AD-enabled compilers for the Fortran programming language are, e.g., ADIFOR [BCKM96], TAF [GKS05], TAPENADE [HGP05, PH05]. AD-enabled compilers for C/C++ are under development and this work introduces or intensifies, respectively, theoretical issues already known from the additional language constructs, specified in the recent Fortran standards. This includes, e.g., the handling of pointers, dynamic memory, polymorphism, etc. Since many of these challenges are still unanswered, only two AD tools for C/C++ are available, at the moment and some restrictions apply. ADIC [BRM97] handles ANSI-C coded functions using the forward mode of AD and TAC++ [VGK04] affords forward and reverse mode AD on a subset of C.

Another relatively new field of application for Automatic Differentiation is the tool MATLAB [Mat06], enjoying increasing popularity in science and economy. This environment is basically a language for technical computing, offering efficient mathematical programming on a high abstraction level, among other things. Here, the Source-to-Source approach is characterized by the transformation of a MATLAB program into an augmented program by usage of an external tool, e.g., MSAD [KF06] that can be processed by MATLAB, afterwards.

Independent of the Source-to-Source tool further code optimizations are often wanted or needed, respectively. This results in the application of mathematically proven methods, as for example the exploitation of sparsity information of matrices [GK06]. Furthermore, known techniques are often extended, e.g., linearity analysis [SH06]. Likewise, new algorithms are developed that not necessarily follow the traditional forward and reverse schemes but implement AD in form of elimination techniques, applied to computational graphs, e.g. [Nau02, Nau04, TFPR01, TFP03].

The remaining parts of this thesis are focused on the second AD implementation strategy, i.e., operator overloading that is described in the following subsection.

2.2.2 Operator overloading

In contrast to developing an AD-enabled compiler as described in the previous subsection, the implementation of the forward and reverse mose, respectively, can be based upon a feature of many modern

programming languages, e.g. C++, Fortran90+, etc., – the operator overloading. As described before, the original function code has to be augmented to allow the derivative computation. Using the technique of operator overloading, this can be done on the operation level. In principle, all overloading based tools offer at least one AD-enabled data type, the user has to work with when starting the derivation process. For the new data type, all basic operations corresponding to the set Φ are overloaded, to allow the augmentation with derivative code directly or indirectly, respectively.

```
class ADtype {
    protected:
        Type value;
        Type derivative;

    public:
        ADtype operator* (const ADtype &arg) {
            ADtype result;
            result.value = value * arg.value;
            result.derivative = derivative * arg.value +
                                value * arg.derivative;
            return result;
        }
}
```

Figure 2.4: Example of an AD-enabled data type for the forward mode of AD

The most obvious approach to implement the derivative computation using the forward mode of AD is exemplarily depicted in Figure 2.4. Using C++ notation, it shows the propagation of derivatives together with function values. With this basic implementation, every occurrence of an elemental function $\varphi_i \in \Phi$ in the source code is replaced during the compilation phase by the corresponding overloaded version. This way, function and derivative value are available almost simultaneously, at runtime. After executing the augmented function, derivative values for all variables of ADtype are available via the variables derivative component. Though this approach offers an efficient implementation of the forward mode of AD, it is not very useful for reverse mode differentiation. This is due to the fact that the necessary program flow information, needed to be reversed, is no longer available as soon as the evaluation of the function is completed.

```
class ADtype {
    protected:
        Type value;
        Type ID;

    public:
        ADtype operator* (const ADtype &arg) {
            ADtype result;
            tapeOperation(MULT, result, ID, arg.ID);
            result.value = value * arg.value;
            return result;
        }
}
```

Figure 2.5: Operator overloading based AD using a taping mechanism

Therefore, to provide reverse mode differentiation, AD tools based on operator overloading need to create an internal representation of the function F. This can be done in various ways, e.g., graphs [BS96], tapes [GJU96], etc. and will be described using the *taping* technique.

The basic strategy as well as the name of the technique itself is derived from the usage of magnetic tapes with strictly sequential read/write access. Though the actual implementation varies, the core idea can be found in many AD tools offering reverse mode computation based on operator overloading. Essentially, only the function value is computed within the overloaded operator or intrinsic function, respectively. In addition, information exactly describing the operation is recorded onto a tape. This information is the type of the operation/function, e.g. MULT, SIN, etc., as well as representations of the involved result and arguments. Using the C++ notation again, Figure 2.5 illustrates the essential structure of a corresponding AD type featuring a taping mechanism. After the evaluation of the augmented function, the created tape now holds the sequence of operations that have been processed, in execution order. Based on this information, the program flow sequence can be easily inverted by evaluating the written tape in reverse order.

The main advantage of the taping approach as well as its heaviest drawback can be found in the program flow representation within the tape. Due to the taping process, all information is reduced to the level of arithmetic operations. This means, information concerning complex program internals, such as heredity, pointers, dynamic memory, etc., introducing significant difficulties to Source-to-Source transformation, is completely excluded. Computing derivatives using the taped information is therefore straightforward, at least theoretically. On the other hand, using the operator overloading approach, nearly all control flow information is lost. This means in particular that all loops get unrolled. Additionally, only those operations of program branches are taped that gets executed as result of the condition. Challenges arising from the reduced control flow representation are discussed in the next subsection.

Tools for Automatic Differentiation based on operator overloading are mainly available for the programming languages C++ and due to its covering also C, the mathematical environment MATLAB and FORTRAN90/95. Many of them, e.g., ADMAT [BLV03], ADOL-C [GJU96], CppAD [Bel07], FADBAD/TADIFF [BS96], and MAD [For06], are using an internal function representation and therefore offer both forward and reverse mode differentiation. Additionally, many specialized tools are available that focus on one of the two basic modes of AD, e.g. AUTO_DERIV [SPF00], COSY INFINITY [BM06], FAD [ADP01] for forward mode AD and RAD [Gay05, BGP06] for the reverse mode.

Depending on their implementation, all tools using a taping mechanism are effected by common issues that are described in the following subsection.

2.2.3 Common issues of tape based overloading strategies

Though computing derivative values on an operation based log file, the tape, enables a simplified view of the derivation problem, the performance of the resulting code suffers in many cases from exactly this simplification. The reduction of the information flow to the level of executed operations is accompanied by a nearly complete removal of control flow information. As a result, two main classes of important information are no longer available.

Loop information Obviously, after restricting taping activities to executed operations an abstract view of a loop's body is no longer available. This includes information about begin and end of the body as well as information on the loop indices and their usage.

Branch information Branches, not taken during the specific execution of a loop iteration, are not written onto the tape and cannot be used in re-computations, therefore.

The primary consequence of the minimized control flow information is the increase in tape size, mainly due to loop unrolling. Depending on the system, the feasible size of the written tape is often a limiting factor when computing function and derivative values based on this information. Even in case, the tape can be written, its storage nature, e.g., main memory, hard disc, etc., mainly defines the access cost of

the inherent information. This cost is an important factor for the overall computation time, especially if lower order scalar derivatives are to be computed.

Besides the unrolling problem several additional restrictions according to the tape usage arise from the reduced control flow information.

- Recalculation of function values as well as computation of derivatives at different base points is only possible as long as the original control flow information is unchanged. This means on the one hand that the number of iterations for each reevaluated loop must match the taped number. On the other hand and in more general, each control decision to be done must meet the taped representative.
- Many parallelization strategies, such as OpenMP [DM98], allow parallel processing of loops. Obviously, such an approach is not applicable without at least basic knowledge about the loop structure.

Computing derivative values in an adequate time often turns out to be a difficult tasks. Advanced concepts, developed to increase the efficiency of Automatic Differentiation based on operator overloading, are addressed in the following chapter.

3 Advanced concepts for operator overloading

Considered from a historical viewpoint, automatic differentiation based on a computational function representation is a quite young technique of computing derivative values. As discussed in Chapter 2, many challenges still have to be met and yet unknown problems may be revealed. Besides AD, additional concepts assisting the evaluation might be contained in or be applied to the function under consideration. Moreover, when using a tape based operator overloading approach, these concepts may enforce an exchange of information and control between the AD-tool and other tools applied to the function. Basic facilities applicable in such situations are discussed in this chapter.

In its simplest application, automatic differentiation based on a taping mechanism consists of two main steps. First, an internal representation, i.e., the tape, of the function is created. Subsequently, derivative values are computed using the generated tape and the forward or reverse mode of AD. If the problem under consideration is sufficiently small, this basic strategy may work well. For larger problems, additional information about the structure of the function should be exploited and a more advanced handling of tapes is desirable. Suitable facilities that have been developed in the scope of this thesis are discussed in Section 3.1.

Due to its nature, AD using the concept of operator overloading often requires the user to possess a higher level of code insight compared to application of source-to-source tools. In principle, all declarations of floating point variables may be altered to use a provided data type, i.e., the type offering the overloaded operators. Doing this is usually no optimal strategy in terms of code efficiency. Rather, only those variables should be changed whose AD-counterparts propagate derivative values. However, identifying this set of variables may be time consuming. To achieve a better trade-off between usability and runtime, the newly designed technique of state-tracking variables is introduced. It may be utilized to reduce the demands on the user when applying an overloading based AD-tool. Employing this approach in the taping process, all floating point variables may be replaced by a special data type that enables the AD-tool to identify the relevant variables at runtime. Details are given in Section 3.2. The development of the state-tracking technique resulted in partial overlaps with an undocumented approach used in the AD-tool CppAD [Bel07]. Differences to this technique are also discussed in Section 3.2.

One of the main challenges AD has to cope with is caused by the increasing availability of parallel computer hardware. In the future, it will probably be a crucial aspect for the acceptance of AD-tools, whether parallelism in the function can be exploited for the differentiation or not. Especially when applying the reverse mode of AD, it is important to detect if the adjoint procedure can be parallelized. Adequate parallelization strategies for tape based solutions that have been derived during the work for this thesis are introduced in Section 3.3.

3.1 Extended tape management

Creating an internal function representation, as done by many operator overloading based AD-tools, is a question of storing selected information, characterizing the evaluated function. This information must be detailed enough to allow the application of the reverse mode of AD on the one hand and, in many cases, allow a recalculation of the original function at the same or a different base point on the other hand. Although the actual implementation may vary, the basic structure is common to most overloading based AD-tools.

Considering a given vector-valued function $y = F(x)$, three types of information must be included into the corresponding tape. The vector x of independent variables has to be identified. Accordingly, the vector of dependent variables y has to be represented. In most cases, the transformation of the mapping F from x to y into an internal representation is the main part of the taping procedure.

This induces the question, in which way variables may be represented within a tape. In this context, two types of variables shall be distinguished: *Augmented* variables are of the special data type that affords the overloaded operators, whereas *non-augmented* variables are not. In the literature that addresses operator overloading based AD, *augmented* variables are often called *active* variables. Throughout this thesis, they are denoted by the term *augmented* to distinguish them from the results of the activity analysis. Within that analysis, all *active* variables are determined, i.e., variables that influence derivative values. Derivative instructions must then be performed only for those instructions that involve *active* variables. In the context of operator overloading based AD, the set of *active* variables forms a subset of the *augmented* variables.

The theory of the forward and reverse mode of AD, as discussed in Section 2.1, shows that for every function variable v a corresponding variable $\dot{v}$ and $\bar{v}$, respectively, is used to propagate the derivative values during the specific AD mode. Furthermore, the two variables $v, \dot{v}$ and $v, \bar{v}$, respectively, are accessed jointly, when handling operations in the derivation process. Using a unique ID for each augmented variable and considering all function and derivative variables to be stored as vectors, it is possible to address v, $\dot{v}$ and $\bar{v}$ safely. Such an ID is equivalent to the index within a vector but can also be treated as a unique name of a function variable. The interconnection between function and derivative values is represented by the ID across the vector borders. In other words, v, $\dot{v}$ and $\bar{v}$ share the same ID. If non-augmented variables are involved in an operation, the numerical value is used instead of an ID.

Using only the IDs of the involved variables does not result in a complete representation of an operation. Additionally, the kind of the operation has to be stored. This can be done by including into the representation an operation code (OPC) similar to that used in the binary format of computer programs. It is obvious that the range of OPCs must comprise all arithmetic instructions that shall be usable in any function to be differentiated. Though the declaration of independent and dependent variables of the represented function is possible without an explicit operation code, it can be beneficial to forgo this potential. That way, the complete tape is structured as a sequence of operations. Accordingly, the range of operation codes needs to be extended with codes for assigning independent and dependent variables.

Taking all necessary information into account, the following basic structure of a taped operation can be used:

$$operation := (\text{OPC, ID, } \{\text{ID} \mid \text{value, ID} \mid \text{value}\}). \tag{3.1}$$

Within (3.1) all elements between the curly brackets are optional and "|" separates two alternatives. Hence, a taped operation consists of an operation code and at least one ID of an augmented variable. The declaration of independent or dependent variables can be considered as example. Additional IDs are required when creating the representations of arithmetic instructions. Furthermore, the value is used whenever a non-augmented variable is involved. Finally, it is assumed that at least one argument of an arithmetic instruction is represented by an ID.

Limiting the function information contained in the tape to the discussed level implicates a major drawback: A switch in the control flow when evaluating the function at a different base point, will not be noticed. It is very likely that function and derivative values will be incorrect, in return. To overcome this problem, additional information is included into the tape that does not influence function or derivative values directly but guaranties the correctness of the results. Whenever a branch point in the program is reached where the decision depends on an augmented variable, a special operation is taped. The latter signals that the value of the corresponding variable must fulfill a given condition, e.g., the value must be less than zero. This condition is determined during the taping process and defines for which values of a certain variable the taped branch is valid. Whenever the condition is violated, an error message can be triggered and a re-taping process might be started if possible.

Thus, the range of operation codes must be chosen to represent the following three kinds of program statements:

- declaration of independent and dependent variables
- arithmetic instructions, i.e., elemental operations and intrinsic functions, including the assignment
- comparison instructions

Consider the following example and a corresponding source code depicted in the left half of Figure 3.1. There, C++ programming notation has been used. Creating an internal representation while evaluating

EXAMPLE 3.1

$$\begin{aligned} y &= F(x) \\ &= \begin{cases} (0.5 * x - 1)^2 & \textit{for } x \geq 2 \\ -1 * (0.5 * x - 1)^2 & \textit{for } x < 2 \end{cases} \end{aligned}$$

F at a point $x \geq 2$ results in a tape as depicted in the right half of Figure 3.1. The meaning of the used

1	declare_indep(x);	(DI , ID_0)	1
2	$t_1 = 0.5 * x$;	(MUL, ID_2, 0.5, ID_0)	2
3	$t_2 = t_1 - 1$;	(SUB , ID_3, ID_2, 1)	3
4	$t_3 = t_2 * t_2$;	(MUL, ID_4, ID_3, ID_3)	4
5	if $(x < 2)$	(GE , ID_0, 2)	5
6	$y = -1 * t_3$;		6
7	else		7
8	$y = t_3$;	(AGN, ID_1, ID_4)	8
9	declare_dep(y);	(DD , ID_1)	9

Figure 3.1: Function vs. tape representation for $x \geq 2$

operation codes is the following:

DI	declare an independent variable
MUL	multiplication
SUB	subtraction
GE	comparison for a variable to be greater or equal to a numerical value
AGN	assignment
DD	the declaration of a dependent variable

For the tape the following mapping between IDs and names of augmented variables has been utilized.

ID	0	1	2	3	4
augm. variable	x	y	t_1	t_2	t_3

Lines with the same number in the two parts of Figure 3.1 show the statements of the source code and the corresponding internal representation. As mentioned above, the ordering of result and operands in the statements is maintained in the tape. Notice also that the comparison operation at line 5 of the tape asserts the validity of the created representation. Whenever the tape gets reevaluated at a point $x < 2$, the assertion fails. Appropriate actions may be triggered in this case.

Currently, many applications that make use of AD benefit from a specific quality: The function to be differentiated can in principle be separated and processed as a whole using an appropriate AD-tool.

However, according to the function structure, additional properties may be exploited for specific parts of the function, e.g., fixed point iterations. This can be done either by using the same AD-tool or by applying external software. The latter may also include other AD-tools. Due to the optimized handling of every function part, a higher performance of the created code can be achieved. In some cases this may also be necessary to afford derivative calculations at all. In the following subsections two possibilities of handling different AD-strategies within the same program are discussed for the tape based AD-approach. In this context, it is assumed that the function $y = F(x)$ as given in (2.1) can be divided into smaller parts G_1 through G_k, such that

$$F = G_k \circ \dots G_2 \circ G_1, \tag{3.2}$$

with

$$y = x_{k+1}, \quad x_{i+1} = G_i(x_i), \quad x_1 = x \qquad i = 1, \dots, k.$$

Thereby, the dimensions of the vectors x_i do not need to be the same for the functions G_i. If different AD-strategies are to be applied to the specific regions G_i, a high level of steering by the user would be required. Reducing this burden by transferring as much work as possible from the user to the AD-tool will increase the attractiveness of the tool on the one hand and offers a good chance of optimizing the overall strategy on the other hand.

3.1.1 Nested taping

Throughout this thesis, the process of nested taping denotes the creation of several tapes by the same AD-tool. It is assumed thereby that the creation of an (*outer*) tape is interrupted by the creation of an (*inner*) tape. Hence, the taping process for an inner tape is always completed before the outer one continues. The nesting may apply to the inner tape too and so forth, creating overall a structure similar to the call tree of a computer program. The applied AD-tool must facilitate an appropriate infrastructure.

To afford the application of different AD-techniques for the various tapes, sufficient information concerning the tape interconnection as well as the intended usage must be included within the internal function representation. This way, no additional user interaction is necessary, once the overall taping process is completed and the derivation procedure has been initiated for the outermost tape.

As one result of the nesting strategy, for a given function F as in (3.2) each non-innermost tape contains at least two sub-functions G_i and G_j. To describe this setting mathematically, the derivative context is defined as follows.

> A **Derivative context** C contains one or several sub-functions G_i of a function F given by (3.2) that are processed to either apply or guide a specific AD-technique. If an internal representation of a context C is created, it is equivalent to exactly one tape. Re-taping a context does not break this rule. Links to other contexts C_j may be part of the context C_i or its internal representation, but may only occur between two consecutive sub-functions within C_i.

If a derivative context C_i does not need to or cannot be represented by a tape, the functionality necessary to afford the belonging derivation process must be available as additional part of the AD-tool or the user program. Context interaction must be included, accordingly. Contrarily, it is assumed that standard AD-technique is applied to every created tape. Changing the technique always requires a context and tape switch, that way. The techniques used for inner and outer context must comply in a meaningful way, moreover. In most situations, e.g., it makes no sense to compute higher order derivatives for the outer context and first order derivatives for the inner context.

For representing context switches within a tape a special operation code (CC - context change) is introduced.

context change operation: (CC, CCID)

Here, CCID is a unique number assigned for every context change, e.g., at the time of first occurrence. It is assumed that for every CCID a structure of data is available, holding all information necessary to perform the belonging context switches. This structure includes the ID of the context to switch to and the AD-technique to be applied, e.g., checkpointing, as well as all information necessary for its application.

A context change consists of the two context switches that result from the inner–outer partitioning of contexts. In terms of the outer tape, the context change can be treated as a way to transfer the control of the execution to another context. The processing of the outer tape is delayed until the work of the inner context is completed. Performing this switch from the outer to the inner context needs several information: The ID of the context switch and the intended AD-technique for the inner context must be known. Per definition, the ID is stored as argument of the representing operation. However, the intended AD-technique for the inner context may not be known at this time. Therefore, the work mode for the inner context is initialized with the current work mode. This information can then be used by the inner context to adjust the applied technique if necessary. Valid work modes are taping and any basic standard forward or reverse mode of AD. This information may not be included into the internal representation, as tape creation and evaluation may differ on that score.

Switching back from the inner to the outer context after task completion again needs the same information used on the way forth. However, the evaluated tape essentially offering this data is not visible at this point of time. Due to this lack of information, the complete handling of context switches in the inner-outer direction must be guided by the AD-tool. As tape information cannot be accessed directly, all necessary details must be saved and recommitted correctly. Assuming a special data structure enclosing all information belonging to a specific context change, a stack of these structures can be constructed. The stack size increases with each switch from an outer to an inner context, since the information describing the context change must be saved. In case the computations for a specific context are finished, the AD-tool examines the stack size and switches the context according to the data on top of the stack. Overall, the computations are finished as soon as the current context is completed and the stack is empty.

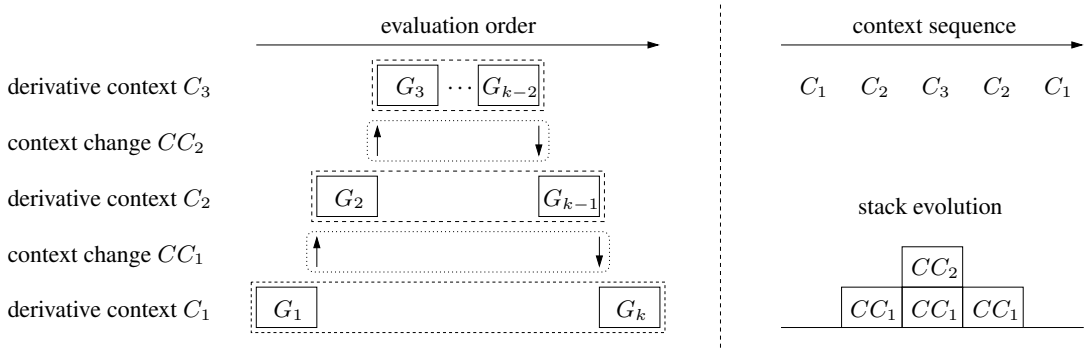


Figure 3.2: General context handling

Figure 3.2 illustrates the general handling of derivative contexts during the creation of an internal representation of a function F, given in (3.2), for an example situation. Thereby, derivative context C_3 is considered to contain all sub-functions G_i, $i = 3, \ldots, k-2$. The left part of Figure 3.2 depicts the interaction of sub-functions – solid rectangles, contexts – dashed rectangles, context switches – upward or downward arrows, and context changes – dotted ovals. Beyond, the corresponding evolution of the stack, which supports the context switches, is illustrated in the right part of the figure. Though exactly one context change is used for all but the innermost derivative context C_3 in this example, this setting is not binding. In particular, a context may contain more than one context change. Assuming that a tape is created for each of the three contexts, the basic tape structure depicted in Figure 3.3 follows.

Evaluating the generated tapes and computing derivatives using the forward or reverse mode of AD is then nearly identical to the usage of standard tapes that do not include context switches. New to the

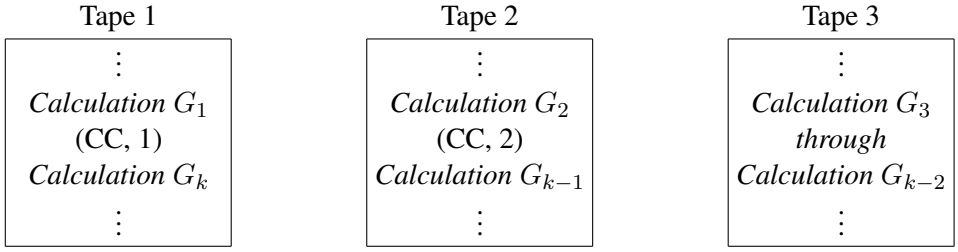


Figure 3.3: Tape representation according to the example of Figure 3.2

derivation procedure is only the special operation forcing a switch of the context. During the taping process, this operation is written at all points, where the user wants to enforce a context switch by applying an advanced AD-technique. The same sequence of changes has to be recreated when evaluating the tape forward and an exactly reversed sequence when applying the reverse mode of AD. The order of context switches when handling a context change operation is always the same, nevertheless. This means in particular that the context handling including the stack management is independent of the AD work mode.

Another important question concerns the exchange of values when performing a context switch. Since the sub-functions contained within all contexts are to be processed by the same AD-tool, a re-usage of variables can be assumed. Accordingly, copy operations for transferring function information during the taping process are not necessary. To guarantee the correctness of the computed derivative values, the derivation procedure must follow the same strategy.

Overall, applying standard AD-techniques to the created tapes and using advanced routines for contexts not represented by tapes, offers a well-adapted computation of the desired derivative values for the outermost tape, step by step.

3.1.2 External differentiated functions

Ideally, AD is applied to a given function as a whole. In practice, however, sophisticated projects usually evolve over a long period of time. Within this process, a heterogeneous code base for the project results quite often. This may include the incorporation of external solutions, i.e., different projects, as well as changes in programming paradigms or even programming languages. Equally heterogeneous, the computation of derivative values appears. Therein, different AD-tools may be combined with hand-derived codes based on the same or different programming languages. To support such settings, it is highly important that interoperability among the different tools is achieved. This can be provided for operator overloading based AD by the concept of external differentiated functions that has been developed within the scope of this thesis.

A sub-function G_j according to (3.2) is considered to be differentiated externally, if the control over the derivation process must be given off by the AD-tool and is resumed afterwards. Thereby, it is not necessary that another AD-tool is used to provide the derivative values. Using, e.g., lookup tables for providing these values would be sufficient, as well. All in all, the applied technique for the external part does not need to be known to the AD-tool.

The basic handling of external functions is quite similar to the nested taping approach. However, instead of changing a context internally, the program control has to be transferred to an external tool. An additional operation code (EF) is introduced to represent this special situation within a created tape. Comparable to the operation (CC), used to represent a context switch when applying nested taping, only one argument needs to be provided to enable the transfer of the program control to a different tool. This

argument is a unique number (EFID), which is used to access a special data structure containing all relevant information of the transfer. Accordingly, the structure of the taped operation is

(EF, EFID).

Assuming a sub-function $x_{j+1} = G_j(x_j)$ according to (3.2) to be differentiated externally, the corresponding data structure would contain the following information:

- the IDs for all elements of the independent vector x_j as well as the dependent vector x_{j+1},
- the number of independents and dependents, directly or indirectly,
- a handle of the external function, e.g., a function pointer.

As the external function should not be allowed to use internal variables of the AD-tool, copy operations are necessary to pass independent and dependent values.

Using this data structure, the general handling of an external differentiated sub-function G_j, embedded within two neighboring sub-functions G_{j-1} and G_{j+1}, can be described as follows.

1. **Taping step:**

   The process starts with the execution of all sub-functions up to G_{j-1}, creating an internal representation. After the completion of G_{j-1}, a mechanism provided by the AD-tool gets activated that controls the execution of the external function. This mechanism basically includes the following actions:

   - transfer the values of the independent vector x_j to user provided memory, applying a data type transformation from the internal to the external format,
   - include the IDs of all involved variables into the data structure representing the external function,
   - append the operation for the usage of an external function to the tape,
   - call the external function using the provided handle,
   - copy the computed values x_{j+1} from the user allocated storage back into the internal variables, again doing a data type conversion.

   Once the execution of the external function is completed, the interrupted taping process is resumed with the evaluation of sub-functions G_{j+1} through G_k.

2. **Forward evaluation:**

   Two basic constellations require the reevaluation of the internal function representation. Firstly, function values may need to be computed for the reverse mode differentiation at a different base point. Secondly, function and derivative values have to be computed at the same or a different base point utilizing the forward mode of AD. This may conclude the derivation procedure as well as it may prepare a higher order reverse mode derivation. Independent of the motivation, the evaluation order of the tape is equivalent to its creation order. Globally, derivative values for $\dot{y} = \dot{F}(x, \dot{x})$ can be computed as follows:

   $$\dot{F} = \dot{G}_k \circ \ldots \dot{G}_2 \circ \dot{G}_1, \tag{3.3}$$

   with

   $$\dot{y} = \dot{x}_{k+1}, \quad \dot{x}_{i+1} = \dot{G}_i(x_i, \dot{x}_i), \quad \dot{x}_1 = \dot{x} \qquad i = 1, \ldots, k.$$

   First, function and derivative values, respectively, are computed for all sub-functions G_1 through G_{j-1}. The next operation that is read from the tape now signals the change into an external part. Using the handle of the external function provided during the taping step and the IDs of the independent and dependent variables of sub-function G_j, control must be transferred to the external function. The latter is responsible for computing $\dot{x}_{j+1} = \dot{G}_j(x_j, \dot{x}_j)$ by whatever technique. It has

to be mentioned that intermediate values of the functions G_j and $\dot{G}_j$, respectively, that are needed for a succeeding reverse mode derivation, have to be handled within the external function. With the completion of the external part, the tape evaluation is resumed after transferring the computed results into internal variables. Thereby, function and derivative values, respectively, are computed for G_{j+1} through G_k according to (3.2) and (3.3), respectively.

3. **Reverse evaluation:**

   Compared to the handling of external functions when using forward mode differentiation, a few aspects are different when evaluating tapes in reverse order. During the derivation process, derivative values for $\bar{x} = \bar{F}(x, \bar{y})$ are computed and propagated the following way:

$$\bar{F} = \bar{G}_1 \circ \ldots \bar{G}_{k-1} \circ \bar{G}_k, \tag{3.4}$$

   with

$$\bar{x}_{k+1} = \bar{y}, \quad \bar{x}_i = \bar{G}_i(x_i, \bar{x}_{i+1}), \quad \bar{x} = \bar{x}_1 \qquad i = k, \ldots, 1.$$

   Due to the reversion, the evaluation starts with the derivation of the sub-functions G_k through G_{j+1}. For the external part, again the handle provided in the taping step must be used to access the external code. As in the forward evaluation, values to be transferred from the AD-tool to the external program part are identified using the IDs contained within the data structure that describes the external function. Derivatives for G_j are determined outside, using the function and derivative values stored during the preceding steps under external control. With the completion of the external part, computed derivatives are transferred back into internal variables. Afterwards, the interrupted derivation process is resumed by the AD-tool and derivatives for the sub-functions G_{j-1} through G_1 are computed.

All in all, by representing external program parts by a special operation within the created tape, the cooperation between different tools in an AD-context can be handled more user friendly. By providing all necessary information at the time of tape creation, the tape evaluation is just as comfortable as is the standard AD handling.

3.1.3 Checkpointing

As described in Section 2.1.2, the reverse mode of AD provides an efficient method to compute discrete adjoint information. Its main advantage is the bargain operation count necessary for computing the gradient of a scalar-valued function. It is only a small multiple of the operation count needed to evaluate the function itself (2.17). However, this advantage is accompanied by a memory requirement proportional to the operation count of the function evaluation. For real-world problems, storing the necessary intermediate values may result in extreme memory requirements. In this subsection, a technique is discussed that was developed within the scope of this thesis and allows to combine the advantages of the binomial checkpointing [Gri92, Wal99] and external differentiated functions.

To achieve a good measure between operation count of the reverse mode and its memory demand, a technique called checkpointing has been developed. Instead of storing all intermediate values necessary to compute the desired derivatives, intermediates are stored only at specific points of the computational process, called checkpoints. Whenever computed values are needed that have been overwritten meanwhile, they are recomputed using the information forming the most recent checkpoint.

In practice, according to the specific tasks, several checkpointing strategies have been developed. If the considered function evaluation has no specific structure, one may allow the user of an AD-tool to place checkpoints somewhere during the function evaluation. This simple approach is provided for example by the AD-tool TAF [GKS05]. Alternatively, the call graph structure of the function evaluation may be

exploited to place checkpoints at the entries of specific subroutines. This so-called joint reversal, see, e.g., [Gri00], leads to a reduction of the memory requirement. The subroutine-oriented checkpointing is used for example by the AD-tools Tapenade [HGP05] and OpenAD [Utk04]. As soon as one can exploit additional information about the structure of the function evaluation, an appropriate adapted checkpointing strategy may be used. This is, in particular, the case if the function evaluation comprises a time-stepping procedure, i.e., the state of the considered system evolves iteratively due to a direct or pseudo-time dependence. If the number of time steps l is known a priori and the computational costs of the time steps are almost constant, one very popular checkpointing strategy is to distribute the checkpoints equidistantly over the time interval. However, it is shown in [WG04] that this approach is not optimal. A more advanced approach is the binary checkpointing used for example in [Kub98]. However, optimal checkpointing schedules can be computed in advance, to achieve a minimal increase in runtime for a given number of checkpoints [Gri92, GW00].

Due to its nature, operator overloading may benefit in a special way when using checkpointing techniques that are based on the repeating of subtasks. Whereas source-to-source tools only need to take care of intermediate values, basically, the complete internal function representation is to be considered when estimating the memory requirements for the overloading approach. Accordingly, large problems may encounter unacceptable runtime increases much earlier, due to the unrolling problem described in subsection 2.2.3. Reusing tapes containing the internal representation of subtasks, e.g., by evaluating sub-functions at different arguments, will decrease the memory requirements, considerably.

Common to all checkpointing techniques is the combination of the following activities.

- storing the system state at a given point of the execution – taking checkpoints
- restoring the system state using the memorized information – restoring checkpoints
- advancing the evaluation to a given state without recording intermediate values
- computing adjoint values using a combined forward and reverse step for a certain part of the function
- combining the previous four aspects in an appropriate way

Having a closer look on these activities reveals that the combined forward and reverse steps may be handled by applying standard AD techniques, i.e., forward and reverse sweeps for the taped function parts. Without further knowledge about the structure of the function, the remaining activities could be left to the user. The situation changes, once the application is based on a time-stepping procedure. There, a given system is changed using a typically large number of transition steps until it reaches its final state. Storing all intermediate values on the way forward when preparing a reverse mode differentiation enforces the memory problem described above. Checkpointing techniques have been used for this type of applications successfully.

Due to the setting, additional information can be exploited, overall minimizing the demands on the user of an AD-tool by encapsulating the evolution part into a separate derivative context. Applying the nested taping approach described in Subsection 3.1.1, only limited information is to be provided by the user during the taping part. Describing the setting mathematically while adjusting (3.2), yields the function $y = F(x)$,

$$F = G_3 \circ G_2 \circ G_1 \tag{3.5}$$

with

$$x_1 = x, \qquad x_{i+1} = G(x_i) \quad i = 1, 2, 3, \qquad y = x_4.$$

Here, G_1 represents some initial computation preparing the main part G_2, i.e., the time-stepping procedure. Sub-function G_3 may be used for completing the function F, e.g., by computing some target values. G_1 and G_3 are assumed to be handled by standard AD in the outer derivative context C_1, which is represented by tape one. Sub-function G_2 that contains the time-stepping part forms the derivative context C_2.

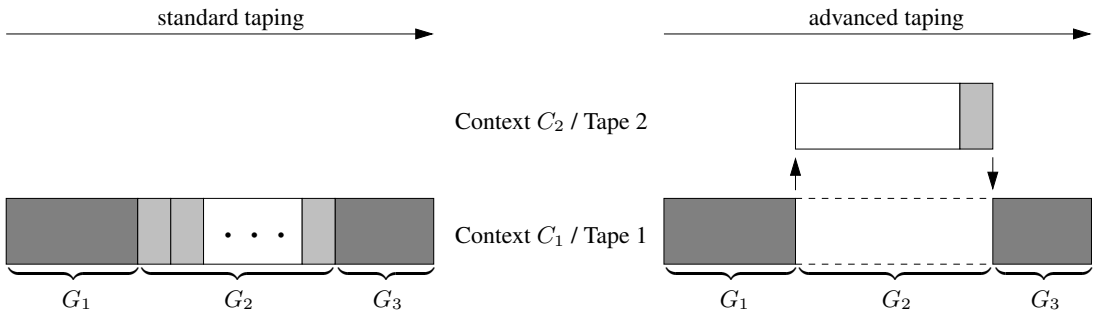


Figure 3.4: Standard vs. advanced taping for time stepping procedures

Applying the notation used by Griewank in [Gri00], the relationship between F and its sub-functions G_1 through G_3 is of type *strong*. Due to the standard taping approach, derivatives are computed using the split mode for all functions with the exception of F, since the global function is always derived in the joint mode. This means that all information concerning the work mode is handed down from the global function F to its sub-functions. Accordingly, sub-functions are in taping, forward or reverse mode, exactly when the global function F is, too. Whereas this is no problem for G_1 and G_3, which are represented by tape one, G_2 needs special attention.

In conformity with the definition of derivative contexts, C_2 may not be represented by a tape as a whole but must apply some advanced AD-technique. In this case, it is the application of checkpointing for computing adjoint values for a sequence of time steps. Assuming that the number of time steps l is known a priori a technique called binomial checkpointing may be applied, which has been proven to be optimal for this class of functions [GW00]. To steer the checkpoint handling for the given cases, the tool *revolve* [GW00] has been developed. However, its usage would suggest to encapsulate G_2 into an external differentiated function. Assuming that the applied AD-tool offers equivalent facilities, G_2 can be derived completely internally, that way avoiding the inflicted data copying.

Due to the contained time-stepping procedure, the sub-function $x_3 = G_2(x_2)$ has the following structure:

$$G_2 = T_l \circ \ldots \circ T_2 \circ T_1 \tag{3.6}$$

with

$$s_1 = x_2, \qquad s_{i+1} = T_i(s_i) \quad i = 1, \ldots, l, \qquad x_3 = s_{l+1}.$$

Mostly, $T = T_1 = \ldots = T_l$ holds, what means that the same function is used for all time steps. In general, only the arguments and the results differ among the steps. Exploiting this property, a second, significant reduction of the memory requirement can be achieved. Taking a step back and considering the case that only a single tape would be created for F, reveals the reason. As depicted in the left half of Figure 3.4, an internal representation of each individual execution of T would be written onto the tape. Besides the fact that a reevaluation of smaller tape parts during the checkpointing procedure would not be possible, a significant part of the tape would be redundant. As the execution of T only differs in the propagated values, the internal representation would be the same for all time steps. Therefore, instead of l identical representations of the time step function T, only a single step must be written as depicted in the right half of Figure 3.4. With regard to the subsequent reverse sweep, the last step is taped as representative and the resulting second tape may be evaluated every time the time step function is to be used.

Though not mandatory, further assumptions can be made to allow an optimal internal handling of the time-stepping part G_2. With exception of the last time step, the result of a certain time step is used as the argument of the succeeding step. Letting the result variables of a step overwrite the argument variables of the same step eliminates the copy operation that would be necessary to transfer information between two steps. From the computer science aspect, the time step function could then be written as $s = T(s)$. Further, it is assumed that the argument vector x_2 of function G_2 contains variables of augmented type

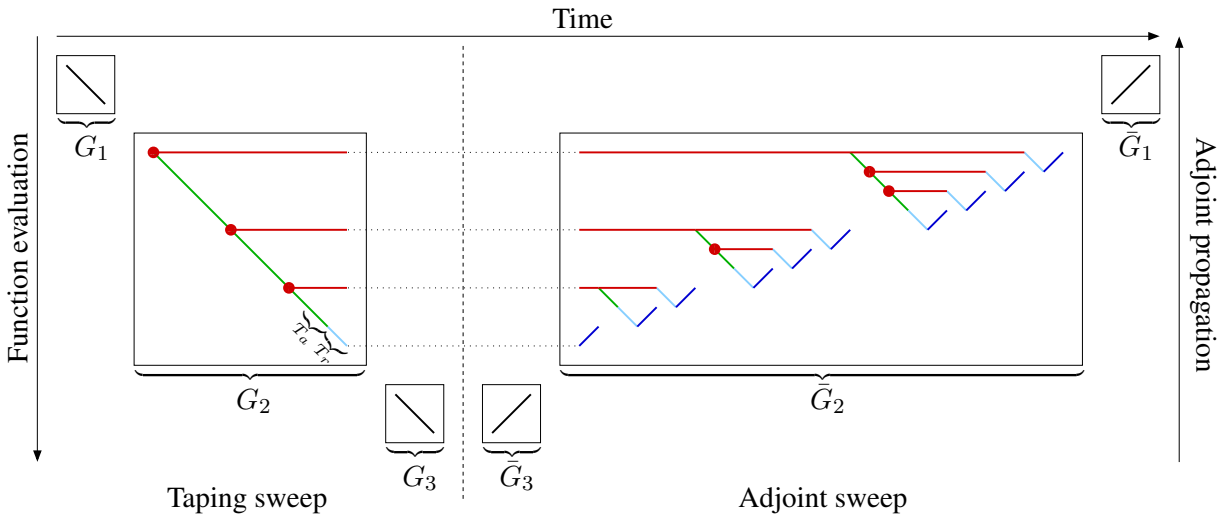


Figure 3.5: Embedded checkpointing – graphical representation

only and is equivalent to the state of the system to be changed by T. The checkpoints that are used during the derivation of G_2 can be handled completely internally, this way. If variables of non-augmented data type are part of the state, additional facilities must be provided by the AD-tool. This means that user-defined functions must be called at the time of checkpoint access to guarantee a correct mapping between the systems state and the checkpoint and vice versa.

Common to all checkpointing strategies is the storing of the initial function state within the first checkpoint. In terms of function G_2, the state s_1 has to be stored first. All remaining checkpoints are assigned later according to the binomial scheme described in [GW00]. To minimize the period of time that state information has to be stored, checkpoints should be written during the procedure that prepares the computation of adjoints in the reverse mode of AD. The preparing step may be either the tape creation including storing of intermediate values in case a first order reverse mode is to be executed subsequently or the computation of derivative values for a written tape applying a higher order forward mode of AD.

Due to the computation of derivatives for G_2 in split mode, the general checkpointing procedure as described in [GW00] cannot be applied directly but needs some small changes. First, the checkpointing procedure has to be split into two evaluation sweeps. Figure 3.5 depicts the procedure for the combination of a taping and a reverse sweep on the function F, defined in (3.5). Checkpoints for G_2 are created during the taping sweep and are depicted by the red circles. Due to the splitting, checkpoints must not be destroyed at the end of the taping sweep. The light blue line in the left half of Figure 3.5 represents the tape creation for the last evaluation of the time step function, i.e., the recording motion T_r. Advancing steps T_a that are depicted by the green lines, are used to transfer the system to a given state. The longer the green lines, the more time steps are represented. Thereby, the length of the light blue line serves as reference for the length of a time step. Adjoint values are propagated during the reverse evaluation for the time steps, depicted by the dark blue line.

Whereas the taping step and the reverse steps can be handled by the AD-tool directly, the optimal strategy for the advancing steps needs further investigation. Even though the version of the time step function T, provided for the taping case, can probably be used for the advancing steps too, this is likely aligned with a runtime drawback. As only function values have to be computed, the user should be pushed to provide a second version of the time step function, that is based on standard data types.

Following these facts, a successful implementation of the checkpointing approach using the context change and nested taping approach needs at least the following information and functions:

- an implementation of the time step function using standard data types – T_a,
- an implementation of the time step function based on augmented data types – T_r,

- information on the size and location of the input and output parameters,
- the number l of time steps to be reversed,
- the number c of checkpoints to be used and
- a so far unused tape number t_i for the inner tape.

Considering the situation from the users point of view, the complexity of the embedded checkpointing part has decreased significantly. Assuming that the necessary functions T_a and T_r have been provided, the derivation procedure for the global function F would follow the scheme below.

1. Initiate the taping process for F.
2. Declare the vector x as independent.
3. Evaluate the sub-function G_1, which is based on augmented data types.
4. Advice the AD-tool to apply a checkpointing procedure to G_2, using the provided functions T_a and T_r, the numbers s, c and t_i as well as the information exactly describing the input and output vector s of the time step function T.
5. Evaluate the sub-function G_3, which is based on augmented data types.
6. Declare the vector y as dependent.
7. Finish the taping process.
8. Evaluate the tape using the provided facilities of the AD-tool.

Considering point 3 and 5 of the above scheme to be handled the usual way when applying a tape based AD-tool, the effort for handling the checkpointing part is reduced to the activities that are summarized in 4. In particular, no user intervention is needed when evaluating the created tape in the forward or reverse mode of AD. With this technique, an improved usability of the AD-tool is achieved which also yields a better maintainability and a higher error safety. Principally, every other checkpointing algorithm besides the binomial checkpointing might be handled analogously.

3.1.4 Fixed point iterations

Another application that may benefit from the nested taping technique is the gradient calculation for fixed point iterations [Chr94, SWGH06]. From the mathematical point of view, this type of task is well investigated. However, the efficient practical handling in terms of AD did not received the same level of attention. In the remainder of this subsection, a brief introduction to the mathematical aspects of the differentiation of fixed point iterations is given, first. It is followed by the discussion of the integration into AD-tools using the concept of nested taping and derivative contexts, which was an object to this thesis.

Fixed point iterations are a common technique of solving partial differential equations (PDEs), e.g., in the field of fluid dynamics. If a PDE cannot be solved directly, an additional time dependency is introduced such that the solution can be computed iteratively for a given set of design parameters as a quasi-steady state of the modeled system. Let the corresponding fixed point equation $H : I\!R^{n_s} \times I\!R^q \rightarrow I\!R^{n_s}$ for the system be given by

$$s = H(s, u), \tag{3.7}$$

where $s \in I\!R^{n_s}$ denotes the state of the system and $u \in I\!R^q$ is the vector of design parameters. The evaluation of a solution for (3.7) is done by computing the iteration G_2

$$G_2 : \quad s_{k+1} = H(s_k, u) \qquad k = 0, 1, \ldots, \tag{3.8}$$

yielding a converging sequence $\{s_k\}$ that depends on the specific control u. Then, equation (3.7) is fulfilled by the limit point s_* of $\{s_k\}$. Assuming that $||\frac{\partial H}{\partial s}(s_*, u)|| < 1$ holds for any pair (s_*, u)

satisfying (3.7), a differentiable function $\phi : \mathbb{R}^q \to \mathbb{R}^{n_s}$ exists, such that $\phi(u) = H(\phi(u), u)$. There, the state $\phi(u)$ is a fixed point of H for a given control u.

In practice, fixed point iterations are often enclosed into a larger computational context that is given by a function $F : \mathbb{R}^n \to \mathbb{R}^m$, $y = F(x)$, which can be described mathematically by a composition of mappings

$$\underbrace{x \overset{G_1}{\mapsto} (s_0, u) \overset{G_2}{\mapsto} (s_*, u) \overset{G_3}{\mapsto} y}_{y = F(x)}.$$

There, a startup calculation G_1 is used to compute the initial state s_0 of the fixed point iteration G_2 as well as the design vector u, based on the parameter vector x. Finally, after the fixed point iteration G_2, the target vector y is computed according to the function G_3.

To optimize the system $y = F(x)$, derivatives of F with respect to x are needed. Depending on the ratio of the dimensions $\dim(y)$ and $\dim(x)$, either the forward or the reverse mode of AD is preferable to compute these values. Below, $\dim(y) = 1$ and $\dim(x) \gg 1$ is assumed for the sake of clearness. Thus, the gradient $\nabla F(x)$ can be computed using the scalar reverse mode of AD. For information on the application of the forward mode see [SWGH06].

Applying the reverse mode of AD onto the maps G_1 and G_3 is considered to be straightforward, i.e., black box AD can be used. For the fixed point iteration G_2 a more advanced handling might by appropriate. Turning to function $\phi(u)$ again, the total derivative $d\phi/du$ is defined by

$$\frac{d\phi}{du}(u) = \frac{ds_*}{du} = \frac{\partial H}{\partial s}(s_*, u)\frac{ds_*}{du} + \frac{\partial H}{\partial u}(s_*, u). \tag{3.9}$$

Due to its structure, (3.9) is a fixed point equation for ds_*/du, itself. Analytically, its solution can be expressed by

$$\frac{d\phi}{du}(u) = \frac{ds_*}{du} = \left(I - \frac{\partial H}{\partial s}(s_*, u)\right)^{-1} \frac{\partial H}{\partial u}(s_*, u). \tag{3.10}$$

Performing calculations according to this scheme might be numerically too expensive. Then, equation (3.10) serves for a deeper analysis of the reverse mode of AD, for this specific type of application. Applying the theory of the reverse mode described in Section 2.1.2 onto equation (3.10) yields for a given vector $\bar{s} \in \mathbb{R}^{n_s}$ the identity

$$\bar{s}^T \frac{ds_*}{du} = \bar{s}^T \left(I - \frac{\partial H}{\partial s}(s_*, u)\right)^{-1} \frac{\partial H}{\partial u}(s_*, u).$$

Setting $\zeta^T := \bar{s}^T \left(I - \frac{\partial H}{\partial s}(s_*, u)\right)^{-1}$ and applying some transformation, reveals the following fixed point equation for the adjoint information

$$\zeta^T = \zeta^T \frac{\partial H}{\partial s}(s_*, u) + \bar{s}^T.$$

Again, a fixed point iteration $\bar{G}_2$ can be used to compute a solution and one obtains for $l = 0, 1, \ldots$

$$\left(\zeta_{l+1}^T, \bar{u}_{l+1}^T\right) = \left(\zeta_l^T \frac{\partial H}{\partial s}(s_*, u) + \bar{s}^T, \zeta_l^T \frac{\partial H}{\partial u}(s_*, u)\right). \tag{3.11}$$

The converging sequence $\left\{\zeta_l^T H'(s_*, u) + (\bar{s}^T, 0^T)\right\}$ yields the unique fixed points

$$\zeta_*^T = \bar{s}^T \left(I - \frac{\partial H}{\partial s}(s_*, u)\right)^{-1} \qquad \text{and} \qquad \bar{u}_* = \zeta_*^T \frac{\partial H}{\partial u}(s_*, u) = \bar{s}^T \frac{ds_*}{du}.$$

It was shown in [Chr94] that the convergence rates are equal for the original fixed point iteration (3.8) and the derivative fixed point iteration (3.11). However, this does not necessarily mean that the number of

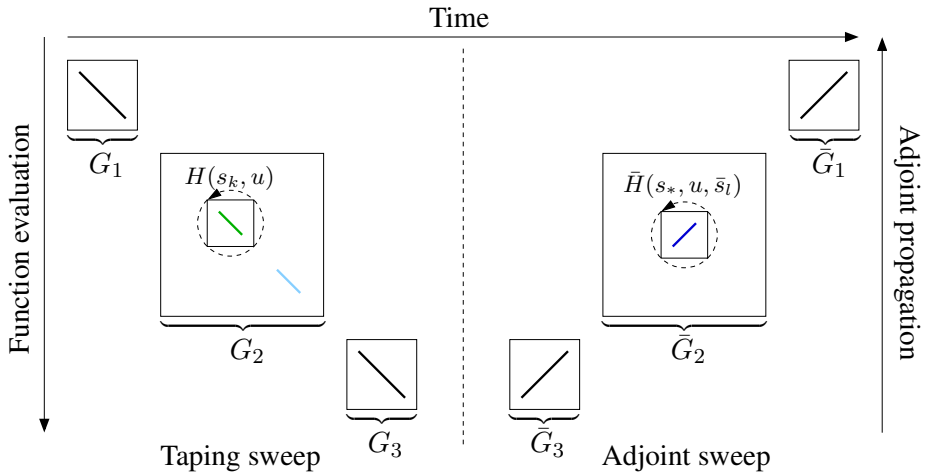


Figure 3.6: Derivation procedure for functions containing a fixed-point iteration

iterations k and l are the same. Computing as many iterations in the reverse mode as done in the forward mode, i.e., in the black box application of AD, is not appropriate in most cases, therefore. Possible negative effects are an unnecessary high number of iterations for the derivative fixed point iteration, an insufficient accuracy of the derivative fixed point or even a missing convergence, in the worst case.

A solution to this problem can be found in a sufficient decoupling of the original and the derivative fixed point iteration. This can be achieved by using a nested taping approach as described above, enabling an advanced handling of the two fixed point contexts. Due to the decoupling, a reduction in the size of the internal function representation is possible, in addition. The principle layout of the taping process is equivalent to the strategy depicted in Figure 3.4, therefore. As for the checkpointing approach, the sub-functions G_1 and G_3 are combined within the derivative context C_1 that is represented by tape one. The latter also contains the special operation that signals to switch to the second context C_2 containing the handling of the fixed-point iteration G_2. Throughout this process a second tape is created, representing the last step of the iteration G_2, i.e., the approximation of $s_* = H(s_*, u)$.

The complete process of computing derivatives using the reverse mode of AD for a function containing a fixed-point iteration is depicted in Figure 3.6. Thereby, it is assumed that a first order reverse mode is applied. In preparation, the function evaluation including the recording of intermediate values and the creation of the internal function representation are combined within the taping sweep. This is depicted in the left half of Figure 3.6. Within a first step, sub-function G_1 is handled the standard way, i.e., the function is evaluated and an internal representation is created as the first part of tape one. After adding the context change operator to the tape, an internal switch to a new context C_2 is accomplished. There, the fixed-point iteration (3.8) is used to solve the fixed-point equation (3.7). Throughout this process, no internal representation of the involved function is created. In Figure 3.6, this process is depicted by the dashed arrow around the box containing a green line that represents the evaluation of H at the kth base point (s_k, u). As soon as the fixed-point s_* has been approximated, function H gets evaluated a last time at the base point (s_*, u), this time to create an internal representation of H while recording intermediate function values. During this process that is denoted by the light blue line in Figure 3.6, a second tape is created. After returning to derivative context C_1, an internal representation for sub-function G_3 is generated and appended to tape one. Again, intermediate values are recorded throughout this process.

To start the computation of adjoint values, the user provides the vector $\bar{y}$ and initiates the evaluation of tape one, e.g., by forcing a first oder reverse mode application. The remaining part of the derivation procedure is now completely automatic, what means that no more user interaction is needed. After computing the adjoint vector $\bar{s}_*$ using the internal representation of G_3, a context switch is performed. Within the new context $\bar{C}_2$, the adjoint fixed-point iteration (3.9) is computed. Throughout this process that is depicted by the dashed arrow around the box containing a dark blue line, tape two and the corresponding intermediate values s_* and u are reused. In contrast, the adjoint values $\bar{s}_l$ corresponding to the systems

state are changed iteratively. The result of the adjoint fixed-point iteration is the vector $(\bar{s}_0, \bar{u})$, that is used as input vector for $\bar{G}_1$, after the first context has been resumed. Finally, as soon as tape one has been processed completely, the result vector $\bar{x}$ may be extracted the usual way.

The nested taping approach in conjunction with separate derivative contexts is capable to decrease the demands on the user when applying AD to more complex functions. Besides this advantage, such complex systems can be handled with adjusted copy overhead and based on a reduced function representation, internally. Another technique that is useful when addressing the latter issues is described in the following section.

3.2 Tape reduction based on activity-tracking

To be suitable for large-scale applications, an AD-tool must afford the computation of derivative values in a process that should be optimized in the highest possible degree. Operator overloading based tools are often said to be not feasible for handling such applications due to their lack of an appropriate data, control and activity analysis. The latter is a crucial aspect in the generation of derivative codes by AD-enabled compilers. By using a static activity analysis at compile time, more efficient derivative codes can be created in the way that the amount of trivial derivative computations is reduced considerably [CNR03].

When applying operator overloading based AD, a static activity analysis is not performed by the compiler as exploiting activity information does not belong to the standard optimization techniques. However, a similar analysis and an optimization of the derivation process based on the gathered results can be performed at runtime. This also allows for recent results of the research in the field of hybrid static/dynamic activity analysis [KRE$^+$06]. In the remainder of this section, a technique is introduced that was developed in the scope of this thesis and allows to apply a runtime activity analysis.

Before going into details, reconsider the coordinate transformation example introduced in Section 2.1, again. This time a situation more common to the application of AD shall be assumed, i.e., only a subset of input and output variables of the user function is relevant for the derivative computation. Here, for the given user function $(y_1, y_2, y_3) = F(x_1, x_2, x_3)$, derivatives of y_1 with respect to x_1 and x_2 shall be of interest only.

An activity analysis carried out by an AD-enabled compiler would determine all variables of the given user function F that derivatives must be computed for. This process results in the generation of the *set of active variables*

$$AV := \{v \,|\, \exists x \in XI : x \prec^+ v \wedge \exists y \in YD : v \prec^+ y\} \cup XI \cup YD. \tag{3.12}$$

Thereby, v is an intermediate variable, XI is the *set of independent variables* $\big(\{x_1, x_2\}\big)$, and YD is the *set of dependent variables* $\big(\{y_1\}\big)$. Furthermore, $\prec^+$ is the *transitive closure* of $\prec$ as defined in [SW89], i.e., $\prec^+ = \cup_{i=1}^{\infty} \prec^i$, where $\prec^i$ is a composition of i times $\prec$. The process of activity analysis starts with a differentiable dependency analysis [HNP05], wherein for each particular structure of the function, i.e., instruction, basic block or subroutine, a dependency set Dep is determined. Denoting by v_b a variable *before* a specific program structure and by v_a a variable *after* it, the dependency set is defined by $Dep := \{(v_b.v_a) \,|\, v_b \prec^+ v_a\}$. Data flow equations are used to determine the set Dep within this analysis. They are based on the more general equations introduced in [ASU86]. Further details on activity analysis are given in [HNP05].

Once the set Dep has been determined for all structures of the function, activity information is propagated along the computational graph using this dependency information. To generate a preferably small but nevertheless correct set of active variables AV, activity information is propagated ideally along the graph in two directions. In one pass, all *varied* variables are determined, i.e., the intermediate variables that possibly depend on at least one element of XI. All *useful* variables, i.e., the intermediate variables that at least one element of YD depends on, are determined in a second pass. The final set AV is build as

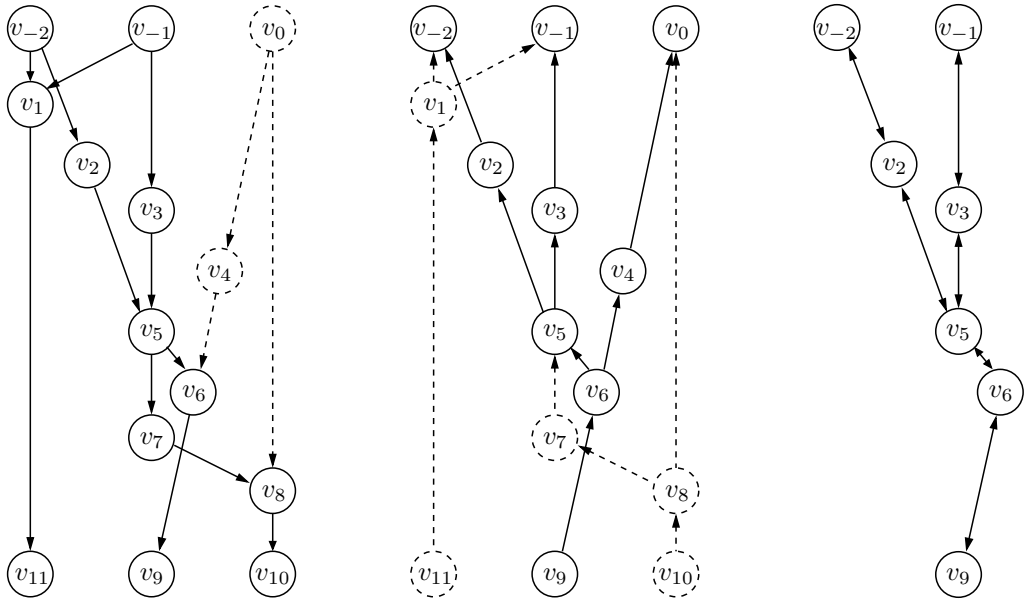


Figure 3.7: Forward, backward activity propagation as well as the intersection for the coordinate transformation problem

union of independent, dependent and the intersection of *varied* and *useful* intermediate variables, thus implementing (3.12).

The result of the activity analysis for the coordinate transformation example is depicted in Figure 3.7. In the left part, activity information is propagated in evaluation direction for the determination of *varied* variables. Thereby, dashed circles represent variables that do not need to be active and dashed arrows depict operations that must not be cared about in the derivation process. The propagation of activity information against evaluation direction to identify *useful* variables is depicted in the middle part of Figure 3.7. Creating the set of active intermediate variables as intersection of the results leads to the graph that is depicted in the right part of Figure 3.7. As can be seen, neither the forward nor the backward propagation of activity information can yield the desired result on its own.

Considering the application of an AD-enabled compiler, only the set of independent variables, the set of dependent variables and the function to be differentiated need to be provided. Extracting the activity information and building an optimized code based on this information is then carried out by the compiler. The derivative code might be called at an appropriate point of the program, afterwards. Though this approach can be summarized this easily, its application is limited due to the range of supported programming languages, i.e., languages for that AD-enabled compilers are available. The compiler development is typically extremely expensive and many challenges that result from the AD component are still unanswered. Especially newer language constructs, e.g., pointers or heredity, are responsible for this difficulties.

Applying an overloading based AD-tool follows the derivation scheme for AD-enabled compilers in many aspects. Due to its nature, certain differences occur, nevertheless. The most important difference in this context concerns the initial modifications of the source code. As mentioned in subsection 2.2.2, all independent, relevant intermediate and dependent variables of the function have to be replaced by an augmented data type. The meaning of this process is in some sense similar to the forward propagation of activity information and, so far, presents the only option of an activity analysis in the overloading approach. In the best case, the set of variables that gets redeclared is equivalent to the set of *varied* variables. Whether this result can be achieved is mainly related to the way these variables are chosen by the user.

3.2.1 Common augmentation strategies

Up to now, two basic techniques have been used when preparing a given source code for the application of operator overloading based AD. These two approaches mainly differ in the users effort and in the accuracy when determining the minimal set of augmented variables.

Global change of the variable type

An obvious strategy that can be applied with minimal effort in many cases, is to redeclare every floating point variable of a given source code to use a provided augmented data type. This can be done, e.g., by utilizing the MAKRO mechanism provided by many programming languages. Code changes are necessary only for a limited part of the provided source files, e.g., header files, this way.

The advantage of obtaining an augmented version of the code very easily and fast is accompanied by significant additional costs when evaluating the resulting code. As an internal representation is created for every operation that involves variables of augmented data types, the size of the required memory to keep this information may be increased drastically. Unfavorable memory access costs may be the result on the one hand and the processing of operations that are irrelevant for the derivation, on the other hand. This results in a higher overall runtime of both the function evaluation and the derivation calculations.

Compiler-aided augmentation

A more advanced technique of building the set of augmented variables is characterized by the usage of the compiler for assisting the user in this process. Thereby, in the first step only those variables are redeclared to an augmented data type that are independents in terms of AD. During the compilation process of the program, the compiler will issue error messages concerning data type conversions that are not allowed. Based on these error messages the user redeclares additional variables, i.e., certain intermediates or dependents, to rely on an augmented data type. Then, the program compilation process is invoked again, possibly issuing different error messages. The loop over changing the type of variables and examining the result by using the compiler has to be carried out until the last error message has been resolved. Compared to the global change strategy, a reduced set of augmented variables is created and a smaller internal function representation is generated during the taping step. However, an overestimation of the set of relevant variables can hardly be avoided, in most cases. Every operation that involves at least one argument of augmented data type will enforce the result of the operation to be of augmented type, too. This is even the case if the propagated derivatives do not influence any single dependent variable. The consequences of this issue are illustrated in the left graph contained in Figure 3.7.

To better rate the effort of the compiler-aided augmentation process, reconsider the coordinate transformation example, again. For the function $(y_1, y_2, y_3) = F(x_1, x_2, x_3)$ derivatives shall be computed only with respect to x_1 and x_2. Applying the compiler-aided technique results in the sequence of augmentation steps that is depicted in Table 3.1. Thereby, a compiler sweep and the augmentation of some variables according to the issued error messages are the components of one step. For each operation, arguments of augmented data type are set bold. As can be seen, even this small example takes seven compiler sweeps, not counting step zero that does not need one. Using this technique for activating a large code that is split across a significant number of files, often turns out to be exhausting or even impossible.

Comparing the compiler-aided augmentation technique and the compiler internal activity analysis reveals several correlations. In particular, the forward propagation of activity information by the AD-enabled compiler and the aided augmentation basically pursue the same goal. If the potentially frustrating complexity of the compiler-aided augmentation technique could be overcome, its underlying idea would well be beneficial to overloading based AD. An appropriate solution is described in the following subsection.

Table 3.1: Compiler-aided augmentation process

step	augmented variables	relevant operations		
0	x_1, x_2	declare independent		
1	v_{-2}, v_{-1}	$v_{-2} = \mathbf{x_1}$	$v_{-1} = \mathbf{x_2}$	
2	v_1, v_2, v_3	$v_1 = \mathbf{v_{-1}}/\mathbf{v_{-2}}$	$v_2 = \mathbf{v}^2_{\mathbf{-2}}$	$v_3 = \mathbf{v}^2_{\mathbf{-1}}$
3	v_5, v_{11}	$v_5 = \mathbf{v_2} + \mathbf{v_3}$	$v_{11} = \arctan(\mathbf{v_1})$	
4	v_6, v_7	$v_6 = \mathbf{v_5} + v_4$	$v_7 = \sqrt{\mathbf{v_5}}$	
5	v_8, v_9	$v_8 = \mathbf{v_7}/v_0$	$v_9 = \sqrt{\mathbf{v_6}}$	
6	v_{10}	$v_{10} = \arctan(\mathbf{v_8})$		
7	y_1, y_2, y_3	$y_1 = \mathbf{v_9}$	$y_2 = \mathbf{v_{10}}$	$y_3 = \mathbf{v_{11}}$

3.2.2 Augmentation using state-tracking variables

The two augmentation strategies described above often have turned out to be not well suited for larger real world problems. Instead, a technique would be necessary that affords to exploit the advantages of the two approaches under extensive avoidance of their drawbacks. In particular, it should be possible to redeclare all variables of floating point type to use an augmented data type without caring about an appropriate activity structure but achieving the optimal structure, nevertheless. Whereas this seems to be contradictory at first glance, it turns out to be possible on closer inspection. Up to now only pre-compile time and compile time techniques have been used in the context of overloading based AD-tools. To achieve the ambitious goal formulated above, runtime information must be taken into account, too. This seems reasonable as the creation of an internal representation that is done, e.g., for tape based overloading tools, is dedicated to the execution of the function rather than its compilation. Then, the possibly repeated evaluation of the created tape is the key component for the runtime of the derivative computations. Burdening the taping process with some additional checks for activity information seems acceptable if the size of the created internal representation is reduced in return.

Reconsidering the activity analysis of an AD-enabled compiler, and comparing it to the taping process of an overloading based AD-tool shows that only the process of determining *varied* variables can possibly be integrated into the taping process due to the evaluation direction. As a result of the overloading strategy, the global view of the source code that is utilized by the compiler is reduced to the level of individual instructions that are processed at runtime. Selective equations and definitions from the compiler based activity analysis can be transformed to reflect this situation and derive an appropriate strategy to achieve a similar result at runtime.

In the differentiable dependency analysis, for every structure S of the source code a set of dependencies Dep is determined, as mentioned before. With the tape based operator overloading approach, the range of structures is reduced to individual instructions that are actually carried out and a sequence thereof, respectively. An instruction can be either the declaration of an independent or a dependent variable, a control flow command or an arithmetic statement. The control flow commands, e.g., comparisons, do not influence the activity property of variables, whereas the declaration of independent and dependent variables do, due to (3.12). Therefore, in an adapted differential dependency analysis, only the arithmetic statements must be taken into account. For an arithmetic instruction $I_k : v = e$, the set Dep is defined by

$$Dep(I_k) := \big\{(d.v) \,|\, \forall d \in DP(e)\big\} \cup \big\{(w.w) \,|\, \forall w \neq v\big\}. \tag{3.13}$$

Thereby, v is the variable that takes the result of the instruction, w may be any other variable, e is an expression that forms the right-hand side of the instruction and DP is the set of variables that occur in differentiable position in e [HNP05]. Due to the overloading, the range of expressions is quite limited and can be summarized as follows.

e	$\text{op}(arg)$	$\text{op}(arg1, arg2)$	$\varphi(arg)$	v	c
$DP(e)$	$\{arg\}$	$\{arg1, arg2\}$	$\{arg\}$	$\{v\}$	$\emptyset$

From left to right, the expression e can be an unary operation, a binary operation, an intrinsic function, another variable or a constant. Compared to the differential dependency analysis that is performed by an AD-enabled compiler, two main differences occur. Firstly, the complexity of expressions is reduced significantly and secondly, no array handling is required. The latter results from the individual inspection of array elements and eliminates the overestimation of *varied* variables that is often necessary when applying an AD-enabled compiler.

As no variables can be introduced or old ones can be deleted by an arithmetic instruction, the sets of defined variables $\{v_b\}$ *before* and $\{v_a\}$ *after* the instruction are identical. This is not true for a sequence of instructions, in general. For the sequential composition $\otimes$ of two instructions I_1 and I_2, the dependency set $Dep(I_1; I_2)$ is defined as

$$\begin{aligned} Dep(I_1; I_2) &:= Dep(I_1) \otimes Dep(I_2) \\ &= \{(v_b.v_a) \,|\, \exists v : (v_b.v) \in Dep(I_1) \wedge\ (v.v_a) \in Dep(I_2)\}. \end{aligned} \tag{3.14}$$

Once the dependency sets for all instructions have been determined, the set of *varied* variables $VV(I)$ of an arbitrary instruction I can be calculated by composition $\otimes$ with the input set XI. For this purpose, the composition operator is overloaded such that it can be applied to a set of variables V and a set of dependencies $Dep(I)$, as follows

$$V \otimes Dep(I) := \{v_a \,|\, \exists v_b \in V : (v_b.v_a) \in Dep(I)\}. \tag{3.15}$$

Due to the transformation of the program into a sequence of instructions, the set of *varied* variables $VV(I_k)$ of the kth instruction can be determined iteratively by building the composition

$$VV(I_k) = XI \otimes \big(\big(\big(Dep(I_1) \otimes \ldots\big) \otimes Dep(I_{k-1})\big) \otimes Dep(I_k)\big). \tag{3.16}$$

Equation (3.16) enforces a strict evaluation order that results in a high demand of storage and computation when determining *varied* variables instruction by instruction. In particular, computing $VV(I_{k+1})$ reusing the information gathered for instruction I_k would require the following steps:

1. determine $Dep(I_{k+1})$,
2. update $Dep(I_1 - I_k) := \big(\big(Dep(I_1) \otimes \ldots\big) \otimes Dep(I_{k-1})\big) \otimes Dep(I_k)$ by adding the self dependencies $(w^*.w^*)$ for all variables w^* that have been declared between I_k and I_{k+1},
3. update XI by adding all variables that have been declared as independent between I_k and I_{k+1},
4. build the composition $Dep(I_1 - I_{k+1}) := Dep(I_1 - I_k) \otimes Dep(I_{k+1})$,
5. identify the set of *varied* variables $VV(I_{k+1})$ by computing $XI \otimes Dep(I_1 - I_{k+1})$.

To derive a more efficient algorithm for computing *varied* variables, associativity must be proved for the two defined cases of $\otimes$ that are utilized in (3.16). The associativity of the dependency composition is given, if

$$\big(Dep(I_k) \otimes Dep(I_{k+1})\big) \otimes Dep(I_{k+2}) = Dep(I_k) \otimes \big(Dep(I_{k+1}) \otimes Dep(I_{k+2})\big) \tag{3.17}$$

holds. For the left part, definition (3.14) yields

$$\begin{aligned} Dep(I_k; I_{k+1}) = \big\{\big(v_b(k).v_a(k+1)\big) \,|\, \exists v_1 :\ & \big(v_b(k).v_1\big) \in Dep(I_k) \wedge \\ & \big(v_1.v_a(k+1)\big) \in Dep(I_{k+1}) \big\}, \end{aligned}$$

$$\begin{aligned} Dep(I_k; I_{k+1}) \otimes Dep(I_{k+2}) &= \big\{\big(v_b(k).v_a(k+2)\big) \,|\, \exists v_2 :\ \big(v_b(k).v_2\big) \in Dep(I_k; I_{k+1}) \wedge \\ &\qquad \big(v_2.v_a(k+2)\big) \in Dep(I_{k+2}) \big\}, \\ &= \big\{\big(v_b(k).v_a(k+2)\big) \,|\, \exists v_1, \exists v_2 : \big(v_b(k).v_1\big)\big) \in Dep(I_k) \wedge \\ &\qquad \big(v_1.v_2\big)\big) \in Dep(I_{k+1}) \wedge \\ &\qquad \big(v_2.v_a(k+2)\big) \in Dep(I_{k+2}) \big\}. \end{aligned}$$

There, $v_b(k)$ denotes an arbitrary variable *before* and $v_a(k)$ denotes a variable *after* the kth arithmetic instruction. In the transformation process the identity $v_2 = v_a(k+1)$ has been utilized. Applying a similar strategy for the right-hand side of (3.17) yields

$$\begin{aligned} D_{k12} &:= Dep(I_{k+1}; I_{k+2}) \\ &= \big\{ \big(v_b(k+1).v_a(k+2)\big) \mid \exists v_2 : \quad \big(v_b(k+1).v_2\big) \in Dep(I_{k+1}) \wedge \\ &\qquad\qquad\qquad\qquad\qquad\quad \big(v_2.v_a(k+2)\big) \in Dep(I_{k+2}) \quad \big\}, \end{aligned}$$

$$\begin{aligned} Dep(I_k) \otimes D_{k12} &= \big\{ \big(v_b(k).v_a(k+2)\big) \quad \mid \exists v_1 : \quad \big(v_b(k).v_1\big) \quad \in Dep(I_k) \quad \wedge \\ &\qquad\qquad\qquad\qquad\qquad\quad \big(v_1.v_a(k+2)\big) \in D_{k12} \quad \big\}, \\ &= \big\{ \big(v_b(k).v_a(k+2)\big) \quad \mid \exists v_1, \exists v_2 : \big(v_b(k).v_1)\big) \quad \in Dep(I_k) \quad \wedge \\ &\qquad\qquad\qquad\qquad\qquad\quad \big(v_1.v_2)\big) \quad \in Dep(I_{k+1}) \wedge \\ &\qquad\qquad\qquad\qquad\qquad\quad \big(v_2.v_a(k+2)\big) \in Dep(I_{k+2}) \quad \big\}. \end{aligned}$$

Here, the identity $v_1 = v_b(k+1)$ has been used. As can be seen, the two transformations yield the same result and the composition of dependencies is associative, therefore.

Now, associativity must be shown for the composition of a set of variables and a composition of dependencies, i.e., that the following identity holds:

$$\big(V \otimes Dep(I_k)\big) \otimes Dep(I_{k+1}) = V \otimes \big(Dep(I_k) \otimes Dep(I_{k+1})\big). \tag{3.18}$$

Considering the left-hand side yields with (3.14) and (3.15)

$$\begin{aligned} VD_k &:= V \otimes Dep(I_k) \\ &= \big\{ v_a(k) \quad \mid \exists v_b(k) \in V : \quad \big(v_b(k).v_a(k)\big) \quad \in Dep(I_k) \quad \big\}, \end{aligned}$$

$$\begin{aligned} VD_k \otimes Dep(I_{k+1}) &= \big\{ v_a(k+1) \quad \mid \exists v_b(k+1) \in VD_k : \big(v_b(k+1).v_a(k+1)\big) \in Dep(I_{k+1}) \quad \big\}, \\ &= \big\{ v_a(k+1) \quad \mid \exists v_b(k) \in V, \exists v : \quad \big(v_b(k).v\big) \quad \in Dep(I_k) \quad \wedge \\ &\qquad\qquad\qquad\qquad\qquad\quad \big(v.v_a(k+1)\big) \quad \in Dep(I_{k+1}) \quad \big\}. \end{aligned}$$

Here, the identity $v_a(k) = v_b(k+1) = v$ has been utilized. Similarly, the right-hand side of equation (3.18) can be transformed as follows

$$\begin{aligned} D_{k01} &:= Dep(I_k) \otimes Dep(I_{k+1}) \\ &= \big\{ \big(v_b(k).v_a(k+1)\big) \quad \mid \exists v : \quad \big(v_b(k).v\big) \quad \in Dep(I_k) \quad \wedge \\ &\qquad\qquad\qquad\qquad\qquad\quad \big(v.v_a(k+1)\big) \quad \in Dep(I_{k+1}) \quad \big\}, \end{aligned}$$

$$\begin{aligned} V \otimes D_{k01} &= \big\{ v_a(k+1) \quad \mid \exists v_b(k) \in V : \quad \big(v_b(k).v_a(k+1)\big) \in D_{k01} \quad \big\}, \\ &= \big\{ v_a(k+1) \quad \mid \exists v_b(k) \in V, \exists v : \big(v_b(k).v\big) \quad \in Dep(I_k) \quad \wedge \\ &\qquad\qquad\qquad\qquad\qquad\quad \big(v.v_a(k+1)\big) \quad \in Dep(I_{k+1}) \quad \big\}. \end{aligned}$$

As can be seen, left-hand and right-hand side of (3.18) yield the same result, what means that the composition is associative. Using the associativity property of $\otimes$, equation (3.16) can be rewritten to

$$VV(I_k) = XI \otimes Dep(I_1) \otimes \ldots \otimes Dep(I_{k-1}) \otimes Dep(I_k). \tag{3.19}$$

Though it seems possible to determine *varied* variables for each instruction individually by carrying out the composition above and checking if the result of the instruction I_k belongs to $VV(I_k)$, it is not efficient. Rather, the set of *varied* variables can be adjusted by considering the activity propagation from instruction to instruction. Under this aspect, consider two successive arithmetic instructions I_k and I_{k+1} and assume that the set of *varied* variables $VV(I_k)$ has already been determined. Exploiting the associativity of $\otimes$ and utilizing (3.19) yields an adjusted formula for computing $VV(I_{k+1})$

$$\begin{aligned} VV(I_{k+1}) &= XI \otimes Dep(I_1) \otimes \ldots \otimes Dep(I_k) \otimes Dep(I_{k+1}), \\ &= VV_b(I_{k+1}) \otimes Dep(I_{k+1}). \end{aligned} \tag{3.20}$$

Thereby, $VV_b(I_{k+1}) := XI \otimes Dep(I_1) \otimes \ldots \otimes Dep(I_k)$ denotes the set of *varied* variables *before* the execution of instruction I_{k+1}. At first glance, the identity $VV_b(I_{k+1}) = VV(I_k)$ seem to hold. However, due to the facilities of the considered programming languages, new variables may be declared between the two instructions I_k and I_{k+1}. This may also include the declaration of additional independent variables xi^* that are *varied* by definition. Accordingly, $VV(I_k)$ must be updated to $VV_b(I_{k+1})$ according to

$$VV_b(I_{k+1}) = VV(I_k) \cup \{xi^*\}. \tag{3.21}$$

With (3.21), equation (3.20) can be rewritten to

$$VV(I_{k+1}) = \big(VV(I_k) \cup \{xi^*\}\big) \otimes Dep(I_{k+1}). \tag{3.22}$$

Recalling the definition (3.13) of $Dep(I)$ shows that by carrying out the composition (3.22) the only change to the set of *varied* variables can follow from the result variable of the arithmetic instruction $I_{k+1} : v = e$. The state of all other variables w will remain unchanged due to the self dependency $(w.w)$ contained in $Dep(I_{k+1})$. With the execution of instruction I_{k+1}, three options of changing the set of *varied* variables exist. They can be expressed in form of the following set differences.

1. $VV(I_{k+1}) \setminus VV_b(I_{k+1}) = \{v\}$:
   Variable v was not in the set of *varied* variables before but has been varied by I_{k+1}. Reconsidering the definitions of Dep and DP, one can see that this is only possible if at least one variable of expression e is in differential position and is also element of $VV_b(I_{k+1})$.
2. $VV_b(I_{k+1}) \setminus VV(I_{k+1}) = \{v\}$:
   Variable v is removed from the set of *varied* variables as result of I_{k+1}. This means that either a constant c is assigned or no variable in differential position is element of $VV_b(I_{k+1})$.
3. $VV(I_{k+1}) = VV_b(I_{k+1})$:
   In this case no changes occur to the set of *varied* variables. This means that the *varied* property of the result variable v is unchanged.

The second option reveals another important difference to the static activity analysis that is performed by AD-enabled compilers. Determining *varied* variables at runtime gives the chance to change the state of variables (*varied* or not) in either direction. When applying an AD-enabled compiler, similar results can only be achieved when using a hybrid static/dynamic activity analysis as suggested, e.g., in [KRE+06].

Now, reconsidering the definition of $VV_b(I_{k+1})$ and allowing for all types of program statements that have been given on page 19, an algorithm for creating and updating the set of *varied* variables at runtime can be derived. Prerequisite is a new type of AD-variable that can store the *varied* information – the state. The basic layout of the state-tracking algorithm is the following:

Evaluate the user function F instruction by instruction and do for all of them :

- if I is a control flow command:
  $\Rightarrow$ keep the *varied* state of all variables
- if I is the declaration of an independent or dependent variable:
  $\Rightarrow$ set the *varied* state of the argument variable to true
- if I is an arithmetic instruction:
  $\Rightarrow$ either set the *varied* state of the result variable to true if the *varied* state of at least one argument is true
  $\Rightarrow$ or set the *varied* state of the result variable to false if the *varied* state of all arguments is false or a constant value is assigned

The above algorithm shall be applied to determine the set of *varied* variables for the coordinate transformation example. Table 3.2 summarizes the evaluation of this set as the function is computed step by step. Thereby, variables that are added to the set of *varied* variables due to a certain instruction are set bold in the corresponding row. It can be seen that at least one argument is element of the *varied* set in the preceding row, in this case. The declaration of independent variables has been omitted for lack of space but the result appears as the set of *varied* variables before the execution of instruction one. Similarly, independents must be declared following the last instruction that is depicted in Table 3.2. By doing this, no changes to the set of *varied* variables would occur, however.

Table 3.2: Building the set of *varied* variables at runtime

operation	instruction	*varied* variables
1	$\mathbf{v_{-2}} = x_1$	$\{x_1, x_2, \mathbf{v_{-2}}\}$
2	$\mathbf{v_{-1}} = x_2$	$\{x_1, x_2, v_{-2}, \mathbf{v_{-1}}\}$
3	$v_0 = x_3$	$\{x_1, x_2, v_{-2}, v_{-1}\}$
4	$\mathbf{v_1} = v_{-1}/v_{-2}$	$\{x_1, x_2, v_{-2}, v_{-1}, \mathbf{v_1}\}$
5	$\mathbf{v_2} = v_{-2}^2$	$\{x_1, x_2, v_{-2}, v_{-1}, v_1, \mathbf{v_2}\}$
6	$\mathbf{v_3} = v_{-1}^2$	$\{x_1, x_2, v_{-2}, v_{-1}, v_1, v_2, \mathbf{v_3}\}$
7	$v_4 = v_0^2$	$\{x_1, x_2, v_{-2}, v_{-1}, v_1, v_2, v_3\}$
8	$\mathbf{v_5} = v_2 + v_3$	$\{x_1, x_2, v_{-2}, v_{-1}, v_1, v_2, v_3, \mathbf{v_5}\}$
9	$\mathbf{v_6} = v_5 + v_4$	$\{x_1, x_2, v_{-2}, v_{-1}, v_1, v_2, v_3, v_5, \mathbf{v_6}\}$
10	$\mathbf{v_7} = \sqrt{v_5}$	$\{x_1, x_2, v_{-2}, v_{-1}, v_1, v_2, v_3, v_5, v_6, \mathbf{v_7}\}$
11	$\mathbf{v_8} = v_7/v_0$	$\{x_1, x_2, v_{-2}, v_{-1}, v_1, v_2, v_3, v_5, v_6, v_7, \mathbf{v_8}\}$
12	$\mathbf{v_9} = \sqrt{v_6}$	$\{x_1, x_2, v_{-2}, v_{-1}, v_1, v_2, v_3, v_5, v_6, v_7, v_8, \mathbf{v_9}\}$
13	$\mathbf{v_{10}} = \arctan(v_8)$	$\{x_1, x_2, v_{-2}, v_{-1}, v_1, v_2, v_3, v_5, v_6, v_7, v_8, v_9, \mathbf{v_{10}}\}$
14	$\mathbf{v_{11}} = \arctan(v_1)$	$\{x_1, x_2, v_{-2}, v_{-1}, v_1, v_2, v_3, v_5, v_6, v_7, v_8, v_9, v_{10}, \mathbf{v_{11}}\}$
15	$\mathbf{y_1} = v_9$	$\{x_1, x_2, v_{-2}, v_{-1}, v_1, v_2, v_3, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, \mathbf{y_1}\}$
16	$\mathbf{y_2} = v_{10}$	$\{x_1, x_2, v_{-2}, v_{-1}, v_1, v_2, v_3, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, y_1, \mathbf{y_2}\}$
17	$\mathbf{y_3} = v_{11}$	$\{x_1, x_2, v_{-2}, v_{-1}, v_1, v_2, v_3, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, y_1, y_2, \mathbf{y_3}\}$

Comparing the result of the state-tracking procedure with the compiler-aided augmentation process for the coordinate transformation example verifies the theoretical assumptions. Both, the set of *varied* variables and the set of variables to be replaced with an augmented data type are identical. In contrast to the compiler-aided approach, the state-tracking technique can benefit from the fact that only one compile sweep is necessary and all variables might be redeclared with the state-tracking data type. Furthermore, a smaller set of *varied* variables can be achieved due to the possibility of resetting the *varied* state of a variable at runtime.

With the state-tracking technique, an efficient way of determining *varied* variables has been derived for the application within the context of operator overloading. The integration of this concept into the overall taping procedure as well as the resulting challenges are described in the following subsection.

3.2.3 Adapted taping procedure

In subsection 2.2.2, the basic layout for the creation of an internal function representation was discussed. A small example can be found on page 19. However, the underlying approach had to be adjusted to properly reflect the state-tracking technique. In addition to the creation of the internal function representation that mainly was discussed up to this point, another aspect of the derivation process must be considered. Generally, values of certain function variables may be required for the calculation of the derivatives. This is usually no problem when applying the forward mode of AD. There, each instruction of the code can be combined with an appropriate derivative instruction. By carefully choosing the sequence of these two

instructions, it can be ensured that the correct function values are used for the derivation. In particular, it must be guaranteed that the argument values of an operation are used in its derivation before the assignment of the result possibly overwrites one of them.

A more challenging task has to be solved when applying the reverse mode of AD. Due to the reversal of the program flow that succeeds the function evaluation, function values that are necessary for the derivation of instructions might have been overwritten, meanwhile. This issue can be solved by either recalculating the required values within the derivation process or by storing the relevant data during the function evaluation and restoring them within the derivative calculations, accordingly. The latter procedure is often called *recording*. Using tape-based operator overloading, the recalculation of function values is in general only possible by reevaluating the created internal representation up to a desired point. The computational effort for recalculating all function values needed in the derivation process this way is unacceptable, in general. Therefore, the recording approach is applied for all overloading based tools known to the author. Utilizing this technique, the value of each variable that is overwritten in an assignment is stored onto a stack-like storage prior to the change of the value. Handling the assignments within the derivation procedure then also comprises the restoring of the overwritten values, accordingly.

Considering the recording of overwritten values under the aspect of state-tracking techniques reveals that this process must also be adapted. Up to now, the commonly applied technique enforces to backup the value of the result variable of an operation prior to the assignment, if the result variable is of the special AD data type. Only in this case, an internal representation of the operation is created. In the reevaluation process, the overwritten value of the result variable is restored, accordingly. This way, the handling of overwritten values is implicitly represented. To keep this one-to-one correlation, this approach must be reduced onto those operations whose result variables have the *varied* state after the execution.

Creating the internal function representation

Taking the described requirements into account, an appropriate AD data type is proposed as depicted in Figure 3.8. There, C++ notation is used again. The new type must at least allow to reflect the state of the belonging variable, i.e., *varied* or *not varied*. For the AD data type depicted in Figure 3.8, this information is represented by a discrete variable. The new data type must also afford the creation of an internal function representation as well as the computation of the function values. As a representative of the overloaded operators, the principal layout of the copy assignment has been given.

Basically, most overloaded operators and intrinsic functions follow a three phase approach. Within the first step, it is determined for the result of the instruction, if it belongs to the set of *varied* variables. This can be achieved by computing the *varied* state of the result as disjunction over the *varied* state of all arguments. The determination of the *varied* state can be omitted for the class of comparison operators. There, the result is of boolean type, typically. Once the state of the result is known, an internal representation of the instruction can be created and the results value prior to the instruction may be recorded. Due to the state-tracking, these two tasks are only performed if at least one argument belongs to the set of *varied* variables. Finally, the standard intention of the instruction is addressed by evaluating the operation or intrinsic function using the value components of the variables.

Within the taping process that is represented by the call to “tapeOperation” in Figure 3.8, the state information of the argument variables must be taken into account once more. Simply using the ID to represent a variable does no longer guarantee the correctness of the computed values. In fact, it is possible that a variable has not been added to the set of active variables but enters the computation, nevertheless. When reevaluating the generated internal representation, the value of this variable is undefined as not all instructions might have been taped. Possible effects can be seen from the following example.

For the function $c = f(a, b) = a * b^2 + 0.5 * b$, derivatives shall be computed with respect to a at the base point $(a, b) = (0.5, 2.0)$. An appropriate source code is given in the left part of Figure 3.9, the created

```
class ADtype {
    protected:
        Type value;
        Type ID;
        bool varied;

    public:
        ADtype operator = (const ADtype &arg) {
            ADtype result;

            varied = arg.varied;

            if (varied) {
                tapeOperation(AGN, ID, arg.ID);
                push();
            }

            result.value = value * arg.value;
            return result;
        }
};
```

Figure 3.8: State-tracking AD-type including taping

internal representation is depicted in the middle part and a value history is presented in the right part. Line 1 comprises the declaration of the independent variable a as well as the initial value assignment for all variables. Thereby, α, β and γ denote arbitrary values that result from memory reusage. All five variables are of the new state-tracking data type and variables with enabled *varied* state are set bold in the source code. Within Figure 3.9, the internal representation for the declaration of the independent variable a as well as line 6, corresponding to the declaration of the dependent variable c, have been omitted for the sake of simplicity. For the internal function representation, the new operation code ADD - addition is introduced.

				a	b	t_1	t_2	c
1				0.5	2.0	α	β	γ
2	$t_1 = b * b$	$= 4.0$	—	0.5	2.0	4.0	β	γ
3	$\mathbf{t_2} = \mathbf{a} * t_1$	$= 2.0$	(MUL, t_2, a, t_1)	0.5	2.0	4.0	2.0	γ
4	$t_1 = 0.5 * b$	$= 1.0$	—	0.5	2.0	1.0	2.0	γ
5	$\mathbf{c} = \mathbf{t_2} + t_1$	$= 3.0$	(ADD, c, t_2, t_1)	0.5	2.0	1.0	2.0	3.0

Figure 3.9: Incorrect internal representation

Reevaluating the function at the given base point based using the created internal representation yields the wrong result as can be seen from Figure 3.10. Again, the Greek letters denote arbitrary values that result from memory reusage. Often, when reevaluating an internal function representation, the new base point can only be chosen in terms of the independent variables. This basically depends on the facilities provided by the applied AD-tool. When using the state-tracking technique this is no longer arbitrary as the internal representation strongly depends on the *varied* state of the variables. It is very likely that even by using the correct initial value of an "unvaried" variable, instructions based on this variable may use an incorrect value. This is due to the fact that not all instructions changing the value of the variable might have been included into the internal representation of the function. In the given example, it does not matter if the correct value (2.0) of b is been supplied or not. Due to the instructions that have not been included as a result of the state analysis, the value of t_1 is arbitrary and the overall result c is not correct, accordingly.

		a	b	t_1	t_2	c
1		0.5	δ	ϵ	ζ	η
3	$t_2 = a * t_1 \quad = 0.5 * \epsilon$	0.5	δ	ϵ	$0.5 * \epsilon$	η
5	$c = t_2 + t_1 \quad = 1.5 * \epsilon$	0.5	δ	ϵ	$0.5 * \epsilon$	$1.5 * \epsilon$

Figure 3.10: Reevaluation of the function based on an incorrect internal representation

As for the reevaluation of the internal representation depicted in Figure 3.9, the application of the reverse mode of AD based on it also yields wrong results. Assuming that the recording of overwritten values is accomplished during the taping process, the stack would hold β (after line 3) and β, γ (after line 5), respectively. These values are used in the reverse sweep, in which the restoring of a value is depicted by the *pop*-function in Figure 3.11. Line 6 that has been omitted in the figures before, is now included and can be treated as an initialization step. Therein, the adjoint values for all variables are set to zero with the exception of $\bar{c}$, which is set to 1.0 according to its state as dependent variable. In return, the adjoint statements for line 1 are left out.

Calculating the required derivate analytically yields the formula $\partial f / \partial a = b^2$. Evaluated at the given base point, the correct result is $\bar{a} = \partial f / \partial a = 4.0$ which is different from the value depicted in Figure 3.11. The main problem in the given example is caused by the incorrect value of t_1 in the computation of $\bar{a}$ in line 3.2. On a first glance, this seems to be caused by a missing push/pop pair for t_1. By including these functions at the correct position in the taping process, i.e., the call to push before evaluating line 4 in Figure 3.9 and the call to pop before line 3.1 in Figure 3.11, the correct value would be computed. However, this approach would fail if the reverse sweep is performed succeeding a reevaluation of the function based on the created internal representation. As can be seen from Figure 3.10, the call to the push function cannot be placed and, furthermore, the wrong value ϵ would be stored for t_1 in any case.

		a	$\bar{a}$	b	$\bar{b}$	t_1	$\bar{t}_1$	t_2	$\bar{t}_2$	c	$\bar{c}$
6		0.5	0.0	2.0	0.0	1.0	0.0	2.0	0.0	3.0	1.0
5.1	$\text{pop}(c)$	0.5	0.0	2.0	0.0	1.0	0.0	2.0	0.0	γ	1.0
5.2	$\bar{t}_2 \mathrel{+}= \bar{c} \quad = 1.0$	0.5	0.0	2.0	0.0	1.0	0.0	2.0	1.0	γ	1.0
5.3	$\bar{t}_1 \mathrel{+}= \bar{c} \quad = 1.0$	0.5	0.0	2.0	0.0	1.0	1.0	2.0	1.0	γ	1.0
5.4	$\bar{c} \quad = 0.0$	0.5	0.0	2.0	0.0	1.0	1.0	2.0	1.0	γ	0.0
3.1	$\text{pop}(t_2)$	0.5	0.0	2.0	0.0	1.0	1.0	β	1.0	γ	0.0
3.2	$\bar{a} \mathrel{+}= \bar{t}_2 * t_1 \quad = 1.0$	0.5	1.0	2.0	0.0	1.0	1.0	β	1.0	γ	0.0
3.3	$\bar{t}_1 \mathrel{+}= \bar{t}_2 * a \quad = 1.5$	0.5	1.0	2.0	0.0	1.0	1.5	β	1.0	γ	0.0
3.4	$\bar{t}_2 \quad = 0.0$	0.5	1.0	2.0	0.0	1.0	1.5	β	0.0	γ	0.0

Figure 3.11: Reverse propagation based on an incorrect internal representation

Turning back to the state-tracking, one can see that the variable t_1 does not belong to the *varied* set for all critical instructions. This means that this variable does not depend on any independent variable at the considered points, per definition. In return, derivative values do not need to be computed or are trivially zero. The only way such variables enter the function as well as the derivative calculation is by means of their function values. Due to the missing connection to the independent variables, the value of these variables can be treated as constants. Accounting for this attribute in the taping process, variables without the *varied* state must be represented by their values rather than by their ID. Applying this result in the taping process for the function $c = f(a, b)$ yields the internal function representation that is given in Figure 3.12.

With this corrected version of the internal representation, the reevaluation of the function as well as the computation of derivatives yield the expected results. The two sweeps are depicted in Figure 3.13, the

```
3   (MUL, t2, a, 4.0)
5   (ADD,  c, t2, 1.0)
```

Figure 3.12: Corrected internal function representation

reevaluation in the upper part and the derivative calculations in the lower part. Thereby, the same notation is used as in the figures above. The recording of values that are overwritten is carried out during the function reevaluation. Accordingly, the stack values are ζ after line 3 and ζ, η after line 5, respectively. An additional benefit of the substitution of variables by constants becomes obvious when comparing the computational effort of the reverse computations represented by the Figures 3.11 and 3.13. Due to the substitution, some nonlinear dependencies have been transformed into linear dependencies. The latter can be handled with reduced effort based on the inherent mathematics.

		a	b	t_1	t_2	c
1		0.5	δ	ϵ	ζ	η
3	$t_2 = a * 4.0 \quad = 2.0$	0.5	δ	ϵ	2.0	η
5	$c = t_2 + 1.0 \quad = 3.0$	0.5	δ	ϵ	2.0	3.0

		a	$\bar{a}$	b	$\bar{b}$	t_1	$\bar{t}_1$	t_2	$\bar{t}_2$	c	$\bar{c}$
6		0.5	0.0	δ	0.0	ϵ	0.0	2.0	0.0	3.0	1.0
5.1	pop(c)	0.5	0.0	δ	0.0	ϵ	0.0	2.0	0.0	η	1.0
5.2	$\bar{t}_2 \mathrel{+}= \bar{c} \quad = 1.0$	0.5	0.0	δ	0.0	ϵ	0.0	2.0	1.0	η	1.0
5.4	$\bar{c} \quad = 0.0$	0.5	0.0	δ	0.0	ϵ	0.0	2.0	1.0	η	0.0
3.1	pop(t_2)	0.5	0.0	δ	0.0	ϵ	0.0	ζ	1.0	η	0.0
3.2	$\bar{a} \mathrel{+}= \bar{t}_2 * 4.0 \quad = 4.0$	0.5	4.0	δ	0.0	ϵ	0.0	ζ	1.0	η	0.0
3.4	$\bar{t}_2 \quad = 0.0$	0.5	4.0	δ	0.0	ϵ	0.0	ζ	0.0	η	0.0

Figure 3.13: Computation of function and derivative values based on the corrected internal representation

A drawback of argument type replacement as described above is the increased complexity that must be coped with when creating the internal function representation. Instead of only one constellation, i.e., all variables involved in a specific instruction own a derivative counterpart, several different situations must be handled. The more arguments are used by an instruction, the more complex this process is. In terms of operator overloading, again the three basic types of instructions must be considered that have been discussed in subsection 3.1, before. Each type requires adaptations that are given below.

- **Declaration of independent or dependent variables**
  This type of instruction can be handled with minimal overhead. An internal representation of these instructions is always created, independent if the considered variable is *varied* or not. Within the created representation, the variables are never replaced by their numerical value. Special attention must be paid if the variable that is marked as dependent does not have the *varied* attribute. In that special case, an additional instruction must be included into the internal function representation. Preceding the declaration as dependent variable, an assignment instruction must be taped that induces the correct numerical value of the variable. Correspondingly, the derivative value is set to zero within the derivation process as result of the assignment of a constant value. A warning message may be issued to inform the user about this state, which is probably unintended.
- **Arithmetic operations**
  The taping effort for this type of instruction is tightly coupled to the number of arguments. For unary operations and intrinsic functions the *varied* state of the argument decides if the instruction

is appended to the internal function representation. If the argument belongs to the set of *varied* variables, an internal representation is created that is based on the IDs of argument and result variable. Otherwise the instruction is ignored in terms of the taping process. A substitution of variables by their numerical value is never enforced.

Compared to the unary case, binary operations and intrinsic functions need a relatively high number of additional checks. As described before, an internal representation gets only created if the result of the instruction has the *varied* attribute. This is set to true if at least one of the arguments has this attribute. When creating the internal representation, it must be determined which variables are *varied* and where to apply the value based replacement. In principle, three cases have to be distinguished, i.e., the two arguments have the *varied* state or only one of them. Each of these cases has to be handled separately.

- **Comparison instructions**
  A suitable internal representation of comparison instructions is the key for a reliable detection of changes in the control flow when reevaluating the function at a different base point. To guarantee the correctness in the sense above, *varied* state information has to be taken into account when applying the state-tracking technique to comparison instructions. This includes to replace the ID of each *non-varied* argument of the comparison by its numerical value computed so far. If none of the involved variables is of *varied* state, no internal representation of the instruction must be created. The latter would consist of the comparison of two constant values that always yields the same results. By not creating this representation, a runtime improvement can be achieved when reevaluating the tape.

With the additional complexity that results from the state-tracking technique, the internal implementation of the overloaded operators and functions holds an significant meaning for the efficiency of the taping process. However, considering the creation of the internal function representation as part of a larger application diminishes its importance in the global view. There, runtime drawbacks that arise from the more complex taping procedure will be compensated by a more efficient derivative calculation, most probably. However, minimizing the preliminary steps that are necessary to afford the derivative calculations is one of the key ingredients to keep the overall runtime as small as possible. Accordingly, it must be ensured that the created internal representation is evaluated as seldom as possible. Challenges that must be handled in this context are discussed in the following subsections.

Three phase approach for reverse mode AD

As for the forward mode, an internal representation of the function must be created that can be used to reverse the program flow in the derivation phase. Assuming that memory is reused for reducing the overall storage requirements, the loss of important information due to overwriting must be taken into account. Hence, AD-tools keep track of overwritten values in the forward sweep and restore them in the reverse sweep. This process is called recording as described in Subsection 3.2.3. Integrating the recording procedure into the taping phase that utilizes a state-tracking technique, inflicts the potential of storing and restoring wrong values. The reason is found in the specific point of the taping process where the recording takes place as illustrated by the following small example.

EXAMPLE 3.2

	Function	*Stack*	*Tape*
1	$t = a * b$	$push\big(value(t)\big)$	$(MUL,\ t, a, b)$
2	$a = 1.0$	—	—
3	$\{$use $a\}$	*	*
4	$a = c$	$push\big(value(a)\big)$	$(AGN,\ a, c)$

There, the variables a, b and t in line 1 as well as c and a in line 4 belong to the set of *varied* variables. In fact, a is removed from the set of *varied* variables as result of instruction 2 and is reinserted as result of line 4. Line 3 may represent a larger block of code, but it is assumed that the numerical value of a is not changed. Accordingly, the value of a is only effected by the instructions 1 and 4. As result of the latter, the numerical value 1.0 is pushed onto the tape. The internal representation is extended according to line 1 and 4 as well as block 3, depending on the involved instructions. Based on the created internal representation, the following sequence of instructions is created within the reverse sweep.

EXAMPLE 3.3

Derivation code

$$
\begin{array}{lll}
4.1 & pop\big(value(a)\big) & \Leftarrow \quad a = 1.0 \\
4.2 & \bar{c} \mathrel{+}= \bar{a} & \\
4.3 & \bar{a} \quad = 0.0 & \\
3 & \bar{*} & \\
1.1 & pop\big(value(t)\big) & \\
1.2 & \bar{a} \mathrel{+}= \bar{t} * b & \\
1.3 & \bar{b} \mathrel{+}= \bar{t} * a & \Leftarrow \quad a = 1.0 \textit{ is used here!} \\
1.4 & \bar{t} \quad = 0.0 &
\end{array}
$$

As can be seen, the wrong value of a is used in the computation of $\bar{b}$ in line 1.3. This results from the unnoticed overwrite of a in line 2 of the original function. The value that has been used in the computation of t and that is also necessary to compute the correct derivative $\bar{b}$ has been lost. An obvious workaround is to create the stack of overwritten values based on the internal function representation. There, the critical assignment to a does not happen and the correct values can be restored in the derivation part. The corresponding basic layout of the overall derivation procedure is given below.

1. An internal representation of the evaluated function gets created. Thereby, no recording of values needs to be performed.
2. The stack that holds all values of variables overwritten within the function evaluation is created based on the internal function representation.
3. Derivative values are computed based on the reverse mode of AD using the internal representation and the stack that have been generated before.

Employing this three phase approach, an increase of the overall runtime is very likely. In practice, the application of AD should always be as efficient as possible since most of the handled problems are time critical. For this reason, a technique is proposed that allows to combine step one and two of the procedure above without loosing the correctness of the computed values.

Two phase approach for reverse mode AD

The main reason for creating the internal function representation and the value stack separately was the unnoticed overwrite of variable values in the context of state-tracking. To combine the two tasks within one phase, it must be ensured that the variable values recognized by the recording process do not change between two consecutive instructions regarding the internal function representation. Computations interfering with two such instructions need to operate on a different value. Accordingly, the state-tracking data type must allow to access two different values that both belong to the represented variable. The AD data type can be adjusted as given below.

```
class ADtype {
    protected:
        Type value_varied;
        Type value_unvaried;
        Type ID;
        bool varied;

    public:
        ⋮
};
```

Now, the value of a variable declared of this type depends on the *varied* property. This means, the component `value_unvaried` is used for the numerical calculations as long as the *varied* state is set to *false* and `value_varied` otherwise. Creating the stack of overwritten values within the taping phase now always relies on the `value_varied` component of the variables. Every time an assignment to a variable with enabled *varied* property occurs, the `value_varied` is pushed onto the stack. Other assignments, i.e., to an *unvaried* result variable, change the `value_unvaried` component only and the overwritten value is not pushed onto the stack. This ensures that the recording process exactly uses the value sequence that would be created when reevaluating the internal representation at the same base point. Applying this technique to perform the recording within the taping phase for Example 3.2, yields the following result.

EXAMPLE 3.4

		Values				
		t		a		
	Function	*unv.*	*var.*	*unv.*	*var.*	*Stack*
		α	β	γ	2.0	
1	$t = a * b$	α	6.0	γ	2.0	*push*(β)
2	$a = 1.0$	α	6.0	1.0	2.0	—
3	$\{$*use* $a\}$	α	6.0	1.0	2.0	$*$
4	$a = c$	α	6.0	1.0	4.0	*push*(2.0)

For the sake of clearness, the specific values $a = 2.0$, $b = 3.0$ and $c = 4.0$ are used for the *varied* versions of the variables. Using the two values for each variable of the AD data type, the correct values are pushed onto the stack, i.e., β for t in line 1 and 2.0 for a in line 4. For this reason, the two separate phases for generating the internal function representation and the stack of overwritten values can be combined into one phase. The overall process of computing derivatives based on the reverse mode of AD can be completed in a two phase approach, this way. This reduction in the number of program phases is, however, not free of charge. Due to the two values of each variable, the memory consumption of the program is increased and additional costs must be taken into account for accessing the values of the variables based on their *varied* state. All in all, this is less expensive than the additional tape evaluation necessary in the three phase approach that has been described above.

Alternative approach used in CppAD

The two techniques, i.e., the three and the two phase approach that have been discussed above were developed to provide the original values of overwritten function variables in the reverse sweep. A different approach is used in the AD-tool CppAD [Bel07]. There, a strategy is utilized that avoids the overwriting of *Variable* values completely. *Variables* in terms of CppAD belong to the set of *varied* variables whereas

Parameters do not. In the internal representation of the function, *Variables* are represented by their IDs whereas *Parameters* are substituted by their values. The overwriting of values is avoided for the internal representation by assigning an unused ID to the result of every operation. Consider Example 3.5 for the basic layout of the approach. There, the arithmetic operation $a = a * b$ is handled by the usage of a compiler generated temporary variable t.

EXAMPLE 3.5

Operation	*Int. representation*	*Calculations*
$t = a * b$	$(\mathit{MUL}, 3, 1, 2)$	$t.value = a.value * b.value$ $t.id = 3$
$a = t$	$-$	$a.value = t.value$ $a.id = t.id$

The technique for creating the internal function representation that this thesis is based on would result in two tape entries: $(\text{MUL}, 3, 1, 2)$ would be written for $t = a * b$ and $(\text{AGN}, 1, 3)$ for $a = t$. These two entries can be merged to $(\text{MUL}, 1, 1, 2)$. Comparing this result to the internal representation created in Example 3.5 reveals the major difference. Though the represented operation requires the overwriting of the function value a, it is transformed by CppAD into an equivalent form that avoids the overwriting. This also prevents the problem of restoring the wrong function values in the reverse sweep and allows to avoid the explicit stack handling. In terms of the function values, the memory consumption of the two approaches is nevertheless the same when reevaluating the created internal function representation for preparing the reverse mode differentiation. Using CppAD, function values that would be overwritten in the original function are preserved within the field of values that gets filled when reevaluating the internal representation. In the overwriting approach, these values are stored and restored using the stack.

Though the CppAD approach allows to reduce the complexity of the implementation, it also has significant disadvantages. Most important, all values that have been computed during the evaluation of the function are hold in a large vector of a specific size, denoted by s. This includes the values of variables that would have been overwritten in the original function. However, these values are only used for reverse mode differentiation and are unnecessarily stored otherwise. Furthermore, memory for storing the derivative information must be provided in the size of s or a multiple thereof, depending on the degree of the derivation. This again increases the overall memory requirement significantly. The necessary storage sizes are depicted in Figure 3.14. Overall, the approach applied in the AD-tool CppAD is an interesting approach to handle the overwriting problem at the cost of a considerably higher memory demand.

So far, all optimizations where applied in terms of the forward evaluation of the function. Reconsidering Figure 3.7 reveals an important fact. In addition to the forward propagation, a reverse analysis of the created internal representation is necessary to create the set of active variables according to formula (3.12). Appropriate facilities that allow this kind of optimization are discussed in the following subsection.

3.2.4 Reverse tape optimization

In contrast to the source-to-source approach where the complete analysis is performed at compile time, operator overloading basically works at runtime. Consequently, this is true for the second analysis phase, i.e., the propagation of activity information from the dependent towards the independent variables, too. The reverse propagation of activity information must be implemented carefully to prevent a significant increase in runtime for the average application. In contrast to the forward analysis where the internal representation of the function, i.e., the tape, could be adjusted during its creation, two different approaches for handling the tape may be pursued in the reverse analysis phase. On the one hand, the tape may not be changed in the reverse sweep. Then, the propagation of activity information must be performed each time

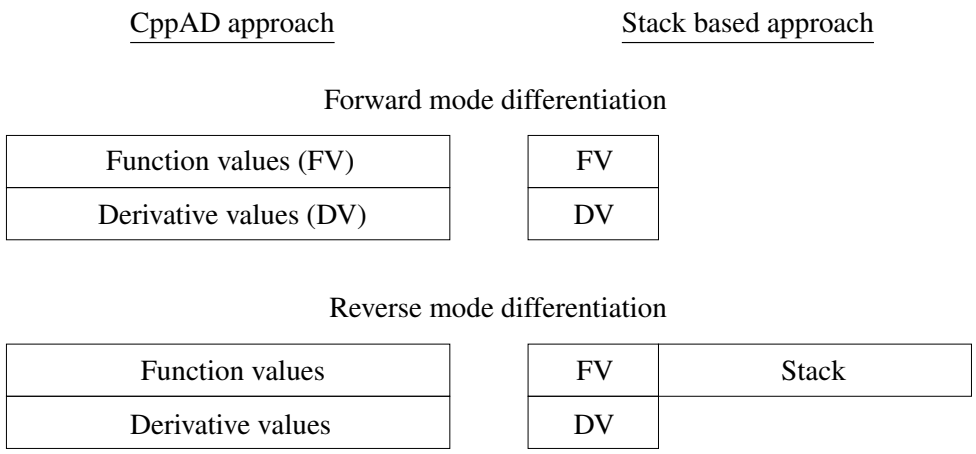


Figure 3.14: Memory requirements for the CppAD and the stack based approach

the tape is used but the correctness of the representation is obviously retained and effort for restructuring the tape can be avoided. This approach, which is useful for applications evaluating the tape only once or a few times, is considered first. On the other hand, the tape may be rewritten during the first usage in a reverse sweep. Additional effort must then be invested for restructuring the tape and maintaining the correctness of the tape for the function evaluation and forward mode differentiation, respectively. However, once the tape has been rewritten correctly, it may be reevaluated in any mode without the need to apply activity analysis. This approach is especially useful whenever the tape is reevaluated several times, e.g., when computing derivatives for a function containing a fixed point iteration.

For deriving a general procedure that allows a reverse optimization without altering the tape, the internal activity analysis of an AD-enabled compiler shall be reconsidered. Activity information is propagated in two directions at compile time, overall yielding the set of variables that derivatives have to be computed for. In the forward propagation the set VV of *varied* variables has been determined according to

$$VV = \left\{ v \,|\, \exists x \in XI : x \prec^{+} v \right\}.$$

This process has been adapted in such a way that it could be applied within the taping process for the operator overloading approach. Hence, all variables represented by an ID in the internal function representation belong to the set VV if the state-tracking technique has been utilized. Another phase of the activity analysis performed by AD-enabled compilers determines the set UV of *useful* variables according to

$$UV := \left\{ v \,|\, \exists y \in YD : v \prec^{+} y \right\}.$$

Finally, the set AV of active variables is constructed according to (3.12) by building the intersection of VV and UV. Then, derivative code is generated only for those variables that belong to the set of active variables.

So far, only the set VV of *varied* variables could be determined by operator overloading based AD-tools when applying the state-tracking technique as described in Subsection 3.2.2. The internal function representation that is created within the taping phase then only holds instructions for these variables. As derivative values are computed based on the created tape it seems reasonable to develop a technique to further optimize the derivation process by determining the set of active variables. This can be achieved by reducing the set of *varied* variables in a reverse state-tracking phase that determines *useful* variables. Due to the utilization of the *varied* information within this process, the set of active variables is the final result.

The basic strategy of the reverse state-tracking approach is equivalent to the state-tracking technique applied within the taping process. However, depending on the applied AD-tool, the *useful* property of

each variable may be stored as part of the variable or within a separate field. From the theory results that the *useful* property is set for all dependent variables, initially. This perfectly matches the reverse direction of the tape evaluation. Specific rules for the state propagation can again be derived from the compiler theory. For reflecting the reverse propagation, the composition operator $\otimes$ is overloaded once more:

$$Dep(I) \otimes V = \left\{ v_b \,|\, \exists v_a \in V : (v_b.v_a) \in Dep(I) \right\}. \tag{3.23}$$

Thereby, the notation of variables *before* and *after* an instruction used for the forward case is retained. Due to the special structure of the internal representation, which is based on individual instructions, the *useful* property of the variables is determined step by step. Accordingly, the set of *useful* variables can be described as a sequence of compositions of dependency sets and the set of dependent variables by utilizing (3.17) and (3.23). The set $UV(I_k)$ for an instruction I_k is given by the formula

$$\begin{aligned} UV(I_k) &= Dep(I_k) \otimes \ldots \otimes Dep(I_{l-1}) \otimes Dep(I_l) \otimes YD \\ &= \bigotimes_{i=k}^{l} Dep(I_i) \otimes YD \\ &= Dep(I_k - I_l) \otimes YD. \end{aligned} \tag{3.24}$$

Thereby, I_l denotes the last instruction that has been added to the internal representation and l is the overall number of instructions recorded within the tape. (3.24) already exploits associativity that can be proven by utilizing (3.23) and (3.17). The basic layout of the proof is equivalent to the proof of (3.18) and is therefore not given, here. Utilizing the associativity of (3.24), the sequence may be evaluated from right to left. This enables the analysis to be done in reverse order, e.g., within the reverse mode phase of AD.

The evolution of the set of *useful* variables can again be put down to the differences of the sets UV for two consecutive instructions I_{k-1} and I_k. In particular, the result of the preceding instruction in the reverse evaluation order, i.e., I_k, can be reused for the computation of $UV(I_{k-1})$, yielding

$$\begin{aligned} UV(I_{k-1}) &= Dep(I_{k-1}) \otimes Dep(I_k - I_l) \otimes YD, \\ &= Dep(I_{k-1}) \otimes UV(I_k). \end{aligned} \tag{3.25}$$

As can be seen, the differences between the two sets of *useful* variables $UV(I_k)$ and $UV(I_{k-1})$ can only result from the dependency structure of the inspected instruction. Reconsidering definition (3.13), general rules for the reverse state-tracking can be derived. Due to the reduced internal representation that has been created in the taping phase the possible range of expressions has been reduced, in particular, e can no longer be a constant. Furthermore, all taped arithmetic instructions have the result and at least one argument of variable type. All known variables represented within the tape but not involved in the considered instruction keep their *useful* state. This is a result of the self-dependency of all variables with exception of the result v in the forward consideration. A more complex handling is applied to all members of $DP(e)$. These variables keep their *useful* state if v does not belong to $UV(I_k)$ or the property is set for all of them if v belongs to $UV(I_k)$. Finally, the former result variable v is removed from the set of *useful* variables if it does not belong to $DP(e)$, itself.

Within the process of reverse state-propagation, the necessary derivative instructions are determined, in addition. From the compiler based activity analysis one can see that derivative calculations are only performed if the result of an instruction belongs to both the set of *varied* and the set of *useful* variables. This approach can now be reused within the reverse state-tracking phase. Here, a variable that belongs to the set of *useful* variables inevitably also belongs the set of *varied* variables. Otherwise it would have been represented by a constant within the tape. Hence, a variable with enabled *useful* property automatically belongs to the set of active variables. If this applies to the result of a represented instruction, derivative calculations are to be performed. All in all, a general algorithm for updating the set of *useful* variables can be generated.

Reevaluate the created internal representation in reverse order instruction by instruction and do for all of them:

- if I is a control flow command:
  - $\Rightarrow$ keep the *useful* state of all variables
  - $\Rightarrow$ do not perform any derivative calculation
- if I is the declaration of a dependent or independent variable:
  - $\Rightarrow$ set the *useful* state of the argument variable to true
  - $\Rightarrow$ perform appropriate derivative calculations depending on the internals of the applied AD-tool
- if I is an arithmetic instruction:
  - $\Rightarrow$ take a backup b of the *useful* state of the result variable v and set the property to false for v
  - $\Rightarrow$ restore the function value of the instructions result variable
  - $\Rightarrow$ if b is true:
    - $*$ set the *useful* property to true for all elements of $DP(e)$
    - $*$ perform derivative calculations according to the represented instruction

Reverse state-tracking applied to the coordinate transformation example

For illustrating the reverse state-tracking, the coordinate transformation example as given in Figure 2.2 shall be reused. First, before addressing the reverse propagation of activity information, the optimization based on the forward state-tracking is to be performed. Again, derivatives shall be computed for y_1 with respect to x_1 and x_2. The resulting internal function representation is depicted in the left part of Figure 3.15. Besides the known operation codes, DIV, SQRT and ATAN are introduced to represent the division operator, the square root function and the arc tangent function. For reasons of clearness, the internal representation is given in reverse order.

The next step in optimizing the derivative calculations is the application of the reverse state-tracking to the created tape. Initially, the set of *useful* variables contains the dependent variable y_1, only. This information is then propagated in reverse order according to the derived reverse state-tracking algorithm. In Figure 3.15, this process is depicted schematically.

As described before, derivative instructions are only to be performed for those instructions of the tape whose result is in the current set of *useful* variables. In Figure 3.15, these variables have been set bold. Since no variable overwrites occur in the function, all assignments of the value 0.0 to the active variable at the end of the derivative handling for an instruction have been omitted. As can be seen from Figure 3.15, the state-tracking considerably reduces the number of performed derivative instructions. Combining the individual derivative operations and assigning $\bar{y}_1 = 1.0$ yields the desired gradient

$$\begin{aligned}\left(\frac{\partial y_1}{\partial x_1}, \frac{\partial y_1}{\partial x_2}\right) &= \left(\frac{v_{-2}}{\sqrt{v_6}}, \frac{v_{-1}}{\sqrt{v_6}}\right), \\ &= \left(\frac{x_1}{\sqrt{x_1^2 + x_2^2 + x_3^2}}, \frac{x_2}{\sqrt{x_1^2 + x_2^2 + x_3^2}}\right).\end{aligned}$$

Consequently, a nearly optimal result could be achieved for the considered coordinate transformation example.

Tape (reversed)	*Useful* variables	Derivative instructions
	$\{y_1\}$	—
(AGN , y_3, v_{11})	$\{y_1\}$	—
(AGN , y_2, v_{10})	$\{y_1\}$	—
(AGN , $\mathbf{y_1}$, v_9)	$\{v_9\}$	$\bar{v}_9 \mathrel{+}= \bar{y}_1$
(ATAN, v_{11}, v_1)	$\{v_9\}$	—
(ATAN, v_{10}, v_8)	$\{v_9\}$	—
(SQRT, $\mathbf{v_9}$, v_6)	$\{v_6\}$	$\bar{v}_6 \mathrel{+}= \bar{v}_9 * \frac{1.0}{2.0*\sqrt{v_6}}$
(DIV , v_8, v_7, v_0)	$\{v_6\}$	—
(SQRT, v_7, v_5)	$\{v_6\}$	—
(ADD , $\mathbf{v_6}$, v_5, value(v_4))	$\{v_5\}$	$\bar{v}_5 \mathrel{+}= \bar{v}_6$
(ADD , $\mathbf{v_5}$, v_2, v_3)	$\{v_2, v_3\}$	$\bar{v}_2 \mathrel{+}= \bar{v}_5, \quad \bar{v}_3 \mathrel{+}= \bar{v}_5$
(MUL , $\mathbf{v_3}$, v_{-1}, v_{-1})	$\{v_2, v_{-1}\}$	$\bar{v}_{-1} \mathrel{+}= \bar{v}_3 * v_{-1}, \quad \bar{v}_{-1} \mathrel{+}= \bar{v}_3 * v_{-1}$
(MUL , $\mathbf{v_2}$, v_{-2}, v_{-2})	$\{v_{-2}, v_{-1}\}$	$\bar{v}_{-2} \mathrel{+}= \bar{v}_2 * v_{-2}, \quad \bar{v}_{-2} \mathrel{+}= \bar{v}_2 * v_{-2}$
(DIV , v_1, v_{-1}, v_{-2})	$\{v_{-2}, v_{-1}\}$	—
(AGN , $\mathbf{v_{-1}}$, x_2)	$\{v_{-2}, x_2\}$	$\bar{x}_2 \mathrel{+}= \bar{v}_{-1}$
(AGN , $\mathbf{v_{-2}}$, x_1)	$\{x_1, x_2\}$	$\bar{x}_1 \mathrel{+}= \bar{v}_{-2}$

Figure 3.15: Propagation of the *useful* information for the coordinate transformation

Reverse activity analysis and adjusted internal representations

As mentioned before, it is possible to alter the internal representation of a function during the first application of the reverse mode. All evaluations of the tape that take place thereafter may then be performed without activity analysis. This way, function recomputation and derivation can be accomplished with a highly reduced effort. However, compared to the reverse activity analysis described so far, i.e., without altering the internal representation, the following issues must be considered:

- Depending on the applied AD-tool, rewriting the internal representation may be difficult to implement. In any case, additional effort at execution time results, which must be outperformed by the reductions of later tape evaluations to achieve a reduction in the overall runtime.
- It must be ensured that changes in the control flow caused by the evaluation at a different base point are detected reliably even if the tape has been adjusted.
- Removing operations from the internal representation also effects the stack of overwritten function values. Correctly adjusting the stack when removing an operation from the tape is therefore crucial for guaranteeing the correctness of derivatives computed with the reverse mode of AD.

Among these considerations, the control flow and the stack issue are of general interest. On a first glance, the reduction of the tape could be achieved by removing an operation from the tape in the moment it is considered in the derivation process if neither the result nor the argument variables of the operation belong to the set of *useful* variables. This, however, may cause wrong results when reevaluating the function based on the reduced tape. The underlying problem in removing control flow operations becomes apparent when considering Example 3.6. There, a function $y = F(x_1, x_2, x_3)$ is given on the left-hand side. Evaluated and taped at a base point for that $x_1 > \sqrt{5.0}$ holds, one obtains the internal representation depicted in the middle column of Example 3.6. Within the application of the reverse mode, which is also applied for further tape reduction, the handling of the operations in line `2` and `1` is critical. This situation is depicted by the question marks in the right part of the example. By examining the function F, it can be seen that y does not depend on x_1. Performing the pure removal approach, the two mentioned instructions would be deleted from the tape. When reevaluating the function using the reduced tape, branch switches cannot be detected, thus mostly yielding wrong results for $x_1 \leq \sqrt{5.0}$.

EXAMPLE 3.6

	Function	*Tape - forward mode*	*Tape - reverse mode*
1	$v = x_1 * x_1$	(MUL, 5, 1, 1)	?
2	if $(v > 5.0)$	(GT , 5, 5.0)	?
3	$y = x_2 + x_3$	(ADD, 4, 2, 3)	(ADD, 4, 2, 3)
4	else	–	–
5	$y = x_2 * x_3$	–	–

Hence, control flow instructions must not be removed from the tape even if only non-active arguments are involved. However, keeping only the control flow instructions may not be sufficient, as can be seen from Example 3.6, too. If the tape is reduced such that only the operations from line 2 and 3 remain, the detection of branch switches is possible but not necessarily correct. In particular, it depends on the randomly valued variable v whether a branch switch is signaled correctly or not. Hence not only the control flow instructions but also the computation chains yielding their arguments must be kept. This can be achieved by adding all argument variables of control flow instructions to the set of *useful* variables.

The second important issue concerns the correctness of the stack of overwritten function values. As has been discussed in Subsection 3.2.3, each arithmetic instruction overwrites a variable storing the result and is therefore associated with the recovery of the previous variable value in the reverse mode phase. This is done by storing the values onto the stack and retrieving them as necessary. Simply skipping the top element of the stack when removing an instruction from the tape does not guarantee the correctness of the reverse mode computations as illustrated in Example 3.7. There, a tape is used that has been

EXAMPLE 3.7

	Forward mode			*Reverse mode*		
	Function		*Stack*	*Derivation*	*UV*	v_1
1	$v_1 = x_1 + 2$	$= 3$	$*$	$\bar{x}_1 \mathrel{+}= \bar{v}_1$	$\{x_1, x_2\}$	$*$
2	$v_2 = v_1 * x_2$	$= 6$	$*$	$\bar{v}_1 \mathrel{+}= \bar{v}_2 * x_2,\ \bar{x}_2 \mathrel{+}= \bar{v}_2 * \boldsymbol{v_1}$	$\{v_1, x_2\}$	4
3	$v_1 = x_2 * 2$	$= 4$	3	—	$\{v_2\}$	4
4	$v_3 = v_1 + 1$	$= 5$	$*$	—	$\{v_2\}$	4
5	$v_1 = v_2 * 3$	$= 18$	4	$\bar{v}_2 \mathrel{+}= \bar{v}_1 * 3$	$\{v_2\}$	4
6	$y_1 = v_1$	$= 18$	$*$	$\bar{v}_1 \mathrel{+}= \bar{y}_1$	$\{v_1\}$	18
7	$y_2 = v_3$	$= 5$	$*$	—	$\{y_1\}$	18
					$\{y_1\}$	

created for a function $(y_1, y_2) = F(x_1, x_2)$ evaluated at the base point $(1.0, 2.0)$. Derivatives are to be computed for y_1 with respect to x_1 and x_2. In the taping phase, a stack of overwritten function values has been created. Values that are pushed onto the stack are depicted in the column "Stack" in the left part of Example 3.7. The evolution of the set of *useful* variables and the performed derivative instructions based on this information are shown in the columns "UV" and "Derivation". For the sake of brevity, all instructions explicitly setting variables to zero in the derivation phase have been omitted. These instructions, e.g., $\bar{v}_1 = 0$ for line 5, are required due to the reuse of variables. For the tape adjustment, it is assumed that the top element of the stack is skipped in case that the current instruction is removed from the tape. In addition, no recovery operation is performed. The different values of the variable v_1 resulting from the adjusted recovery process are presented in the corresponding column in Example 3.7. Using the mentioned skipping approach, the computed derivatives are definitely incorrect due to the calculations in line 2. There, the wrong value $v_1 = 4$ is used in the derivation. The correct value $v_1 = 3$ would have

been restored during the handling of the operation in line 3 that, however, has been removed from the tape due to the missing activity. Hence, a more sophisticated technique of adjusting the stack must be found to guarantee the derivative correctness.

For the given example, the value of variable v_1 is adjusted the first time during the handling of the instruction in line 5 in the reverse computations. If the value $v_1 = 3$ would be restored at this time instead of the value $v_1 = 4$, the correct value would be used in the computations for line 2. Then, the calculated derivatives would be correct, finally. More generally, if an instruction is removed from the tape, the corresponding overwritten value of the result variable must be removed from the stack. In addition, the next instruction of the reduced tape that overwrites the same variable must be found and the related value on the stack must be replaced by the value that just has been removed. This procedure requires write access to stack elements that are not on top. Since the overwritten values may well be organized as standard vector, this is no serious problem. The new replacement strategy, however, is only applicable under a specific condition. It must be ensured that the actually skipped value is not required by any derivative instruction corresponding to the reduced tape between the removed instruction and the next instruction manipulating the same variable. Considering the elimination of line 3 in Example 3.7, one observes that the variable v_1 must not be used as argument of any derivative instruction corresponding to line 4. If such an operation would be part of the reduced tape, all arguments would be elements of the set of *useful* variables, in particular the critical variable. Hence, the instruction manipulating this variable cannot be object to removal. This contradiction proves that, considering the removal of an operation, up to the previous operation changing the same variable no instruction exists that uses this variable as an argument. Thus, the replacement strategy for correcting the stack of overwritten variables may be utilized.

Implementing this approach requires the AD-tool to fulfill some special properties. Firstly, as said before, the adjusted sequence of overwritten values must be arranged in a structure that allows the access to all elements at any given time. If the original structure exhibits the same property, the adjusting algorithm may be performed in place, i.e., the original is overwritten. In the remainder of this subsection, it is assumed that this property holds. Secondly, the structure must consist at least semantically of the sequence of values that are overwritten during the function evaluation appended by the vector of the final values of all variables used in computation of the function. Thereby, the sequence of overwritten values is denoted by ov and the vector of final values by fv. Even though the overall structure is not a stack due to the required free access to all members, it is called stack in the remainder of this section and is referred to by st. Denoting by $\circ$ the concatenation of variable sequences, the following property holds

$$st = ov \circ fv.$$

In preparation of the reverse activity analysis, a new vector referred to by st_r is allocated for holding references to the stack. Its size must be chosen such that for all independent, intermediate and dependent variables of the function a reference can be stored. The elements of st_r initially point to the corresponding elements of fv. An additional reference, denoted by st_n, is used to mark the base of the reduced stack and initially points to the first element of fv. Furthermore, a reference st_c is defined. It points to the stack value that is used in the recovery process corresponding to the currently handled instruction of the tape. This reference is initialized with the same value as st_n. In the stepwise handling of the taped instructions, st_c is changed to traverse ov, accordingly. If the considered instruction is removed from the internal function representation, the value referenced by st_c is copied to the stack position referenced by the ith element of st_r, where i is the index of the corresponding result variable. Otherwise, st_n is adjusted to reference the preceding element of st, i.e., st_n is decremented, and the value referenced by st_c is copied to the stack position referenced by st_n. Additionally, the ith component of st_r is updated by assigning it the value of st_n. Updating st_r this way maintains a vector of references to stack positions that the last recovered value of a certain function variable is taken from. While adjusting tape and stack using the activity information, the recovery of function values is always performed. This ensures that the derivative calculation yields correct results during the reverse activity analysis.

The application of the described algorithm to Example 3.7 is depicted in Figure 3.16. The evolution of the new stack is presented in the left half. Based on the stack created in the taping phase, the adjusted version is created reusing the stack memory, illustrated by the gray background. Asterisks are used to represent

Instr.	st														st_c	st_n	st_r						
	0	1	2	3	4	5	6	7	8	9	10	11	12	13			x_1	x_2	v_1	v_2	v_3	y_1	y_2
	$*$	$*$	3	$*$	4	$*$	$*$	1	2	18	6	5	18	5	7	7	7	8	9	10	11	12	13
line 7	$*$	$*$	3	$*$	4	$*$	$\mathbf{*}$	1	2	18	6	5	18	$\mathbf{*}$	6	7	7	8	9	10	11	12	13
line 6	$*$	$*$	3	$*$	4	$\mathbf{*}$	$\mathbf{*}$	1	2	18	6	5	18	$*$	5	6	7	8	9	10	11	**6**	13
line 5	$*$	$*$	3	$*$	**4**	**4**	$*$	1	2	18	6	5	18	$*$	4	5	7	8	**5**	10	11	6	13
line 4	$*$	$*$	3	$\mathbf{*}$	4	4	$*$	1	2	18	6	$\mathbf{*}$	18	$*$	3	5	7	8	5	10	11	6	13
line 3	$*$	$*$	**3**	$*$	4	**3**	$*$	1	2	18	6	$*$	18	$*$	2	5	7	8	5	10	11	6	13
line 2	$*$	$\mathbf{*}$	3	$*$	$\mathbf{*}$	3	$*$	1	2	18	6	$*$	18	$*$	1	4	7	8	5	**4**	11	6	13
line 1	$\mathbf{*}$	$*$	3	$\mathbf{*}$	$*$	3	$*$	1	2	18	6	$*$	18	$*$	0	3	7	8	**3**	4	11	6	13

Figure 3.16: Stack adjustment for Example 3.7

arbitrary values that may reside in uninitialized memory. Changes to the stack st and the vector st_r are set bold within the corresponding lines. Additionally, the considered values of the original stack are set bold. The different elements of the stack are numbered from 0 through 13. These numbers are used as entries of st_r representing memory addresses. Once the adjustment of tape and stack is completed, the evaluation in the reverse mode of AD can be performed without any activity analysis. This is depicted in Figure 3.17. In the first step the vector of function values is initialized using the values of the vector

	Derivation	Recovery	Functions values						
			x_1	x_2	v_1	v_2	v_3	y_1	y_2
		init	1	2	18	6	$*$	18	$*$
line 6	$\bar{v}_1 \mathrel{+}= \bar{y}_1$	$y_1 = *$	1	2	18	6	$*$	$*$	$*$
line 5	$\bar{v}_2 \mathrel{+}= \bar{v}_1 * 3$	$v_1 = 3$	1	2	3	6	$*$	$*$	$*$
line 2	$\bar{v}_1 \mathrel{+}= \bar{v}_2 * x_2,\ \bar{x}_2 \mathrel{+}= \bar{v}_2 * v_1$	$v_2 = *$	1	2	3	$*$	$*$	$*$	$*$
line 1	$\bar{x}_1 \mathrel{+}= \bar{v}_1$	$v_1 = *$	1	2	$*$	$*$	$*$	$*$	$*$

Figure 3.17: Derivation for Example 3.7 based on the reduced tape and stack

fv. Thereafter, derivation instructions are performed for each operation of the reduced tape. Column "Recovery" in Figure 3.17 depicts the restoring of the result variable of the taped instruction within each step. As can be seen, the correct values of all arguments of each operation are available in the considered line. When all instructions are performed, the initial state of all function variables is reproduced.

Taking the solutions to the control flow and the stack issue into account, a new algorithm evolves that is suitable for applying activity analysis in the reverse mode phase for operator overloading based AD. The algorithm is presented in Figure 3.18. Due to the higher complexity of this algorithm compared to the variant without tape and stack adjustment, runtime improvements can only be achieved under certain conditions: The new tape and stack must be evaluated preferably without applying activity analysis. Furthermore, the higher the number of reevaluations, the more likely is a decrease in the overall runtime. Therefore, the decision in favor of one of the three possibilities, i.e., plain reverse mode, activity analysis every time or activity analysis only the first time, is highly dependent on the properties of the considered application.

Initialize the variables st_c and st_n to reference the first element of fv, the vector storing the function values using the corresponding elements of fv and the elements of the vector st_r to reference the corresponding element of fv. Reevaluate the created internal representation in reverse order instruction by instruction and do for all of them:

- if I is a control flow command:
  - $\Rightarrow$ set the *useful* state for all involved variables to true
  - $\Rightarrow$ do not perform any derivative calculation
- if I is the declaration of a dependent or independent variable:
  - $\Rightarrow$ set the *useful* state of the argument variable to true
  - $\Rightarrow$ perform appropriate derivative calculations depending on the internals of the applied AD-tool
- if I is an arithmetic instruction:
  - $\Rightarrow$ take a backup b of the *useful* state of the result variable v and set the property to false for v
  - $\Rightarrow$ update st_c to reference the preceding position of st
  - $\Rightarrow$ restore the function value of the instructions result variable
  - $\Rightarrow$ if b is true:
    1. set the *useful* property to true for all elements of $DP(e)$
    2. perform derivative calculations according to the represented instruction
    3. update st_n to reference the preceding position of st
    4. copy the value referenced by st_c to the stack position referenced by st_n
    5. update the component of st_r that corresponds to the result variable of the instruction to the stack position referenced by st_n

    else
    1. remove the instruction from the internal function representation
    2. copy the value referenced by st_c to the stack position referenced by the ith component of st_r where i corresponds to the results variable of the instruction

Figure 3.18: Algorithm for the reverse mode activity analysis including tape and stack adjustment

3.3 Parallelization strategies

So far, the emphasis of the newly developed techniques laid on the optimization of the tape creation and the sequential computation of derivative information, respectively. To solve most challenging projects in science and technology in a limited period of time, an additional component is required. The computations involved in these projects are often too expensive to be solved in a serial environment. They can only be handled efficiently using parallelism. Due to increasing availability of small-scale parallel computing platforms, i.e., the new standard office and home PC's, parallelization approaches will be considered more often even for middle scale or small tasks. Therefore, like many other techniques, automatic differentiation based on operator overloading must be extended in a way that allows to benefit from parallel computing facilities.

In the following subsection, an overview of so far available parallelization techniques for AD is given. Subsequently, methods that have been investigated and developed for this thesis are discussed. They are presented by the increasing order of inherent requirements.

3.3.1 State of the art

While analyzing the approaches that have been developed within the last decades, it turned out that all work done in the field that is known to the author is dedicated to the source-to-source technique for automatic differentiation. Several streams of ideas have been investigated. Sometimes, the mapping of a certain contribution is difficult due to the fact that it benefits from several ideas. Nevertheless, a classification is given in the following.

Uninterrupted reverse computation

As discussed in Subsection 3.1.3, a major decrease in the memory requirement of the reverse mode can be achieved by the application of checkpointing. However, this is only possible by increasing the computational effort. Due to the required recomputation of function values, the reverse propagation of adjoint information is performed piecewise at the frequency the required function values are available. By utilizing parallel computing facilities, the required recomputations can be performed by additional processors. Thus, the overall runtime of checkpointing based derivations can be reduced considerably. Correctly timed, all function information needed for the specific adjoint phases can be provided such that a single processor can compute these phases uninterruptedly [Ben96, Wal99].

Pipelined forward mode of AD

A comparable new approach is dedicated to the concurrent derivation of loop-iterations [Man02]. In contrast to the standard approach, the computation of directional derivatives immediately starts after the completion of the specific loop iteration rather than after the completion of the loop. Then, the runtime ratio $\omega_F \in [2, 2.5]$ of the forward mode can be exploited for parallelization. Without any restriction in terms of the loop dependencies, the second iteration of the loop can be started as soon as the derivation procedure for the first iteration has been initialized. This way, a software-pipelining model results. Exploiting the associativity of the chain rule, global derivative values are finally assembled using the computed local derivatives of each loop iteration. However, the limited degree of parallelism that results from the small ratio $\omega_F \in [3, 4]$ significantly restricts the application of this technique for parallel scalar forward mode derivation. As soon as the vector forward mode using p directions is to be applied, e.g., for the computation of sparse Jacobian's, the more convenient ratio $\omega_{VF} \in [1 + p, 1 + 1.5p]$ [Gri00] allows a higher degree of parallelism.

Function/AD-inherent parallelism

Significant effort has been invested to construct techniques that allow to benefit from parallelism inherent in either the function itself and the corresponding derivative computation, respectively. The key aspect of these approaches is to be seen in the uncoupled consideration of the function and its derivation. To be more precise, parallelism inherent in the function is not used as basis for constructing a parallel derivative procedure.

Early work in this field has been focused on computational graphs, e.g., [JG90, BGJ91]. There, optimized graphs are generated for the function as well as the related derivative function. Operations that can be handled in parallel are located at the same level of the considered graph. This insight is then exploited in the generation of optimized parallel forward or reverse mode program code.

Another approach [BBH02] has been developed for the automatic differentiation and parallelization of straight-line code that can be represented by an arithmetic circuit. The latter is a special directed, acyclic graph that consists of only leaves, addition and multiplication nodes. For this type of graph, the derivation procedure can be formulated as a sequence of graph transformations. They can be described by basic operations of linear algebra, mentioning the matrix$\times$matrix-multiplication as the most important within this context. Standard techniques for parallelizing these operations then lead to the parallel derivation procedure.

In addition, attention has been payed to parallelism that is available only for special versions of derivative computations. There, independent operations of the same type are executed in parallel when computing the Hessian of a given function or applying the vector forward mode of AD. For details, see [BLR+02, BRV06].

Transferred parallelization

Many approaches of implementing parallel AD try to utilize or transform user-provided parallelization information belonging to the function in a way that allows to benefit in the associated derivative computation, too. Accordingly, function and derivative computations possess roughly the same degree of parallelism. Data partitioning techniques are mostly exploited to achieve this result in the forward and reverse mode of AD.

Depending on the memory organization of the targeted computing platform, i.e., shared or distributed memory, different parallelization strategies are considered. Shared memory parallel programming often relies on OpenMP [DM98] whereas message-passing strategies like MPI [HW99] or PVM [DGMS94] are used for systems featuring distributed memory. At first glance, applying AD to OpenMP-based parallel functions is comparably straightforward when exploiting the forward mode [BLR+02]. Parallel AD based on message-passing systems, however, needs a higher level of attention. Especially data-flow analysis has been object to several investigations [Hov97, HB98, SKH06].

Message passing has been used for derivative computations in several projects. In many cases, hand-written parallelization functions have been provided for the derivation to serve as counterparts for the parallelization directives used within the function [CLGM96, HM00, HHG05]. In contrast, source-to-source compilers may be implemented to detect and replace MPI communication routines by special derivative counterparts automatically [FD99a, FD99b].

In addition to the approaches summarized above, the technique of Computational Differentiation (CD) may be exploited. CD combines available high level mathematical knowledge about the function and AD. Thus, by avoiding the black-box application of AD, improvements in memory usage or result accuracy can be achieved [HNRS99, BBH00]. If CD is used for transforming the function in a way that reveals independent subtasks, parallelization techniques will benefit as well.

For further details on the presented techniques, the reader is referred to the mentioned publications as well as the references given therein.

Analyzing the methods that have been used so far, a special conclusion can be drawn: All approaches known to the author focus on the source-to-source approach of implementing AD. However, a steadily growing number of applications from science and engineering is written in programming languages that are currently either not or at least not well supported in terms of source-to-source AD. Nevertheless, parallelization in combination with AD is often desired and can be provided in many cases by the use of operator overloading. Appropriate techniques are discussed in the remainder of this section.

3.3.2 Tapeless derivative computation

The two basic modes of AD, forward and reverse, are often provided by the same AD-tool. Therefore, the underlying implementation is usually tailored to the higher complexity of the reverse mode when applying operator overloading based AD. For tasks that allow to compute derivative information based on the forward mode of AD, e.g., when computing sparse Jacobian's, a more efficient implementation strategy exists. It is called *tapeless* AD and has been described in Section 2.2.2. Exemplarily, the multiplication operator is depicted in Figure 2.4. The approach is mainly characterized by the replacement of the arithmetic operations that propagate function values by operations that compute both function and derivative values. Due to the fact that the function value is also computed once the replacement has

been performed, this process is also called augmentation. Though a reverse mode differentiation based on this approach is not possible, it offers a second, important property besides the good compiler optimization: It is well suitable for a parallelization of the resulting code, no matter which paradigm, e.g., OpenMP [DM98], MPI [Hem94, HW99], is applied.

Applying a given parallelization strategy to spread computations among several processing elements, i.e., processors or cores of multi-core processors, always preserves the structure of the augmented operators and functions. This results from the specific time that the parallelization instructions are inserted into the source code. Based on the tapeless approach, the original function is the object of the parallelization efforts. The augmentation of the code for providing derivative information is done within the compilation process, subsequently. At this time, the parallelization is already complete. Hence, the more important question concerns the correctness of the augmented parallel code.

Assuming that the user function is parallelized correctly, i.e., no data access conflicts occur, the correctness of the parallel derivation procedure can be shown. In this context, correctness of the parallel derivation means that the correct derivative values are computed for the given user function, no matter whether the latter itself is correct or not. Due to the chain rule of calculus, derivatives may be computed on the basis of elemental operations and intrinsic functions. Therefore, the computed derivative information is correct if no data conflicts are introduced by the augmentation. As can be seen from (2.8), the derivative component of the result of an instruction is computed as the total derivative with respect to the constructed time dependence t, i.e.,

$$\dot{v}_i = \sum_{j \prec i} \frac{\partial \varphi_i(v_j)_{j \prec i}}{\partial v_j} \dot{v}_j.$$

From this setting, several conclusions for the derivation can be drawn:

1. For each argument of the original instruction, the derivative component of the augmented variable is accessed.
2. The derivative value of the result is changed in any case.
3. The function values of the arguments are needed frequently but not always.
4. In general, argument variables are only read-accessed whereas result variables are only write accessed.

In summary, one can conclude that the access structure of the augmented instruction is the same as that of the original instruction. Due to this property, data access conflicts are excluded as long as the original function does not exhibit such problems. In this case, the computed derivatives are correct.

Despite the high compatibility to the parallelization models, some limitations occur nevertheless. Firstly, care must be taken when optimizing the code for a given computing platform, e.g., cache optimization, as the applied derivative augmentation most probably breaks the intended technique. This applies to both the application of tapeless derivative computations for serial and parallel source codes. The most efficient solution to this drawback is the use of an “inlining” approach. Most high level programming languages that provide operator overloading also offer a technique for replacing calls to functions and operations by the corresponding definitions. Using this technique results in a code that contains both function and derivative instructions within the same compilation unit. It is then up to the compiler to exploit the more global view on the source code and provide highly optimized binaries.

A second limitation must be accepted for some less frequently used constructs of certain parallelization techniques. Reduction operations provided by MPI are a convenient example. They are only defined for a given set of data types that do not include AD data types. The only solution available at the moment is to provide a user written version of the reduction operation that can handle AD type variables. Hence, the desired functionality can be provided but induces additional efforts on the user side.

All in all, the tapeless variant of operator overloading based AD offers high compatibility to all parallelization techniques known to the author. Furthermore, due to the "inlining" technique and the avoidance of the tape handling, a comparably high efficiency of the derivative code can be achieved. The tapeless derivation should be preferred for all types of application for that derivatives can be computed efficiently using the forward mode of AD. However, reverse mode differentiation is often to be preferred due to the inherent advantages that have been described in Section 2.1.2. Appropriate solutions to handle tape-based reverse mode derivation processes in parallel are discussed in the following subsections.

3.3.3 Task-parallel AD-environment

As indicated by the name, the task-parallel approach basically identifies program parts – tasks – that can be handled independently of each other and may thus be executed simultaneously. Accordingly, the achieved execution type is often called *task parallelism*. It is to be distinguished from *data parallelism* where the same task is performed by many cooperators on a certain subset of the available input data, see subsection 3.3.4. For many parallel computer programs, a mixture of these two techniques is applied [DFF+03, GR03, GGKK03].

Throughout this subsection, pure task parallelism is addressed. Accordingly, the following properties arise from the combined application of this technique and automatic differentiation based on operator overloading:

- Computations for each task are done in serial.
- Data allocation is done by the tasks individually.
- Tape creation and evaluation is done by the same task without interruption by the enclosing program.
- Computations that do not concern AD and therefore do not involve facilities of the AD-tool may be handled within the preceding or subsequent phases.

The basic layout of this approach is depicted in Figure 3.19. There, the non-AD-phases are illustrated by white objects whereas AD-specific actions are represented using different shades of gray. Furthermore, the abbreviation "DA" points to the different data allocation phases.

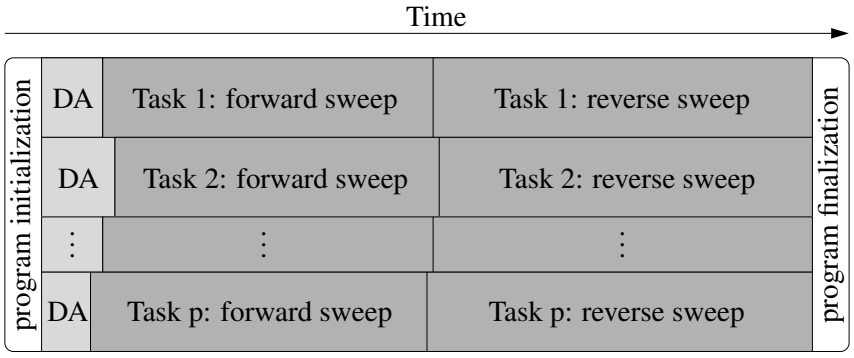


Figure 3.19: Basic layout of task-parallel calculations in an AD-environment

As can be seen, the demands on the AD-tool are quite reasonable and can be summarized as the supply of a task-safe environment. The term *task-safe* is abutted to the well-known paradigm *thread-safety*. This means in particular that the execution of a program containing several scheduling entities, i.e., processes or threads, must terminate with the correct result. A prominent example of missing thread-safety is given by the combination of OpenMP and static variables in C/C++. Static variables are implemented as shared variables in the common address space of all threads. Write-access to this variables possibly inflicts a data race that may in return cause the computation of wrong results.

Task-safety only requires a limited set of properties. Firstly, it must be ensured that a tape generated by a specific task cannot be overwritten or forged by another task. Secondly, the data integrity must be ensured by dedicating used function and derivative variables to the specific tasks. Parallelization by use of MPI does only need to take care of the first point as the separated address space of the involved processes provides the data integrity automatically. This is not true for OpenMP-based parallelizations where all threads operate in the same address space. Due to the structure of the program, it seems fairly obvious to create a separate copy of the AD-environment for every working thread. This must include all relevant control information that is involved in the creation of variables of AD data type. Variables declared and defined within parallel regions are treated private when using OpenMP. AD-type variables must be implemented such that this condition is satisfied. This is especially true for compiler generated temporaries that are used when dividing composed arithmetic instructions into elemental operations and functions as illustrated int the following example.

EXAMPLE 3.8

Instruction	*Elemental operations*
$d = a * b * c$	$t_1 = a * b$
	$t_2 = t_1 * c$
	$d = t_2$

There, assuming several threads performing the same instruction, it must be ensured that the compiler generated temporaries t_1 and t_2 are unique to the treads. Otherwise, a race condition is possible that may cause the computation of wrong results. Whereas compiler generated temporaries of standard data type are always thread-safe, variables of AD data type need special attention. It must be ensured that the ID representing the variable within the internal function representation is unique when executing in serial as well as in parallel. Without the copying of the AD-environment, the assignment of the ID must be protected within the parallel environment. This could be done by use of a critical section but would result in a major drawback. Due to the nature of the critical section, only one thread can execute the guarded code at a specific time. All other threads would be forced to wait at the beginning of the critical section. Overall, a time shifted behavior would result that is illustrated in Figure 3.20. There, the critical

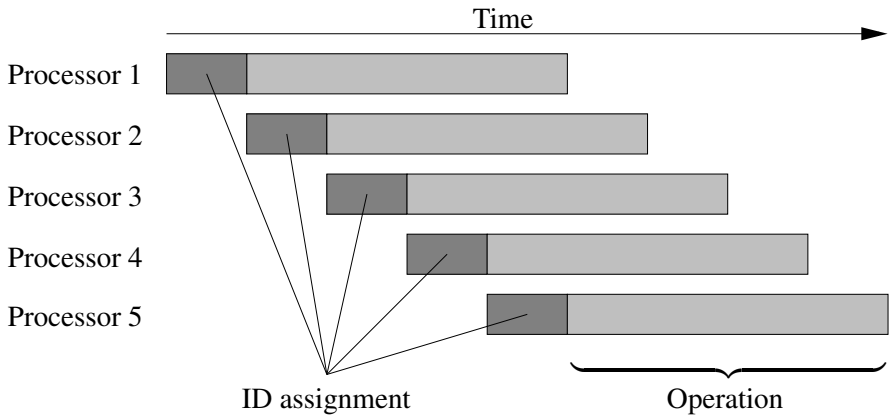


Figure 3.20: Layout of a parallel execution with critical section protected ID assignment

section protected generation of the temporary variables is depicted by the dark gray boxes whereas the arithmetic instructions producing their results are represented by the light gray boxes.

Determining the maximal benefit that can be drawn from the parallelization is somewhat difficult for this setting. A common measure for qualifying the result of the parallelization is given by the speedup $Sp(p)$ that is defined as the ratio of the serial and the parallel runtimes, i.e.,

$$Sp(p) := \frac{T(1)}{T(p)}. \tag{3.26}$$

Thereby, $Sp(p)$ denotes the speedup using p processing elements and $T(1)$ and $T(p)$ are the serial and parallel runtimes, respectively. A well known limitation of the speedup is called Amdahl's law identifying the maximal speedup $\max\big(Sp(p)\big)$ as the ratio of the serial runtime $T(1)$ and runtime T_s of the serial program parts [Amd67], i.e.,

$$\max\big(Sp(p)\big) = \frac{T(1)}{T_s}. \tag{3.27}$$

This result can be derived by determining the limit $p \to \infty$ for equation (3.26) and substituting $T(1)$ by $T_s + T_p$ and $T(p)$ by $T_s + \frac{T_p}{p}$, yielding

$$\begin{aligned} \lim_{p\to\infty} \frac{T(1)}{T(p)} &= \lim_{p\to\infty} \frac{T_s + T_p}{T_s + \frac{T_p}{p}} \\ &= \frac{T(1)}{T_s}. \end{aligned}$$

Thereby, T_p denotes the serial runtime of the program parts that might be run in parallel. Assuming that the original function could be evaluated in a complete parallel and balanced manner, thus eliminating the speedup limitation, Amdahl's law would nevertheless be the limiting factor when computing derivatives according to the scheme depicted in Figure 3.20. Due to the required critical section that protects the ID assignment during the creation of variables, a program part is generated that can be handled only serially. Though this serial part is introduced at instruction level, the average ratio between serial and parallel code handling the specific instructions determines the ratio for the overall program. This is caused by the property that all arithmetic instructions with exception of assignment based operations, e.g., $=$, $+=$, ..., involve compiler generated temporaries. However, the main limitation of the speedup is not caused by the frequently used critical sections but by their high costs compared to the relatively small guarded instructions. The resulting ratio between serial and parallel parts of the program is thus unfavorable. Accordingly, it is very unlikely that a significant speedup can be achieved using this approach. For that reason, the environment copying approach is to be preferred.

Mostly, a specific function of the tool has to be called that prepares the application for the parallel execution. This function may be called prior to the parallel region, see e.g., CppAD, and sets the maximal number of tapes to be handled simultaneously or is called via the constructor of a tool provided private variable, e.g., ADOL-C (see Section 4.1).

Taking all demands into account, the skeleton of a corresponding application can be derived:

1. program initialization
2. preparing calculations – this may include parallel computations away from AD
3. call to the special function of the AD-tool that prepares the parallel execution – this may include the copying of the control structures of the AD-tool
4. buildup of the parallel environment if not done in the previous step
5. data allocation – task private
6. forward sweep – task private
7. reverse sweep – task private
8. destruction of the parallel environment – possibly including data exchange from the parallel to the serial part of the program
9. final computations – this may include parallel computations not differentiated automatically
10. program termination

Using this schedule, the task parallelism can be exploited by applications that employ automatic differentiation. However, as pointed out in [DFF+03, p. 50]: *"... — task parallelism is typically limited to small degrees of parallelism"*. Restricting parallelization strategies to this level would be too strong and would prevent higher speedups for many applications. Techniques for exploiting data parallelism contained in AD-based computations are studied in the following subsections.

3.3.4 Data-parallel AD-environment

As mentioned in the previous subsection, data parallelism describes the subdivision of the data domain of a given problem into several regions. These regions are then assigned to a given number of processing elements, which apply the same tasks to each of them. Data parallelism is commonly exploited in many scientific and industrial applications and exhibits a "natural" form of scalability. Since the problem size for such applications is normally expressed by the size of the input data to be processed, an upscaled problem can typically be solved using a correspondingly higher number of processing elements at only a modestly higher runtime [DFF+03].

In the remainder of this section, data parallelism is extended beyond the pure nature that has been described above. Accordingly, it shall be allowed that the complete set of independent variables XI of the given function F may be used by all p processing elements PE_i, $i = 1, \ldots, p$, for reading. Denoting by $XI_r^{(i)}$ and $XI_w^{(i)}$, respectively, the read and write accessed subsets of XI, respectively, for the various processing elements, it holds that

$$\forall PE_i: \quad XI_r^{(i)} \subseteq XI, \qquad XI_w^{(i)} = \emptyset. \tag{3.28}$$

This allows for parallel functions that, e.g., handle the evolution of systems consisting of many components. There, computations for the individual components can be performed independently, provided that the interaction among them can be determined using XI. Derivative information for such applications can be provided using the scheme depicted in Figure 3.21. Compared to the task-parallel approach depicted in Figure 3.19, the data-parallel layout discussed here exhibits only a small number of differences:

- All processing elements perform the same task.
- A single input data allocation phase is dedicated to the whole program.
- An additional AD finalization phase is appended to the derivation process.

In contrast to the general read access in terms of the independents, write access may only be allowed for distinct subsets $YD^{(i)}$ of the dependent variables. For a given number p of processing elements it is required that

$$YD = \bigcup_{i=1}^{p} YD^{(i)} \qquad \text{and} \qquad YD^{(i)} \cap YD^{(j)} = \emptyset \qquad \text{with} \quad i, j \in [0, p],\ i \neq j.$$

The set of intermediate variables $IV^{(s)}$ associated with the serial function evaluation is considered to be reproduced for all processing elements yielding p sets of variables $IV^{(i)}$, $i = 1, \ldots, p$. In this way, intermediate values can be used at the certain processing element without the potential of memory conflicts. Overall, considering the subset $YD^{(i)}$ and the set of intermediate variables $IV^{(i)}$, which are used exclusively by the processing elements PE_i, it holds that

$$\forall PE_i \, \forall PE_j: \begin{cases} YD^{(i)} \cap YD^{(j)} = \emptyset & IV^{(i)} \cap IV^{(j)} = \emptyset \quad \text{for} \quad i \neq j \\ YD^{(i)} = YD^{(j)} & IV^{(i)} = IV^{(j)} \quad \text{for} \quad i = j \end{cases}. \tag{3.29}$$

Any function exhibiting the properties (3.28) and (3.29) is considered correctly parallelized in the sense that data races are debarred.

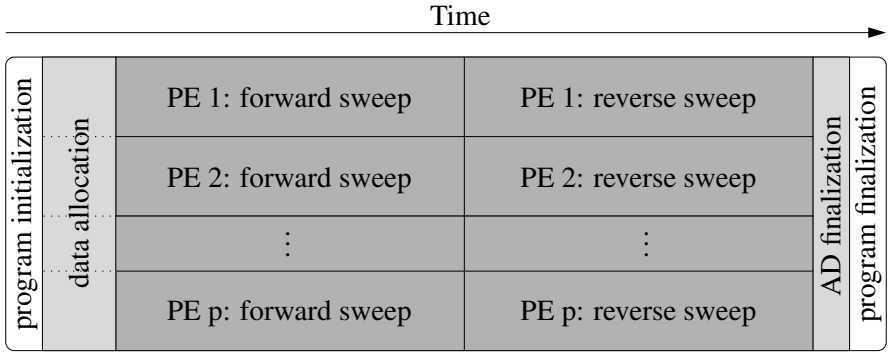


Figure 3.21: Basic layout of data-parallel calculations in an AD-environment

Obviously, equation (3.29) allows to compute derivatives for the given function as long as only the sets $IV^{(i)}$ and $YD^{(i)}$ are involved. Due to the distinct nature of the sets defined for the processing elements, forward and reverse mode of AD can be applied safely. A less obvious situation is given as soon as independent variables are involved in the computation. As known from the theory of the reverse mode, see Subsection 2.1.2, read accessed function variables result in write accessed derivative variables. More precisely, the following relation holds

$$\begin{array}{ll} \underline{\text{Function}} & \underline{\text{Derivative (reverse mode)}} \\ v_i = \varphi_i(v_j)_{j \prec i} & \bar{v}_j \mathrel{+}= \bar{v}_i * \frac{\partial \varphi_i(v_j)_{j \prec i}}{\partial v_j} \qquad \forall j \in \{j : j \prec i\}. \end{array}$$

As can be seen, due to the required instructions in the reverse mode, the data access layout of the function variables is reversed for the adjoint variables. Hence, read accesses on the independent variables x_k, $k = 1, \ldots, n$, induce the potential of data races in the adjoint computations. Consider the following example given as C-code.

EXAMPLE 3.9

Function

```
for (i = 0; i < p; ++i)
    IV[i] = 2 * XI[0];
```

Derivative (reverse mode)

```
for (i = p - 1; i >= 0; --i)
    XI_bar[0] += IV_bar[i] * 2;
```

Here, the various `IV[i]` are intermediate variables and `XI[0]` is an independent variable. The corresponding counterparts in the derivation phase are `IV_bar[i]` and `XI_bar[0]`. As long as function and derivative computations are performed in serial, no data access conflicts occur. This is no longer true if the two phases are executed in parallel. Then, a data race results for `XI_bar[0]`. A common solution to this problem is to declare the critical incrementation operation, i.e., +=, as atomic. Data access conflicts due to the concurrent use of the same variable are impossible thereby. However, this most likely causes a increase in runtime as the parallel code may serialize at the atomic operation. For the given class of applications featuring the properties (3.28) and (3.29), a better solution can be constructed.

Similar to the handling of intermediate variables, different sets of adjoint variables $\overline{XI}^{(i)}$ can be provided for each processing element corresponding to the set XI. Adjoint values may then be updated locally by each processing element independently and thus globally in parallel. Due to the additive nature of the derivative computations, global adjoint values may later be assembled using the local information produced by the various processing elements. As this assembling requires additional communication in a distributed memory environment, it should be executed for all relevant adjoints in a single step. Thus, the assembling step must be performed after the last update of an adjoint variable $\bar{x}_k^{(i)} \in \overline{XI}^{(i)}$.

However, updates of adjoints $\bar{x}_k^{(i)}$ are principally possible at any point of the reverse computations. This results from the property of the function that independent variables may be accessed at any given time of the function evaluation. Thus, a single adjoint assembling step is only possible if no $\bar{x}_k$ is used as an argument of a derivative instruction before all updates on the set $\overline{XI}^{(i)}$ have been performed. For the considered type of applications that feature the properties (3.28) and (3.29), this potential conflict can never occur. Since the individual independent variables are accessed for reading only, their derivative counterparts cannot appear on the right-hand side of any derivative instruction. Hence the assembling phase that computes the global derivative values can be safely moved to the end of the derivation process. This is depicted by the *AD finalization* phase in Figure 3.21.

The parallel computation of derivatives for the considered type of applications is demonstrated in Example 3.10. There, the considered function and the examined base point are presented in the upper left part. In the upper right part a schematic source code, e.g., the main computation routine, is given in a C-like programming style. Based on this scheme, the evaluation procedures for the serial and the parallel execution can be created as depicted in the central part of Example 3.10. Finally, the corresponding derivation procedures are shown in the lower part. In Example 3.10, result values of the function and derivative evaluation have been marked by use of circles. Furthermore, variables that are updated in the derivation process and have been used before are set bold.

The additionally introduced AD finalization phase that is used for assembling the global derivatives is depicted by a gray box in the parallel derivation (line `9`). Having a closer look on potential runtimes reveals an important aspect on the use of the assembling phase: It reduces the speedup of the parallel computation. Assuming each operation depicted in Example 3.10 to consume exactly one clock cycle, speedups can be computed for the function and the derivation phase.

Function	Derivation
$Sp(2) = \frac{T(1)}{T(n)}$	$Sp(2) = \frac{T(1)}{T(n)}$
$= \frac{8 \text{ cycles}}{4 \text{ cycles}}$	$= \frac{16 \text{ cycles}}{9 \text{ cycles}}$
$= 2$	≈ 1.8

Here, the linear speedup of the function parallelization cannot be achieved when parallelizing the derivation procedure. The assembling phase that is responsible for this decrease can however not be omitted as it allows the execution of the main computations concurrently. As can be seen, using the same set of adjoint variables $\overline{XI}$ for the two processing elements in Example 3.10 would result in an unsafe derivation procedure. Data races would result for line `5` up to line `8`. By utilizing the adjoint assembling, only a single synchronization after line `8` becomes necessary. Then, the correctness of the result can be guaranteed.

Data parallelism in many programs is often implemented using loops. Assuming that the special data access structure that results from (3.28) and (3.29) applies to the mentioned function, a technique can be constructed that allows to parallelize the corresponding derivation procedure appropriately. The details are discussed in the following subsection.

3.3.5 Loop-level parallelization

Using the data-parallel derivation approach described in the previous subsection and the concept of the external differentiated functions, a technique can be designed that allows to compute derivatives for functions containing parallel loops. This implies that an explicit loop-handling approach is applied, i.e., OpenMP. In contrast to the simplified application that has been considered in the previous subsection, i.e., that the complete function is parallelized, in practice parallel parts of the function are often embedded in a serially handled context. In particular it is assumed that the loop is preceded by a serial startup

EXAMPLE 3.10

Function

$$(y_1, y_2) = F(x_1, x_2)$$

with

$$y_i = (x_i * x_1 + x_i * x_2) * x_i \qquad i = 1, 2$$

$$x_1 = 1, \quad x_2 = 2$$

C-like source code

```
for (i = 1; i < 3; ++i) {
    v_i1 = x_i * x_1
    v_i2 = x_i * x_2
    v_i3 = v_i1 + v_i2
    y_i = v_i3 * x_i
}
```

Reduced evaluation procedure

	serial		*parallel* *PE 1*		*parallel* *PE 2*	
1	$v_{11} = x_1 * x_1$	$= 1$	$v_{11} = x_1 * x_1$	$= 1$	$v_{21} = x_2 * x_1$	$= 2$
2	$v_{12} = x_1 * x_2$	$= 2$	$v_{12} = x_1 * x_2$	$= 2$	$v_{22} = x_2 * x_2$	$= 4$
3	$v_{13} = v_{11} + v_{12}$	$= 3$	$v_{13} = v_{11} + v_{12}$	$= 3$	$v_{23} = v_{21} + v_{22}$	$= 6$
4	$y_1 = v_{13} * x_1$	$= (3)$	$y_1 = v_{13} * x_1$	$= (3)$	$y_2 = v_{23} * x_2$	$= (12)$
5	$v_{21} = x_2 * x_1$	$= 2$				
6	$v_{22} = x_2 * x_2$	$= 4$				
7	$v_{23} = v_{21} + v_{22}$	$= 6$				
8	$y_2 = v_{23} * x_2$	$= (12)$				

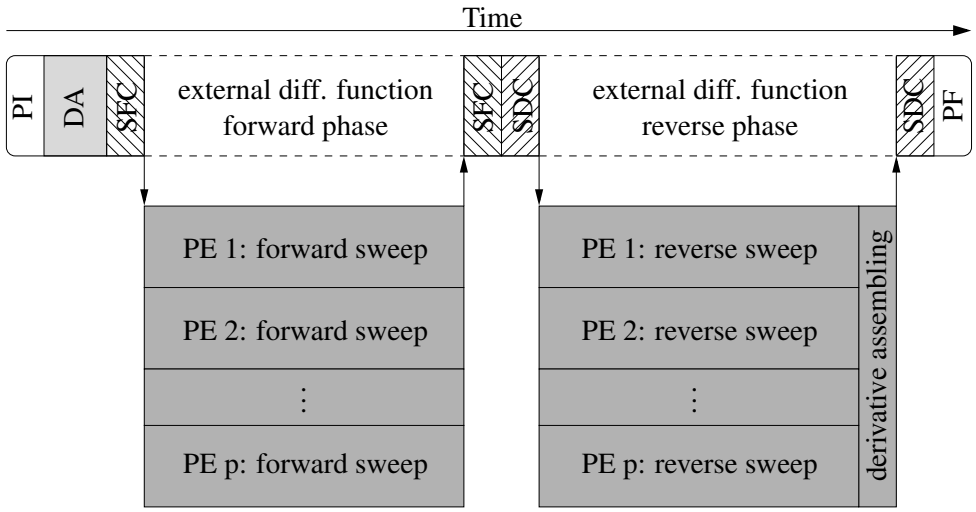
Reduced reverse mode derivation procedure for $\bar{y}_1 = \bar{y}_2 = 1$

	serial		*parallel* *PE 1*		*parallel* *PE 2*	
1	$\bar{v}_{23} \mathrel{+}= \bar{y}_2 * x_2$	$= 2$	$\bar{v}_{13} \mathrel{+}= \bar{y}_1 * x_1$	$= 1$	$\bar{v}_{23} \mathrel{+}= \bar{y}_2 * x_2$	$= 2$
2	$\bar{x}_2 \mathrel{+}= \bar{y}_2 * v_{23}$	$= 6$	$\bar{x}_{11} \mathrel{+}= \bar{y}_1 * v_{13}$	$= 3$	$\bar{x}_{22} \mathrel{+}= \bar{y}_2 * v_{23}$	$= 6$
3	$\bar{v}_{21} \mathrel{+}= \bar{v}_{23}$	$= 2$	$\bar{v}_{11} \mathrel{+}= \bar{v}_{13}$	$= 1$	$\bar{v}_{21} \mathrel{+}= \bar{v}_{23}$	$= 2$
4	$\bar{v}_{22} \mathrel{+}= \bar{v}_{23}$	$= 2$	$\bar{v}_{12} \mathrel{+}= \bar{v}_{13}$	$= 1$	$\bar{v}_{22} \mathrel{+}= \bar{v}_{23}$	$= 2$
5	$\bar{\boldsymbol{x}}_{\boldsymbol{2}} \mathrel{+}= \bar{v}_{22} * x_2$	$= 10$	$\bar{\boldsymbol{x}}_{\boldsymbol{11}} \mathrel{+}= \bar{v}_{12} * x_2$	$= 5$	$\bar{\boldsymbol{x}}_{\boldsymbol{22}} \mathrel{+}= \bar{v}_{22} * x_2$	$= 10$
6	$\bar{\boldsymbol{x}}_{\boldsymbol{2}} \mathrel{+}= \bar{v}_{22} * x_2$	$= 14$	$\bar{x}_{12} \mathrel{+}= \bar{v}_{12} * x_1$	$= 1$	$\bar{\boldsymbol{x}}_{\boldsymbol{22}} \mathrel{+}= \bar{v}_{22} * x_2$	$= 14$
7	$\bar{\boldsymbol{x}}_{\boldsymbol{2}} \mathrel{+}= \bar{v}_{21} * x_1$	$= 16$	$\bar{\boldsymbol{x}}_{\boldsymbol{11}} \mathrel{+}= \bar{v}_{11} * x_1$	$= 6$	$\bar{\boldsymbol{x}}_{\boldsymbol{22}} \mathrel{+}= \bar{v}_{21} * x_1$	$= 16$
8	$\bar{x}_1 \mathrel{+}= \bar{v}_{21} * x_2$	$= 4$	$\bar{\boldsymbol{x}}_{\boldsymbol{11}} \mathrel{+}= \bar{v}_{11} * x_1$	$= 7$	$\bar{x}_{21} \mathrel{+}= \bar{v}_{21} * x_2$	$= 4$
9	$\bar{v}_{13} \mathrel{+}= \bar{y}_1 * x_1$	$= 1$	$\bar{x}_1 = \bar{x}_{11} + \bar{x}_{21}$	$= (11)$	$\bar{x}_2 = \bar{x}_{12} + \bar{x}_{22}$	$= (17)$
10	$\bar{\boldsymbol{x}}_{\boldsymbol{1}} \mathrel{+}= \bar{y}_1 * v_{13}$	$= 7$				
11	$\bar{v}_{11} \mathrel{+}= \bar{v}_{13}$	$= 1$				
12	$\bar{v}_{12} \mathrel{+}= \bar{v}_{13}$	$= 1$				
13	$\bar{\boldsymbol{x}}_{\boldsymbol{1}} \mathrel{+}= \bar{v}_{12} * x_2$	$= 9$				
14	$\bar{\boldsymbol{x}}_{\boldsymbol{2}} \mathrel{+}= \bar{v}_{12} * x_1$	$= (17)$				
15	$\bar{\boldsymbol{x}}_{\boldsymbol{1}} \mathrel{+}= \bar{v}_{11} * x_1$	$= 10$				
16	$\bar{\boldsymbol{x}}_{\boldsymbol{1}} \mathrel{+}= \bar{v}_{11} * x_1$	$= (11)$				

calculation and followed by a serial finalization phase. Computing derivatives for this type of functions applying the reverse mode of AD and utilizing a tape- and operator overloading-based AD-tool inflicts two major challenges:

- The parallelization directives of OpenMP are not part of programming languages such as, e.g., C++. Accordingly, these directives cannot be overloaded and, therefore, cannot be recognized by the AD-tool.
- All tape- and operator overloading-based tools known to the author do not create an internal function representation that includes the loop structure itself. Rather, loops are always unrolled, creating a long sequence of instruction.

All in all, a situation occurs that prevents the recognition of both the loop structure and the parallelization statements. Without appropriate user interaction, a parallelization of the derivative calculation is impossible under these circumstances. User steering can be provided by employing the concepts of external differentiated functions and nested taping. The resulting layout of the computation of function values and derivative information is depicted in Figure 3.22. To overcome the problem of missing information



PI - program initialization
DA - data allocation
SFC - serial function computation
SDC - serial derivative computation
PF - program finalization

Figure 3.22: Loop level parallelization for data parallel applications

concerning the loop structure and the parallelization, the serial taping process must be interrupted at the begin of the parallel loop contained within the function. Then, the external differentiated function concept allows to switch to a user provided function that can be used to create separate tapes for the single loop iterations or chunks thereof. Once the loop has been performed, the interrupted taping process is resumed. Computing derivatives applying the reverse mode of AD basically follows the same strategy but in reverse order. Therefore, the initial step in the derivation phase is the application of the appropriate reverse variant to the first tape. At the position where the parallel loop has been completed during the function evaluation, the derivative handling is interrupted and a second user provided function is called. The latter is responsible for handling the derivation of the parallel loop. This can easily be accomplished by applying the technique that has been described in the previous subsection. As soon as the derivative assembling phase has been completed, control is given back from the user routine to the AD-tool that resumes the derivation of the outer, serial tape.

Conceptually, to allow the application of the data-parallel approach discussed in the previous subsection, the requirements (3.28) and (3.29) must be met by the parallel loop. In particular, all processing elements

may access the complete input data set of the loop for reading. This includes the independent variables of the overall user function and the function values computed up to this time. Therefore, the applied AD-tool must feature a mechanism to provide these data to all processing elements. As the data are only accessed for reading, a simple copying function might be used for this purpose. The remaining variables, i.e., intermediates and output variables of the loop iterations, local to each processing element are considered to be private anyway. Overall, this allows a safe derivative calculation within the parallel environment. Data transfer between the different stages of the derivation process is handled using derivative context switches. This way, the demands on the user are reduced significantly. Furthermore, this allows not only to handle derivative calculations in parallel but also to couple this approach with the other techniques that have been described in this chapter. Details on the implementation, the applied AD-tool and the considered applications are presented in the following chapter.

4 Concept validation

After deriving the new concepts for Automatic Differentiation (AD) in Chapter 3, this chapter focuses on the aspects of the integration into an existing AD-tool and the application to selected numerical examples. ADOL-C, a package for the automatic differentiation of algorithms written in C/C++ [GJU96], has been chosen for being the base of the tool-based work. In the following section, a short overview of the tool and the provided infrastructure are given first. Thereafter, adoptions for enabling the techniques presented in Chapter 3 are discussed in more detail. The second section is then dedicated to three applications that make use of the newly provided facilities of ADOL-C. For each of them an introduction to the background of the application is presented and resulting challenges are revealed. Observations and consequences from the results gathered during runtime studies conclude this chapter.

4.1 ADOL-C: a tool for automatic differentiation

ADOL-C is an operator overloading based AD-tool that allows to compute derivatives for functions given as C or C++ source code. The development was started by Andreas Griewank in the early 1980s and has been continued up to the present. Thus, ADOL-C exhibits a longer history than many other tools used among the AD community and is considered highly valuable. The clear structure of the tool and the underlying approach always guaranteed an adequate entrance to the world of AD. In return, ADOL-C often benefited from the talented developers by being raised to the highest level of available techniques.

Applications that utilize ADOL-C can be found in many fields of science and technology. This includes, e.g., fish stock assessment by the software package CASAL [BFD$^+$05], computer-aided simulation of electronic circuits by fREEDA [HKL$^+$05] and the numerical simulation of optimal control problems by MUSCODII [BLSS03]. Currently, ADOL-C is further developed and maintained at the TU Dresden, Institute of Scientific Computing by a research group that is guided by Andrea Walther. Many techniques that are discussed in Chapter 3 are meanwhile incorporated into the tool and will be available with the next version.

4.1.1 Initial state

ADOL-C provides facilities for the creation and evaluation of tapes, i.e., the internal function representations, created during separate so-called taping phases. A taping phase encapsulates the evaluation of the implemented function or a part thereof at a given base point. The creation of the tape is started with a call to the tool-provided routine *trace_on* and finalized by calling *trace_off*. When referring to the tape, a unique identifier *tag* is used that is passed to *trace_on*. Between the two calls, an internal representation of every operation is created that involves variables of an augmented data type. This special data type is called *adouble* and is used for the representation of scalar values. Choosing the set of variables that has to be declared of an augmented data type is basically up to the user. However, the compiler-aided augmentation strategy that has been described in Subsection 3.2.1 may be utilized to significantly reduce the set of relevant variables. Two rarely used binary shift operators, <<= and >>=, are used to identify independent and dependent variables, respectively. As soon as the taping phase is completed, specific drivers provided by ADOL-C may be applied. The basic drivers allow to reevaluate the function at the same or a different base point and to compute derivative information of any order using the forward or reverse mode of AD based on the created internal function representation. In addition, advanced drivers

are available for determining and exploiting the sparsity structure of derivative matrices, for solving Ordinary Differential Equations (ODEs), determining higher order derivative tensors and for computing derivatives of implicit or inverse functions. A corresponding source code that takes into account all requirements is given in Example 4.1. There, a tape with number 1 is created while evaluating the function

EXAMPLE 4.1

```
...
adouble xa[2], ya;                        // augmented variables
double xp[2], yp, xp_bar[2], yp_bar;      // standard variables
xp[0] = 1.0; xp[2] = 2.0;                 // initialization

trace_on(1, 1);                           // start taping
xa[0] <<= xp[0]; xa[1] <<= xp[1];         // declare independent variables
ya = xa[0] * xa[0] + xa[1] * xa[1];       // evaluate the function
ya >>= yp;                                // declare dependent variables
trace_off();                              // stop taping

yp_bar = 1.0;                             // initialize adjoint variables
fos_reverse(1, 2, 1, &yp_bar, xp_bar);    // compute gradient in reverse mode
...
```

$y = f(x) = x_1^2 + x_2^2$ at the base point $(1.0, 2.0)$. The second argument to *trace_on* enforces the recording of overwritten function values and prepares the immediately following call to *fos_reverse* that computes the derivative values. At this point any other driver routine provided by ADOL-C may be called as well, assuming that an appropriate initialization is performed before.

Internally, each variable of an augmented data type stores a unique identifier, the so-called location. Locations are acquired during the variables construction and are released during destruction. Variable values are stored within a large field referred to as *store*. Within the store, the individual values are addressed using the location of the corresponding *adouble* variable. Due to the arrangement in form of a large vector, the final values of all variables that have been used at any given point of the function, are still available when the evaluation and reevaluation, respectively, is completed. These values can then be appended to the stack of overwritten function values and form the initial point of the restoring activities in the reverse mode differentiation.

The internal representation created by ADOL-C consists of up to four different sub-tapes. They are denoted by operation, location, value and Taylor tape. The latter represents the stack of overwritten function values and is only created if explicitly requested. When creating the internal representation of a given elemental function, several actions are performed. First, a specific code identifying the elemental function is written onto the operation sub-tape. Then, the locations of involved variables are written onto the location sub-tape and the constant values used by the elemental function are written onto the value sub-tape. Finally, if the creation of the stack has been requested, the value of the result variable is written onto the Taylor tape before it is overwritten. During the evaluation of the tape, the involved drivers read back the codes from the operation tape and extract the corresponding elements from the other tapes. It is ensured by the implementation of ADOL-C that the same quantity of information is taken from the sub-tapes that has been written during creation. Thus, a correct evaluation of the created tapes can be guaranteed.

So far, ADOL-C could only create several tapes within the same program as long as the different taping phases did not overlap. This results from the status information being dedicated to the whole program rather than individual tapes. The key features that allow to eliminate this restriction are discussed in the following subsection.

4.1.2 New tape management

Tape status information utilized by ADOL-C can be classified into two groups. On the one hand, one has information that are only required when creating or evaluating a specific tape. This includes data for controlling the employed buffers, e.g., size and current position as well as general information, e.g., hard disc access status and overall sub-tape sizes. These data referred to as *dedicated tape information* (DTI) may be reset as soon as the tape is recreated. In addition, static information, e.g., sub-tape names are maintained that are created only once, at first access. On the other hand, information that is shared amongst all tapes must be handled. The most prominent representative is the store as well as the control information for handling it. These data is referred to as *global tape information* (GTI) and its life time and handling are dedicated to the whole program.

In principle, all information could be dedicated to the individual tapes. In that case, a store containing the values of augmented variables must be provided for every tape. This would provide a dedicated working environment for every tape and the context switch would be reduced to the switch of the tape information. However, the distinction between dedicated and global tape information as implemented in ADOL-C offers two advantages.

- Due to the dedication of the *global tape information* to the whole program rather than to individual tapes, augmented variables are usable outside the taping phases. This is true even if no tape is created during the whole program. In addition, this allows the commonly use of global variables that are initialized before the first statement of the program entry routine is executed.
- Sharing the store amongst all tapes also enables the calculations for a tape to directly benefit from computations done in earlier phases of the program. In particular, no copy operations are necessary to transfer the results or, if necessary, any other values of the preceding tape to the current tape. Especially in a situation where the creation of tapes is nested, the reuse of variable values is very likely.

To support the techniques described in Chapter 3, several changes had to be applied to the internals of ADOL-C. First, all variables containing status data have been united into two structures representing dedicated and global tape information. One instance of the global tape information is created prior to the execution of the program entry routine. As long as computations are performed serially, no additional instance is necessary. For each tape, an instance of the dedicated tape information is created. In addition, a vector and a stack storing references to these DTI instances are available. A reference is appended to the vector each time a new tape and the corresponding DTI instance are created. Using this vector, the control information of the individual tapes are addressable at any time. The stack on the other hand records references to the DTI instances of tapes that have been superseded in a nested environment. As can be seen from Figure 4.1, an additional DTI instance is available. This special instance (cDTI) stores a copy of the control information of the current tape and is used during the whole computations. In the case that a context switch occurs during a nested taping or evaluation process, all information contained in this structure is copied to the DTI instance of the superseded tape. Afterwards, the information contained in the DTI instance of the replacing tape is copied into cDTI. For enabling this setting, a reference to the DTI instance of the current tape is stored as part of the global tape information. Context switches may occur as part of the routines *trace_on* and *trace_off*. In the first case, a reference to the DTI instance of the superseded tape is pushed onto the stack. Otherwise, the reference to the DTI instance of the previous tape is popped from the stack and is used to continue the work on this tape.

When evaluating tapes by applying the reverse mode of AD, ADOL-C allocates two basis vectors. One vector holds the derivative values and the other is used for restoring the intermediate function values. The allocation is repeated for every tape. Based on this setting, ADOL-C currently faces a significant challenge when determining derivatives in a nested environment: Function values that are stored via the stack all emanate from the same source, the store. However, in the restoring process activities are distributed across several tapes and, thus, function values may be stored in different vectors, accordingly.

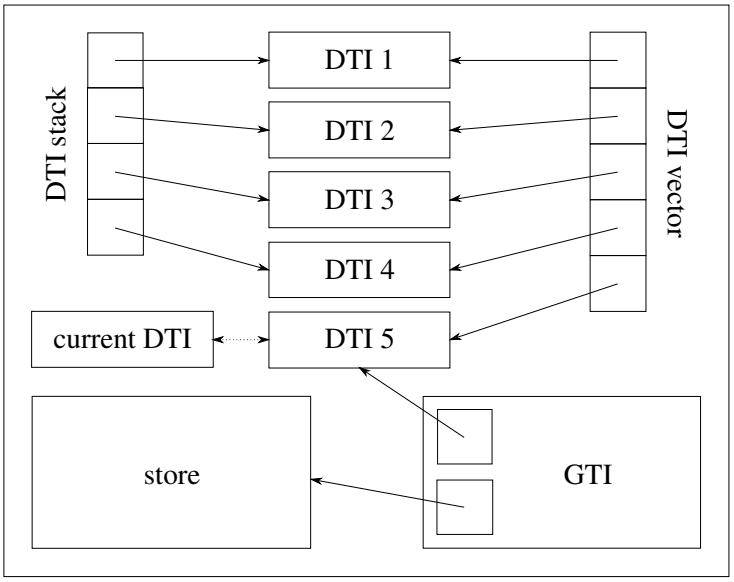


Figure 4.1: New tape management of ADOL-C

This induces the possibility of restoring function values by the correct tape but in the wrong vector. Example 4.2 depicts a corresponding situation where two tapes are created in a nested manner and the corresponding derivation process yields incorrect results due to a wrong restore of a function value. There, Stack 1 is associated with Tape 1 and Stack 2 with Tape 2, accordingly. In contrast, the store is

EXAMPLE 4.2

Tape 1	*Tape 2*	*Store*	*Stack 1*	*Stack 2*
trace_on(1, 1);				
...		$v1 = 2.0$	$*$	
$v2 = v1 * v1$			$*$	
...			$*$	
⟶				
	trace_on(2, 1);			
	...			$*$
	$v1 = 1.5;$	$v1 = 1.5$		2.0
	...			$*$
	trace_off();			
⟵				
...			$*$	
trace_off();				

shared by all of them. Performing a reverse mode differentiation, two vectors $ST1$ and $ST2$ that hold the function values corresponding to the individual tapes are created. Initially, the value 1.5 would be assigned to the variable $v1$ contained in both vectors. Evaluating the tapes and restoring function values using the stack, $v1$ contained in $ST2$ would be set to 2.0. This step yields the correct value for Tape 2, as can be seen from the evolution of the store. However, the value of $v1$ contained in $ST1$ is not corrected. As soon as the evaluation of Tape 1 reaches the representation of $v2 = v1 * v1$, the derivative computation uses the wrong value 1.5 instead of the correct 2.0. Therefore, it is necessary to adjust the internal restoring strategy of ADOL-C to obtain the correct derivative values. This can be achieved by using the same vector for restoring the function values at least as long as the computation is performed

within a nested environment. All tapes automatically benefit from restoring a function value, accordingly. In this case, the size of the store that has been reached at the completion of the outermost tape can be utilized in the allocation of the vector that holds the function values in the reverse mode.

Taking these facts into account, the derivation process may be nested and will nevertheless yield correct results. Then advanced techniques like checkpointing and fixed point iterations can be exploited, significantly reducing storage and runtime. However, this always requires user steering. Facilities that enable ADOL-C to automatically reduce tape sizes are described in the following subsection.

4.1.3 Augmented data type & activity tracking

The basic idea of activity tracking and its application to operator overloading based AD as described in the Sections 3.2.3 and 3.2.4 has been implemented in two phases. Forward activity tracking has been incorporated into the taping facilities of ADOL-C as result of the work for this thesis. In contrast, routines enabling optimizations in the opposite manner are available as part of the tape evaluation drivers. For practical reasons, only the variant of the reverse technique has been implemented that does not result in an adjusted tape.

To enable the activity tracking during the taping process, the meaning of the location stored within the augmented variables is extended. Now, the location is not only used to address individual values within the store but is also used as index into the field of state information. This field that is referred to as *state_vector* evolves in the same way as the store. In particular, if the size of the store is doubled the size of the state field is doubled too. Possible values stored within the state field are 0 – variable not varied – and 1 – variable varied. These values are stored using the C/C++ data type `char`.

The two different numerical values that are stored for each variable are also addressed using the single location. For this purpose two types of stores are available that are located one after another. These two fields are referred to as *store_varied* and *store_unvaried*. They are always of the same size and are allocated and deallocated together. Figure 4.2 summarizes the relations between an augmented variable and its location on the one hand and the two stores and the state vector on the other hand.

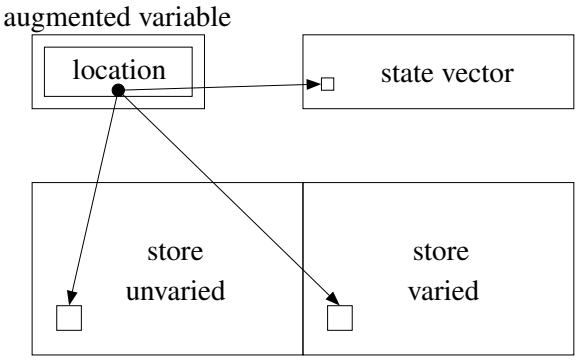


Figure 4.2: Relationship between the ADOL-C provided augmented variable, its location, the two stores and the state vector

For managing the stores, ADOL-C uses a global counter called *ADOLC_NUM_ALIVE* that stores the number of augmented variables that are currently used in the program. In addition, two pointers to the stores and a variable storing the store size are utilized. When allocating a new augmented variable, ADOL-C increments *ADOLC_NUM_ALIVE* and checks if more values have to be retained than can be managed by the store. In this special case, the store must be extended. Accordingly, ADOL-C doubles the stores in the following way. A new vector is allocated in the dimension of four times the current store size. If this allocation succeeds, two copy processes are invoked. One process copies the values

from *store_unvaried* to the beginning of the new vector. The second process transfers the values from *store_varied* to the second half of the new vector. By doing so, the second and the fourth quarter of the new vector remain unchanged. Finally, the memorized store size is doubled, the old stores are deallocated within one call of the appropriate routine of the operating system and the two pointers to the stores are updated.

Addressing the correct numerical values associated with the augmented variables can now be accomplished by simple calculations. For this purpose, a preprocessor macro `STORE` is defined for addressing the correct value in dependence of the current activity state. The definition of the macro is given below using C/C++ syntax.

```
#define STORE(X) store_unvaried[state_vector[(X)] * store_size + (X)]
```

Example 4.3 presents the definition of the overloaded multiplication operator provided by the *adouble* data type that uses this macro. The first step within the overloaded multiplication operator (Line 2) is

EXAMPLE 4.3

```
 1    adouble adouble::operator * (const adouble &arg) {
 2       locint location = next_loc();
 3       if (!state_vector[loc] && !state_vector[arg.loc])
 4          state_vector[location] = 0;
 5       else {
 6          state_vector[location] = 1;
 7          if (state_vector[loc] && state_vector[arg.loc]) {
 8             put_op(mult_a_a);
 9             put_loc(loc);
10             put_loc(arg.loc);
11             put_loc(location);
12          } else {
13             if (state_vector[loc]) {
14                put_op(mult_d_a);
15                put_loc(loc);
16                put_loc(location);
17                put_val(store[arg.loc]);
18             } else {
19                put_op(mult_d_a);
20                put_loc(arg.loc);
21                put_loc(location);
22                put_val(store[loc]);
23             }
24          }
25          write_scaylor(store_varied[location]);
26       }
27       STORE(location) = STORE(loc) * STORE(arg.loc);
28       return location;
29    }
```

to acquire an unused location. It is stored in the variable location of type `locint` that is ADOL-C's location type. Store adjustments that may be necessary due to the new location are handled within the call to `next_loc()`. Thereafter, it is checked if the two arguments are of *unvaried* state (Line 3) and if so, the state of the result variable is set to *unvaried*, too (Line 4). In any other case, the results state

is set to *varied* (Line 6). Only then, the taping process is invoked. Different internal representations are created depending on the specific constellation of the arguments *varied* properties. If both arguments are *varied*, a multiplication of two variables is represented. The arguments and the result are identified by their locations in that case (Line 8 through 11). If only one argument is *varied*, a multiplication of a variable and a constant is represented. In that case, only the location of the specific argument and the result is added to the tape. In contrast, the value of the *unvaried* argument is used. The two possible constellations are represented by Line 14 through 17 and Line 19 through 22, respectively. Within the process of creating the internal representation, the three ADOL-C-provided functions `put_op()`, `put_loc()` and `put_val()` are utilized to store either an operation code, a location or a constant value onto the appropriate sub-tape. By calling the `write_scaylor()` function of ADOL-C (Line 25), the value of the result variable is written onto the stack before it is changed. Taking the value from the *store_varied* ensures that the value of the variable represents their last varied state. Finally and independent of the *varied* state of the result, the original meaning of the operation is realized by computing the multiplication of the variable values. Here, the `STORE` macro is applied to increase the readability and maintainability of the code (Line 27). The overloaded operator is completed by returning the location of the result, implicitly constructing the augmented result variable by calling the appropriate constructor.

Reverse state tracking in ADOL-C

The reverse state tracking as discussed in Subsection 3.2.4 has been implemented only partially. In particular, only the drivers for the first order reverse tape evaluation have been adapted. For complexity reasons, only the special version has been implemented, that does not result in an adjusted tape. Rather, the state tracking results in the skipping of unneeded derivative computations and counting of the number of relevant operations. The second version with tape adjustments is principally also within reach but requires much more fundamental changes of ADOL-C. For this reason, it has been decided not to include this task into the scope of this thesis. It should however be addressed in a later project.

Changes that are necessary to enable the reverse state tracking are comparable to the forward variant. First, a new vector containing the state information must be provided. Its elements are again addressable using the locations stored within the tape. Based on this information, the *useful* property of the result can be determined. If the state is not set, all derivative instruction belonging to the handled operation are simply skipped. Furthermore, a global variable that holds the number of skipped operations is incremented. This counter and runtime measurements may later be interpreted as done in Subsection 4.2.1.

4.1.4 Facilities enabling parallel derivation

ADOL-C has been developed over a long period of time under strict sequential aspects. Although the generated tapes have been used for a more detailed analysis and the construction of parallel derivative code, e.g., [Bis91], ADOL-C could hardly be applied out of the box within a parallel environment, so far. The most convenient chance in this context is dedicated to the use of ADOL-C in a message passing system based on distinct processes for all cooperators. This requirement is fulfilled by, e.g., MPI. Due to separated address space and the realization of cooperators as processes of the operating system, the ADOL-C environment is multiplied. In particular, all control variables used in ADOL-C are available within each process exclusively. From the users point of view, only two conditions must be met to allow a successful application of ADOL-C. Firstly, the uniqueness of the created tape name must be ensured. This can be achieved by carefully choosing the tag, e.g., in dependence of the process' rank. Secondly, it must be considered that data transfer between the working processes is not reflected by the internal representation crated by ADOL-C. Besides of these restrictions, ADOL-C may be applied as usual. This allows to compute derivatives for functions that implement data partitioning techniques, especially when only a limited degree of communication is necessary.

Many parallel applications rely on a high amount of synchronization to communicate computed information at given points among involved cooperators. In a message passing environment this would also means to invoke more or less expensive transfer routines. Therefore, such applications are typically parallelized for a shared memory environment using OpenMP. Within the scope of the work for this thesis, extensive enhancements have been added to ADOL-C that allow the application in such cases. Originally, the location of an augmented variable is assigned during its construction utilizing a specific counter. Creating several variables in parallel results in the possibility to loose the correctness of the computed results due to a data race in this counter. Initial tests based on the protection of the creation process by use of critical sections yielded unambiguous facts. Even when using only two threads in the parallel program, runtime increased by a factor of roughly two rather than being decreased. For this reason, a separate copy of the complete ADOL-C environment is provided for every worker thread, as described in Subsection 3.3.3.

In addition to this decision, another issue had to be answered. As already identified by G. M. Amdahl [Amd67], every parallel program possesses a certain fraction that can only be handled serially. In many situations not only the parallel part of the function is object to the derivation efforts but also the serial parts. This entails the question of how to transfer information between the serial and parallel program segments and vice versa.

From serial to parallel
: Data transfer is this direction can be performed quit easily. For all variables alive at the moment when the parallel region starts, a copy may be created for each thread.

From parallel to serial
: This is the more difficult direction as it requires to decide which values from which thread should be copied to the serial part. Furthermore, the handling of variables created within the parallel part must be solved.

For the current implementation the following decisions have been made.

- The handling of parallel regions by ADOL-C comprises only augmented variables but not user variables of standard data type.
- Control structures utilized by ADOL-C are duplicated for each thread and, with exception of the global tape information, are default initialized during the first creation of a parallel region. The values of these control variables are then handed on from parallel region to parallel region.
- For performance reasons, two possibilities of handling the global tape information have been implemented. In the first case, control information including the values of augmented variables are transferred from the serial to the parallel variables every time a parallel region is created. Otherwise, this process is invoked only during the creation of the first parallel region. In either case, the master thread creates a parallel copy of the variables for its own use, too.
- No variables are copied back from parallel to serial variables after completion of a parallel region. This means, results to be preserved must be transferred using variables of standard data type.
- The creation or destruction of a parallel region is not represented within the initiating tape. Coupling of serial and parallel tapes must therefore be arranged explicitly by using the construct of external differentiated functions.
- Different tapes are used within serial and parallel regions. Tapes begun within a specific region, no matter if serial or parallel, may be continued within the following region of the same type.
- Tapes created during a serial region can only be evaluated within a serial region. Accordingly, tapes written during a parallel region must be evaluated there.
- Nested parallel regions are not supported and remain object to later enhancements.

All in all, the described facts result in augmented variables with a special property that depends on the specific handling of the global tape information. With the start of a new parallel region either a *threadprivate* or a *firstprivate* behavior, respectively, [DM98] is simulated. This means that the value of the augmented variable is taken either from the previous parallel region or from the serial region, respectively. In either case, the value used within the parallel region is invisible from within the serial region.

Initializing the OpenMP-parallel regions for ADOL-C is only a matter of adding a macro to the outermost OpenMP statement. Two different macros are available that are only different in the way the global tape information are handled. Using `ADOLC_OPENMP`, these information including the values of the augmented variables are always transferred from the serial to the parallel region. In the other case, i.e., using `ADOLC_OPENMP_NC`, this transfer is performed only with the encountering of the first parallel region. An application that benefits from the avoided copy effort is described in Subsection 4.2.3. Due to the inserted macro the OpenMP statement has the following structure:

```
#pragma omp ... ADOLC_OPENMP   or   #pragma omp ... ADOLC_OPENMP_NC
```

The remaining source code representing the function does not need to be changed. However, appropriate actions may be necessary to combine the evaluation of the created tapes in meaningful way.

Applying the newly created facilities of ADOL-C, derivatives for more sophisticated functions can be computed. Three examples, corresponding runtime information and their interpretation are presented in the following section.

4.2 Applications & numerical results

Numerical results that allow to evaluate the techniques that are described in Chapter 3 have been gathered using examples of different complexity and with different properties. First, the possible advantages of checkpointing facilities are demonstrated by means of the optimization for an industrial robot. The application of state-tracking facilities is demonstrated thereafter with the shape optimization of an airfoil. Finally, the parallelization of derivative information is examined using the numerical expensive time propagation of a quantum plasma. This concluding example also demonstrated in which way the new advanced techniques can be combined since the successful handling of the derivation is only possible by applying both the checkpointing facilities and the concept of external differentiated functions.

4.2.1 Industrial robot

The numerical example that serves to illustrate the runtime effects of the checkpointing procedure is an industrial robot as depicted in Figure 4.3 that has to perform a fast turn-around maneuver. This kind of robot is typically used in the automobile industry performing spot welding tasks. It features a low power consumption and allows to move workpieces with a maximum weight of 200 kg at a speed of up to 5 m/s. Angular coordinates $q = (q_1, q_2, q_3)$ that are associated with the robots joints are utilized in the modeling of the system. Figure 4.3 clarifies the meaning of q_2 and q_3 whereas the remaining coordinate q_1 refers to the angle between the base and the two-arm system. The robot is controlled via three control functions u_1 through u_3 that denote the respective angular momentums applied to the joints (from bottom to top) by electrical motors. Minimizing the energy-related objective

$$J(q,u) = \int_0^{t_f} [u_1(t)^2 + u_2(t)^2 + u_3(t)^2]\, dt$$

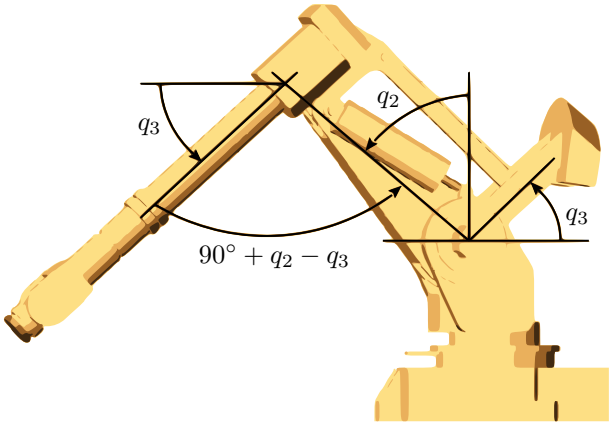


Figure 4.3: Industrial robot ABB IRB 6400

where the final time t_f is given, forms the control problem that shall be considered. The robots dynamics obeys a system of three differential equations of second order:

$$M(q)\,\ddot{q} = v(q,\dot{q}) + w(q) + \tau_{\text{friction}}(\dot{q}) + \tau_{\text{reset}}(q) + u \tag{4.1}$$

where $M(q)$ is a 3×3 symmetric positive definite matrix containing moments of inertia, called a generalized mass matrix. Thereby, centrifugal and Coriolis force entries are represented by the vector v and w contains the gravitational influence. Forces that are induced by dry friction and reset forces are modeled by means of τ_{friction} and τ_{reset}, respectively. For a complete description of the motion, see [BK03].

The robots task to perform a turn-around maneuver is expressed by means of initial and terminal conditions as well as control and state constraints. A state/control solution (x_0, u_0) for the inherent optimal control problem may be computed in advance, i.e., before performing the motion. In practice however, perturbations $p = (p_1, p_2, p_3, p_4)$ to various components of the equation system may occur. Thereby, p_1 through p_3 model perturbations of the angular coordinates, whereas p_4 characterizes changes in the tool weight that enter into (4.1) at various points. In an unperturbed system each component of p takes the zero value. In real-time applications as the given robot, computing a new optimal solution (x_p, u_p) for a perturbed system is often too time-consuming. Instead, parametric sensitivities, i.e., dx_0/dp and du_0/dp, are used to approximate the new optimal state trajectory x_p and the new optimal control u_p according to

$$x_p \approx x_0 + \frac{dx_0}{dp}\triangle p, \qquad u_p \approx u_0 + \frac{du_0}{dp}\triangle p.$$

Thereby, $\triangle p = p - p_0$ is utilized. The required parametric sensitivities may be computed in advance along with the solution for the unperturbed system. For further details on the approach, see [GW03]. However, for illustrating the runtime effects of the checkpointing facilities integrated in ADOL-C, only the gradient computation of $J(q,u)$ with respect to u is considered throughout the remainder of this subsection.

To compute an approximation of the trajectory x, the standard Runge-Kutta method of order 4 is applied for the integration. This results in about 800 lines of code. Due to ratio of input and output variables, the reverse mode of AD is indicated for computing the required derivatives. Thereby, the iterative nature of the Runge-Kutta method allows to apply and benefit from the checkpointing technique. For details on the checkpointing interface provided by ADOL-C, see [KW06].

Function and derivative computations were performed using an AMD Athlon64 3200+ (512 kB L2-cache) and 1GB main memory. The resulting averaged runtimes in seconds for one gradient computation are shown in Figure 4.4, where the runtime required for the derivative computation without checkpointing,

i.e., the basic approach (BA), is illustrated by a dotted line. In contrast, the runtime needed by the checkpointing approach (CP) using $c = 2, 4, 8, 16, 32, 64(, 128, 256)$ checkpoints is given by the solid line. To illustrate the corresponding savings in the memory requirements, Table 4.1 shows the tape sizes for the basic approach and the tape and checkpoint sizes required by the checkpointing version, respectively. The tape size for the latter varies since the number of independents is a multiple of the number l of discretization steps used in the approximation of x and u.

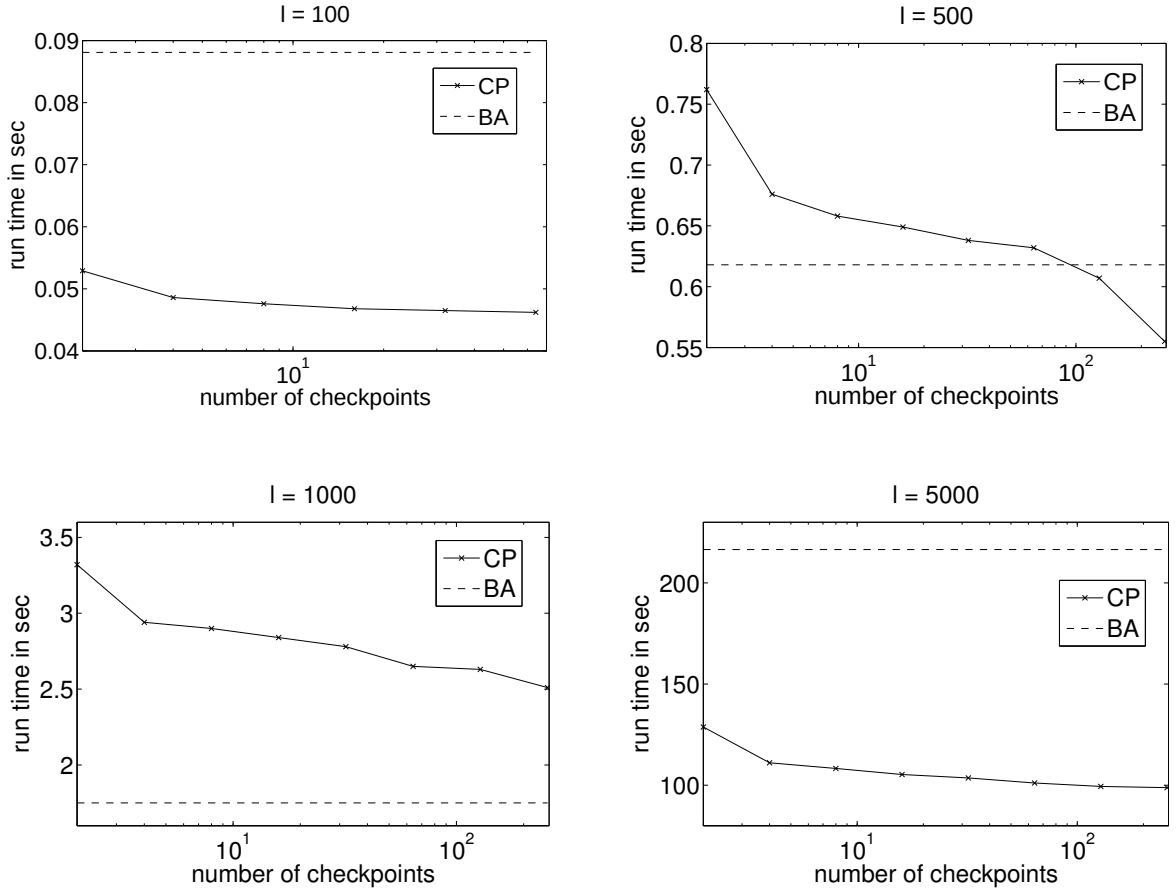


Figure 4.4: Comparison of runtimes for $l = 100, 500, 1000, 5000$

One basic checkpointing assumption, i.e., the more checkpoints are used the less runtime the execution needs, is clearly depicted by case $l = 1000$ in Figure 4.4. The smaller runtime for the basic approach completes the setting. Analyzing the provided memory architecture and comparing against the required tape and stack sizes clearly reveals the underlying problem. In the sum, the tape and even only one checkpoint cannot be hold within the L2-cache at the same time. According to its usage, the tape quite often displaces the checkpoint within the cache. Contrary, when accessing a checkpoint, the tape is partially displaced within the cache. Due to this behavior, only a small quantity of tape and nearly no checkpoint data can be used out of the cache. The missing data need to be loaded from main memory every time it is required. In presence of the checkpointing-induced recomputations of function values, no runtime reductions can be achieved, accordingly. All in all, these properties characterize situations that do not benefit from the checkpointing approach when considering runtime aspects. Nevertheless, the memory requirement is reduced significantly. Therefore, it is recommended to not use the technique for this constellations unless the saved memory is required for the computation of other information.

Analyzing the remaining three cases reveals an important consistency – the possibility to achieve a runtime improvement. Most clearly, this is obvious from the cases $l = 100$ and $l = 5000$, respectively. In

# time steps l	100	500	1000	5000
	without checkpointing			
tape size (Byte)	4.388.720	32.741.979	92.484.730	1.542.488.152
	with checkpointing			
tape size (Byte)	79.367	237.367	434.867	2.014.912
checkpoint size (Byte)	11.440	56.240	112.240	560.240

Table 4.1: Memory requirements for $l = 100, 500, 1000, 5000$

these situations a smaller runtime was achieved consistently even though checkpointing was used. These results are affected by an insight well known, i.e., computing from a level of the memory hierarchy that offers cheaper access cost may result in a significant smaller runtime. In the case of $l = 100$ checkpoints, the computation could be redirected from mostly main memory access to the L2-cache for a significant part of the checkpointing information. Furthermore, the cache handling entails the storing of the most recently used checkpoints within the cache whereas the seldom accessed checkpoints reside in main memory. A partially different situation is given by the use of $l = 5000$ checkpoints. There, the runtime reduction is not caused by a smart cache handling rather than by the avoidance of hard disk access. Using the basic approach, a significant part of approximately 50 percent of the tape must be reloaded from hard disk during the derivation process. Applying the checkpointing technique, this drawback could be completely eliminated, and computations where performed from main memory. In both cases, $l = 100$ and $l = 5000$, the savings in the memory access costs were high enough to compensate the recomputation of intermediate values, which were necessary due to the checkpointing. These results clarify the usefulness of the checkpointing technique. They also reveal that this approach is not restricted to the avoidance of hard disk access as it is typically used for.

The remaining case from Figure 4.4 ($l = 500$) depicts, to the estimation of the author, the most interesting situation of the robot example. There, a runtime reduction is not guaranteed. It is rather a result of carefully choosing the simulation criteria to achieve the desired profit. For the different number of checkpoints utilized in this case, the tape and a small number of checkpoints can be hold within the L2-cache of the processor. Again, the caching strategies guarantee that the most frequently used checkpoints attain this advantage. It is then a question of the ratio between the rate of recomputation and the saving from the cheaper memory access if the basic approach can be outperformed. With increasing number of checkpoints, the recomputation rate drops significantly and the runtime benefits accordingly. As can be seen from Table 4.1, the runtime of the basic approach could be undercut using $c = 128$ and $c = 256$ checkpoints. Once more a reduction in runtime could be obtained although a significant higher number of operations were performed.

All in all, the robot examples clarifies the value of the checkpointing approach. It also becomes obvious that the methods control possibilities must be used carefully so that the technique can play to its strength.

4.2.2 Shape optimization of an airfoil

The second example that serves to illustrate the state-tracking approach derived in Section 3.2 is chosen from the field of aerodynamics and addresses the shape optimization of an airfoil with respect to given design parameters. Numerical calculations are dedicated to an inviscid RAE2822 airfoil, simulated for a flight at a Mach number of 0.73 and an angle of attack of 2°. Within the optimization process, the given initial shape of the airfoil is changed by a parametrized deformation using a design vector referred to as P. The aim of the optimization is to minimize the drag of the airfoil that depends on its shape, which itself can be described in terms of P. In this process, sensitivity information regarding the drag with respect to the design vector are used for the deformation of the airfoil. Accordingly, the whole

computation chain from the static initial shape to the drag coefficient is the object of the derivation. This chain is implemented using four different tools. First, the surface deformation using the current design vector P is computed by a tool called `defgeo`. Thereafter, the difference vector to the original shape is calculated utilizing the program `difgeo`. Based on this information, the grid that is used during the flow computation is adjusted. This is done by a tool called `meshdefo`. Finally, the drag resulting from the investigated shape is computed using the DLR's flow solver `TAUij`. The complete chain for computing the drag coefficient including the input and output of each step is depicted in Figure 4.5. All of the four

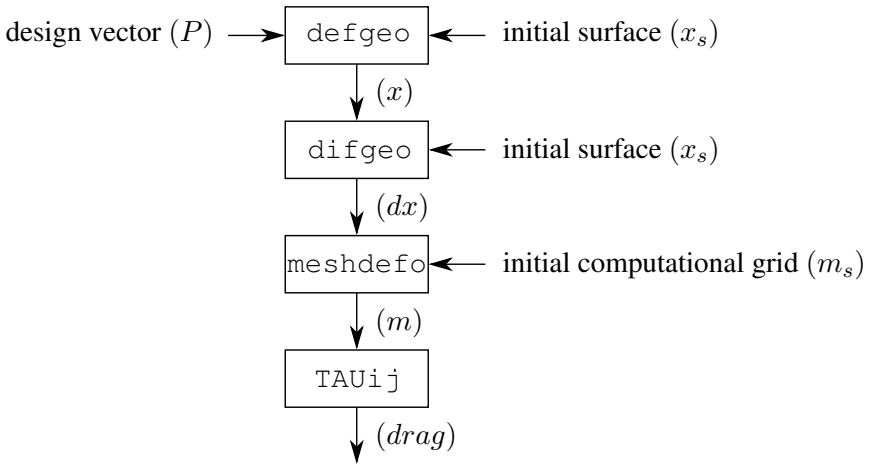


Figure 4.5: Drag of the RAE2822 airfoil: complete computation chain

programs are written in C. The `TAUij` code comprising about 6000 lines of code is a quasi 2D version of the `TAUijk` tool that itself is based on the well-known DLR `TAU` code. [GMWW07]

Due to the compatibility to the programming language C++, the derivation of the computation chain could be performed using the AD-package ADOL-C. Differentiating the four steps depicted in Figure 4.5 yields the formula

$$\frac{\partial drag}{\partial P} = \frac{\partial drag}{\partial m} \cdot \frac{\partial m}{\partial dx} \cdot \frac{\partial dx}{\partial x} \cdot \frac{\partial x}{\partial P} \,. \tag{4.2}$$

Therein, the first term on the right hand side represents the differentiation of `TAUij`, the second term the differentiation of `meshdefo`, the third term the differentiation of `difgeo` and the last term the differentiation of `defgeo`. In the computation chain yielding the draft coefficient, `difgeo` computes the difference $dx = x - x_s$. The differentiation of this step, i.e., $\frac{\partial dx}{\partial x}$ yields the identity matrix. Taking this fact into account, formula (4.2) can be reduced to

$$\frac{\partial drag}{\partial P} = \frac{\partial drag}{\partial m} \cdot \frac{\partial m}{\partial dx} \cdot \frac{\partial x}{\partial P} \,. \tag{4.3}$$

Out of the three partial derivatives, $\frac{\partial m}{\partial dx}$ and $\frac{\partial x}{\partial P}$ can be computed applying black box AD. In contrast, $\frac{\partial drag}{\partial m}$ contains a fixed point iteration that can be handled more efficiently using the corresponding technique described in Subsection 3.1.4. The layout and properties of the employed ADOL-C facilities are discussed in [SWGH06]. This paper also presents results of the complete optimization process. However, in the remainder of this subsection, the application of the state-tracking facilities of ADOL-C shall be illustrated. For this purpose, the complete optimization process is reduced to roughly one optimization step, i.e., the computation of the drag coefficient and the corresponding gradient. Furthermore, the inherent fixed point iteration is relaxed by removing the convergence control, i.e., a fixed number of steps is used for the original and the derivative fixed point iteration. Analyzing the properties of the state-tracking approach utilizing this framework yields some important advantages.

- Applying the state-tracking technique requires only an exchange of the ADOL-C library as the user code already contains all changes to benefit from ADOL-C.

- The results produced by the state-tracking code can easily be validated using the original code and its intermediate results.
- Significantly less runtime must be invested but the gathered results concerning the state-tracking nevertheless reflect the achievable benefits to the original program.

All computations have been performed on the *SGI ALTIX 4700* installed at the Center for Information Services and High Performance Computing (ZIH), TU Dresden. The characteristics of this system are summarized in Table 4.2.

Processors	1024 $\times$ Intel Itanium II Montecito @ 1.6 GHz (Dual Core) 2 $\times$ 9 MB L3-cache each
Peak performance	13.1 TFlop/s for the whole system
Main memory	4 GB per core and 6.5 TB total
Interconnection	NumaLink4 (SGI proprietary)

Table 4.2: Characteristics of SGI ALTIX 4700 as installed at the ZIH, TU Dresden

Computing the derivatives used in an optimization step based on the relaxed fixed point iteration requires ADOL-C to maintain three tapes. The memory requirements and runtimes of one optimization step are summarized in Table 4.3. All computations are based on the grid resolution of 321 $\times$ 65 points. The first observation that arises from Table 4.3 classifies the impact of the tape size on the overall memory requirements. As can be seen, the total tape size forms the major part of the program memory. Obviously, reducing the tape sizes is the best way of controlling the overall memory requirements. Using the state-tracking approach, an average reduction of about 13 % can be achieved for the tape size and about 11 % overall. The difference between these two values results from the additional memory that is related to the maintenance of the state information.

Analyzing the observed runtimes allows a more detailed view on the advantages and disadvantages of the state-tracking approach in its current implementation. Due to the inherent fixed point iteration, which presents the most expensive part of the function, the handling of Tape 2 is of special interest. In contrast, the runtimes for the 10000 iterations of the original fixed point iteration are not compared against each other since this code is based on the standard `double` data type. Therefore, the time difference is dedicated to the runtime environment. Comparing the reduction of the size of Tape 2 and the reduction of the runtime when evaluating it reveals an interesting but nevertheless not unexpected result: The runtime decreased by a factor that considerably exceeds the tape size reduction. This is caused by the replacement of nonlinear operations from the original program with linear operations in the tape that can occur when not all arguments are in the *varied* state. During the tape evaluation, less expensive formulas can be exploited to derive these linear operations. Additionally and truly surprising, also the taping time could be reduced by applying the state-tracking approach. This means that the definitely expensive checks for the state of a variable are more than compensated by the avoidance of the taping routines. However, it seems reasonable that a considerably reduction must be achieved to allow this effect. Whenever no or only a small reduction in the tape size is possible, the taping time is likely to increase. Considering the creation and evaluation of the remaining tapes, i.e., Tape 1 and 3, yields a result similar to the observations for the fixed point iteration. Again, a runtime reduction could be achieved for these calculations, which provide the major part of the remaining derivative computations that are referred to as “Minor calculation” in Table 4.3.

The higher effort of the overloaded operators that include state-tracking can clearly be seen from the evolution of the runtime that is necessary to perform the initialization of the drag computation. During the initialization phase specific computations are performed using `double` and `adouble` variables. However, no tape is created during this step. As can be seen, the runtime increases significantly when switching from the standard version to the state-tracking. Two major consequences result from this ob-

	Standard version	State-tracking	Reduction
Tape sizes			
Tape 1	$\approx$ 90 MB	$\approx$ 80 MB	$\approx$ 11 %
Tape 2	$\approx$ 2330 MB	$\approx$ 2030 MB	$\approx$ 13 %
Tape 3	$\approx$ 2380 MB	$\approx$ 2040 MB	$\approx$ 14 %
Total	$\approx$ 4800 MB	$\approx$ 4150 MB	$\approx$ 13 %
Program memory	$\approx$ 4950 MB	$\approx$ 4400 MB	$\approx$ 11 %
Runtimes			
Initialization	84 s	117 s	$\approx$ -39 %
Fixed point (10000 iterations)	1981 s	1997 s	–
Creation tape 2	15 s	13 s	$\approx$ 13 %
Reverse (2000 $\times$ tape 2)	18114 s	14932 s	$\approx$ 18 %
Minor calculation	61 s	56 s	$\approx$ 8 %
Program	20255 s	17115 s	$\approx$ 15 %

Table 4.3: Measurements for one step of the optimization of the RAE2822 airfoil

servation. Firstly, from the users view, it should be ensured that computations on `adouble` variables are only performed during a taping phase, especially when applying state-tracking. Secondly, addressing the internals of ADOL-C, the implementation of the state-tracking should be analyzed regarding the non-taping mode. It might be possible to reduce the effort in this mode and complete the overall positive results of the state-tracking technique. In summary, considering the whole computation chain, a significant reduction of the program runtime could be achieved that confirms the potential of the technique.

Reductions similar to the forward state-tracking can also be expected from the reverse activity-tracking. Respective conclusions could be drawn from the use of an appropriate implementation. However, the latter only comprises a few selected operations and is therefore not applicable to the airfoil optimization program. It has mainly been implemented to approve the correctness of the algorithm derived in Subsection 3.2.4. The completion of the code and the integration into the other concepts developed for this thesis remain subject to future work. The necessary effort to perform this task is far beyond the scope of this thesis.

All in all, the state-tracking technique allows a significant reduction of the internal function representation used in the derivation of C/C++ codes. When applied carefully, it directly yields a runtime reduction. This is even true for programs that have been adjusted extensively for the use of standard operator overloading before, as can be seen from the airfoil optimization example. Furthermore, it allows to initially prepare a given code for a subsequent derivation without even thinking about activity evolution. Thereafter, more or less expensive optimizations may be used to identify calculations that does not need to rely on the provided augmented data types. With the state-tracking facilities, a technique hitherto exclusively dedicated to source-to-source transformation is now also available for operator overloading.

4.2.3 Time propagation of a 1D-quantum plasma

The last example discussed in this thesis is taken from physics and mainly demonstrates the parallel derivation of a given function. However, due to its complex structure it is also necessary to apply most of the techniques derived in Section 3, in particular, nested taping, external differentiated functions and checkpointing facilities. Only the combination of these techniques allows the derivation based on the reverse mode of AD for this example. For better comparison and to clarify the need of reverse mode differentiation, runtimes are compared against the fasted forward mode that can be provided by operator overloading, i.e., the tapeless forward mode. It shall be noted that the properties of the given application significantly influenced the layout of the parallelization techniques developed in Subsection 3.3. The

implementation of the function was performed by N. Gürtler for his diploma thesis [Gür06], and the differentiation was realized in a cooperation between the Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen and the TU Dresden. To the knowledge of the author, the parallel derivation of the given function including the coping with the high internal complexity and inherent challenges currently features uniqueness and establishes a new level of the application of operator overloading based AD.

Often referred to as the "forth" state of matter, the term plasma describes an (partially) ionized gas consisting of neutral and positively and negatively charged particles, i.e., atoms, electrons and ions. Thereby, electrons and ions appear with approximately equal charge density. According to some estimates, up to 99 % of the material in the visible universe is in the plasma state. Therefore, plasma research is especially relevant to astrophysics but also in the analysis of effects of smaller scale. The probably most common phenomena in the Earth's atmosphere that involves plasmas is lightning. Other natural plasmas occur, e.g., in the ionosphere and the magnetosphere. Plasma physics also plays an important role in many applications from human hand. The most noticeable representative is probably the attempt to gain control of the thermonuclear fusion. Any progress in this field marks a step towards a clean and nearly unlimited power supply. One of the major challenges in controlling fusion is the establishing of magnetic confinements to contain the plasma. Hence, understanding the correlations between the plasma and interacting fields is often the key component for analyzing plasma effects. [BG05]

Here, in contrast to classical plasma physics, the particles are represented by a N-particle wave function $\Psi(1,\ldots,N)$ and the system can be described by multi-particle Schroedinger equations. For reduction of the complexity, spin effects are neglected. A direct solution, however, is numerically highly expensive. Since an approximation is often sufficient to describe the physical behavior, the simulation is based on Quantum-Vlasov equations that neglect interchange and correlation effects. As an entry point into quantum plasma simulations and for reasons of complexity only the one-dimensional case is modeled. However, due to the necessary discretization in the order of N dimensions, the direct solution is still very expensive. For the analysis of expected values for many distributions, calculations based on a representative ensemble of quantum states are sufficient. Then the following equation system can be exploited:

$$\begin{aligned} \imath\frac{\partial\Psi_i}{\partial t} &= -\tfrac{1}{2}\frac{\partial^2\Psi_i}{\partial z^2} + V_i\Psi_i \qquad i\in[1,N] \\ \Delta V_i &= \Delta(q_e\Phi) = -4\pi\tau_i \qquad i\in[1,N] \\ \tau &= \tau_e + \tau_I \end{aligned} \tag{4.4}$$

There, $\imath$ is the imaginary unit, $\Psi_i(z,t)$ is the wave function of the particle i, V_i is the interaction potential of the ith particle, q_e is the electron charge, Φ is electrostatic potential and τ, τ_e and τ_I, respectively, are the charge density of the overall system, the electrons and the ions, respectively. Furthermore, τ_i represents the charge density τ of the overall system reduced by the charge density of the ith particle. Prior to the numerical simulation, (4.4) needs to be discretized in t and z using $t_n = t_0 + n\Delta t$ with $n = 1,\ldots,T$ and $z_j = z_0 + j\Delta z$ with $j = 1,\ldots,K$. Applying cyclic boundary conditions yields the sparse cyclic tridiagonal system

$$U_i^{+,n+1}\Psi_i^{n+1} = U_i^{-,n}\Psi_i^n. \tag{4.5}$$

For details on the discretization and the definition of the operator U, the reader is referred to [Gür06, Gür07]. With (4.5), a complete description of the time propagation of the discretized wave function Ψ is given. Therein, the term Ψ_i^n denotes the wave function of the ith particle at time n. Directly calculating the interaction potential V_i that is part of the operator U turns out to be difficult. This is due to the determination of the necessary boundary conditions for V_i, while cannot easily be found while keeping the potential V for the entire system cyclic. Rather, V_i is computed as the difference between the potential V of the complete system and the self-contribution of the particle. For details see [Gür06]. Taking all

computations into account, an equation set is formed, which must be solved for each wave function for each time step. This set is given in Figure 4.6, ordered according to the inherent dependencies. There, ω_P complies to the classical plasma frequency, L is the length of the simulation interval and N denotes the number of particles. Wherever necessary, an additional index j is used for a variable if it represents

No.	Equation
1	$M_V x_V = \omega_P^2 \left[1 - \frac{L}{N} \sum_{l=1}^{N} \lvert\Psi_l\rvert^2\right]$
2	$M_V y_V = u_V$
3	$V = x_V - \frac{v_V \cdot x_V}{1+v_V \cdot y_V} y_V$
4	$V_{i,j} = V_j - 2\pi q_e^2 \left[j\Delta z - 2\Delta z^2 \sum_{m=1}^{j-1} \sum_{k=1}^{m} \lvert\Psi_{i,k}\rvert^2\right]$
5	$Mx = U^- \Psi^n$
6	$My = u$
7	$\Psi^{n+1} = x - \frac{v \cdot x}{1+v \cdot y} y$

Figure 4.6: Equation set for one iteration and one particle

the value valid for the cell j. The equations summarized in Figure 4.6 are the result of transformations using the Sherman-Morrison formula and allow to compute the solution of the system on the basis of tridiagonal matrices. Accordingly, the definition of M_V, u_V and v_V as well as M, u and v satisfy:

$$U = M + u \otimes v \tag{4.6}$$

$$\text{diag}(1, -2, 1) + e_1 \otimes e_K + e_K \otimes e_1 = M_V + u_V \otimes v_V \tag{4.7}$$

Thereby, $\otimes$ denotes the tensor product. In each step of the time integration according to Figure 4.6, the potential V for the entire system is computed first. This involves the equations 1 through 3. To allow for different charge densities for each wave function, V is computed for each wave function separately. Using the value of V, the interaction potential V_i is determined subsequently according to equation 4. Thereafter, the actual propagation of the wave function is performed based on the equations 5 through 7.

The final step that follows the time propagation of the plasma computes the expected value $< \eta >$ of the particle density. Equation (4.8) presents the discrete version of this target functions.

$$< \eta > = \sum_{i=1}^{N} \sum_{j=1}^{K} z(j) \Delta z |\Psi_{i,j}|^2 \tag{4.8}$$

The reduction of the high amount of output information resulting from the time propagation to a single value allows an easier evaluation of the entire system.

Reconsidering Figure 4.6 for challenges resulting from the parallelization of the code reveals an important fact: Equation 1 requires read access to all wave functions Ψ_i at the current time step n. Since the time propagation is performed in situ, it must be ensured that no Ψ_i^n is replaced with Ψ_i^{n+1} before its value has been processed. Hence, it is necessary to synchronize the work among the threads. Due to this insight, the computations for one time step are split into two sections that are implemented using two separate loops. The first loop is responsible for the determination of potential V of the entire system and the calculation of interaction potential of each wave function. Thereafter, the second loop propagates the wave functions for one time step. Synchronization is achieved automatically due to implicit barrier at the end of each OpenMP parallelized loop.

With [Gür06], a code for simulating the time propagation of a one-dimensional ideal quantum plasma has been developed that is based on the equation described above. It creates the source of the differentiation efforts and features the program layout that is depicted in Figure 4.7. There, all parallelization

```
...
startup_calculation(..);
for (n = 0; n < T; ++n) {
   #pragma omp parallel
   {
      #pragma omp for
      for (i = 0; i < N; ++i)
         part1(..);
      #pragma omp for
      for (i = 0; i < N; ++i)
         part2(..);
   }
}
target_function(..);
...
```

Figure 4.7: Basic layout of the plasma code including parallelization statements

statements are already included. The layout is presented using C++ notation and the OpenMP statements are adjusted accordingly.

Simulation details

Simulations based on the given plasma code turned out to be extremely costly for a reasonably realistic plasma of at least 1000 wave functions. For the proof of concept of the code as well as its derivation a reduced constellation is however sufficient. The following conditions are met for all runtime measurements that are discussed in this subsection.

- number of wave functions $N = 24$
- simulation time $t = 30$, discretized with $T = 40000$ time steps
- length of the simulation interval $L = 200$, discretized with $K = 10000$ steps
- plasma frequency $\omega_P = 1.23$

All units are transformed to the atomic scale, see [Gür06]. Proper simulations require the study of several plasma periods that is, related to the electron charge density, given by $2\pi/\omega_P$. The most important runtime reduction, however, results from computing only the first 50 of the 40000 time steps. After this period, the correctness of the derivatives can already be validated and the characteristics of the runtime behavior that are of special interest are already fully visible. To preserve the numerical stability of the code, the time discretization is based on $T = 40000$ steps nevertheless.

Differentiation aspects

Before the actual derivation strategy can be determined, the given function must be analyzed regarding the number of input and output variables. Overall, the plasma code computes the expected value of the particle density using the given N wave functions and the two global parameters Δt and Δz. The wave

functions are given as normal distribution with an expected value z_i and a standard deviation σ_i. Initially, they satisfy the equation

$$\Psi_i = \frac{1}{\sqrt{\sqrt{\pi}\sigma_i}} e^{-\frac{(z-z_i)^2}{2\sigma_i^2}} e^{\imath k_i z}. \tag{4.9}$$

Accordingly, each wave function features an initial impulse k_i. All in all, this leads to $3N + 2$ input variables and one output variable. For an analysis of the physical dependencies as well as the quality of the discretization, derivatives of the single output with respect to all inputs may be of interest. Recalling the theory described in Chapter 2, a clear conclusion regarding the work mode of AD to be applied can be drawn: For the given ratio of input and output variables, derivatives should be computed using the reverse mode of AD.

As already discussed in Chapter 2, performing a reverse mode differentiation always entails the question regarding the management of the overwritten function values. Roughly speaking, about the same number of overwritten values must be handled as operations are performed during the execution of the function. Furthermore, when applying the operator overloading based AD-tool ADOL-C for the reverse differentiation, an internal function representation must be stored that also requires memory in a multiple of the operation count. Altogether, this creates a high memory demand that must be rated carefully. Considering the given 24-particle-plasma and the derivation of only one time step serially, the storage of about 2.4 GB data is required. Multiplying this value with the number of time steps to be performed, yields the impressive memory requirement of roughly 96 TB. This amount of data cannot be stored within the main memory of any existing computer known to the author. Scaling the plasma to a more realistic size of more than 1000 wave functions requires to adjust the memory guess accordingly. Furthermore, parallelizing the code such that each particle is processed by a different thread increases the memory demands once more. Though only the computations for one wave function must be represented by the created tape on each thread, certain operations at the beginning and the end of the tape must now be performed on each thread. Summing up over all threads leads to the mentioned increase in the memory requirements. In any case, the black box application of operator overloading based AD is not advisable or is even impossible. The only applicable solution to this challenge yields the first and most important demand on the derivation of the code – checkpointing techniques as discussed in Subsection 3.1.3 need to be incorporated.

Analyzing the parallelization aspects of the code reveals the second, important challenge. Applying ADOL-C results in the creation of internal functions representations that neither contain the loop structure nor the parallelization statements. A possible solution that may be exploited to overcome this situation is described in Subsection 3.3.5. However, this approach cannot be used directly due to the presence of the checkpointing facilities. To be more precise, ADOL-C currently considers the provided time step function to be of serial nature. This means in particular that if a parallel time step function is provided, the differentiation by use of ADOL-C would fail as representing parallel functions by a single tape is currently not possible. Hence, the time step function must be provided in a way that allows to influence the execution schema of the function itself as well as its derivation. Again, a technique developed in Section 3.1 answers the challenge. Using an external differentiated function within each time step, control is given to the user during the function calculation and the corresponding derivation. In this situation, the parallelization approach described in Subsection 3.3.5 is applicable. Combining all techniques for the given program and considering the execution using p processors yields $p + 3$ derivative contexts, both during the function evaluation and its derivation.

Function evaluation

C1 The outermost derivative context is used for the function initialization and the computation of the target function. All calculations are recorded onto tape 1. In this phase, the checkpointing facilities of ADOL-C are invoked causing the switch into context C2 and back.

C2 This context serves for performing the actual time propagation of the plasma. Thereby, $T - 1$ time steps are executed based on the original, non-augmented version of the time step function.

These computations are performed in parallel due to the layout of the provided code. No tape is created during this period. However, checkpoints are taken according to the steering of `revolve`. Finally, an augmented version of the time step function is executed for computing the last time step. Thereby, an internal representation, i.e., tape 2, is created. To prepare the parallel execution of this time step, an external function is called. This enforces another context switch.

C3 At this point of the program execution the ADOL-C-provided macro `ADOLC_OPENMP_NC` invokes a special internal routine that is responsible for the creation of p new derivative contexts. Each of them is associated to one of the involved working threads. Thereby, each thread is responsible for the propagation of one or several wave functions depending on the work load distribution. For this reason, the two inner loops depicted in Figure 4.7 are replaced by user guided codes that manage the assignment of wave function to threads.

* The term "*" denotes the remaining p inner contexts that exist concurrently. Within each context the assigned wave functions are propagated serially. Thereby, p new tapes are created whose identifier are chosen by the user.

Derivation

C1 The differentiation of the function is started with the outermost context C1. Applying the standard reverse mode drivers of ADOL-C, derivatives are computed using tape 1. This process is interrupted once the special operation signaling the context switch is reached. Then, control is given to context C2 for deriving the actual time propagation of the plasma.

C2 Based on the preliminary work done during the function evaluation, the derivation of the time propagation is performed under the steering of `revolve`. The actual layout of this derivation procedure depends on the considered number of time steps and number of checkpoints. An example is depicted in Figure 3.5. Parallelization of the forward sweeps is achieved either directly when executing the non-augmented version of the time step function or user guided as described for the function evaluation. In the latter case, additional contexts are created again. Due to the layout of the augmented version of the time step function, tape 2 mainly consists of the call of the external function. During the derivation using this tape, a context switch is enforced accordingly.

C3 This context is responsible for providing the parallel derivation environment and the subsequent assembling of the final derivatives of a time step. Similar to the function evaluation, the ADOL-C-provided macro `ADOLC_OPENMP_NC` is utilized to recreate the required p derivative contexts. Subsequent to the decomposition of this environment due to the completion of the inner computations, the AD-assembling phase is invoked.

* The p concurrent, innermost contexts are used to calculate preliminary derivatives for the tapes associated with each thread. Here, standard AD-technique is applied by calling the appropriate reverse mode driver of ADOL-C. These preliminary information create the base of the AD-assembling phase.

Most aspects of the context handling are hidden from the user and the appropriate actions are performed internally by ADOL-C. However, four different functions are required during the derivation process to enable the parallelization facilities. Along with the main program, they are depicted in Figure 4.8 using C++ notation. There, the call structure is partially represented by the plotted arrows. Note that an arrow does not necessarily represent a direct call between two functions as internal functions of ADOL-C may by executed in-between. Rather, an arrow connects the two points where the control leaves and reenters the user code, respectively. Considering the code structure presented in Figure 4.8 clarifies that the provided layout of the original code as given in Figure 4.7 was changed significantly. The smallest adjustments were necessary to provide the time step function `timestep(..)` based on non-augmented data types. In particular, only the outermost loop controlling the time propagation was removed. However, the numerical counterpart based on the `adouble` data type, i.e., `timestep_AD`, only contains the call of the external function, i.e., `ext_timestep_AD`. The latter firstly creates the

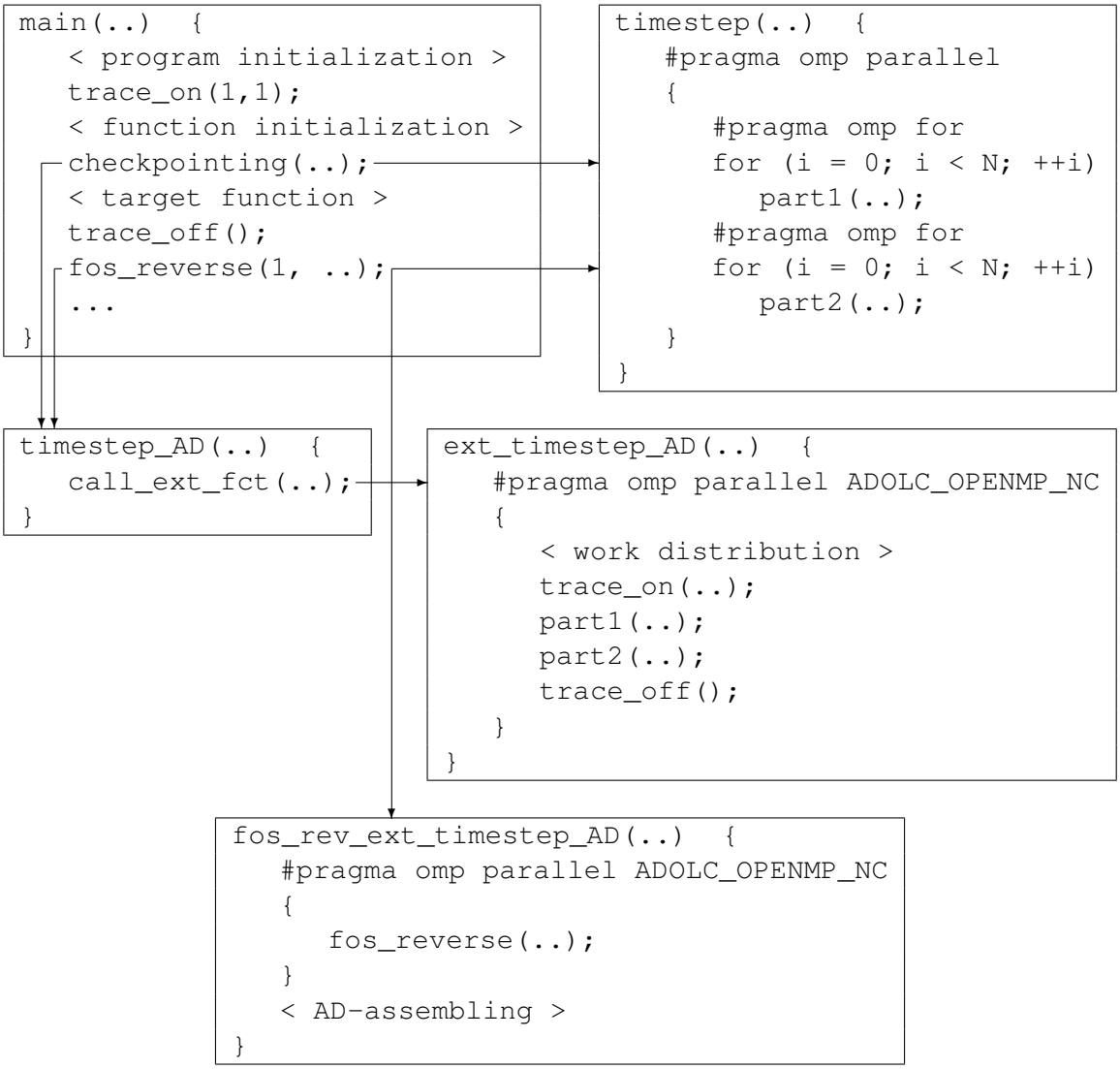


Figure 4.8: Derivation of the plasma code – function layout

parallel AD-environment as soon as the `ADOLC_OPENMP_NC` code is reached. Within this environment each participating thread determines a set of wave functions that it processes. Thereby, the applied distribution algorithm guaranties that each wave function is propagated only once and that load balance is achieved among all threads. Accordingly, the two main loops controlling the work distribution in `timestep` are not contained in `ext_timestep_AD`. Furthermore, the implicit barrier of the first loop that is used for the thread synchronization in `timestep` does not need to be replaced with an explicit barrier as the dedicated environment of each thread prevents race conditions. Compared to this setting, the layout of the derivation routine `fos_rev_ext_timestep_AD` is quite simple. After creating the parallel environment the tape generated in `ext_timestep_AD` is evaluated and the derivatives of the whole time step are assembled using the results of all threads. Synchronization efforts during the tape creation are not necessary for the derivation since each thread works in its own dedicated environment.

Due to the properties of the checkpointing facilities provided by ADOL-C, two work modes can in principle be utilized for the derivation of the time propagation. On the one hand, one may reuse the tape written for a specific time step as often as possible. Then creating a new tape is only necessary if a branch switch is detected during the evaluation of a tape. On the other hand, one may enforce the creation of a new tape every time the stack of overwritten function values is created. This results in a higher computational effort but also guaranties the correctness of the computed derivatives if a branch switch

results from calculations based on non-augmented data types. In the given example of the plasma code, an LU-decomposition with pivoting is performed that is based on non-augmented information. As soon as this information changes, a new tape must be created. However, due to utilized data type, such situations cannot be detected by ADOL-C. The restructuring of the code, such that changes in the pivoting are detectable from the tape, requires significant effort and has not been done so far. Therefore, all runtime measurements described in following subsection are based on a version of the plasma code that enforces the retaping of the time step function for every time step. It should be noted that once appropriate changes allow to avoid the strict retaping, the runtime behavior of the reverse mode differentiation will benefit accordingly.

Runtime results

All runtime measurements have been performed using the *SGI ALTIX 4700* system installed at the TU Dresden. The hardware specifications of the system are given in Table 4.2. For performance analysis the software *VNG – **V**ampir **N**ext **G**eneration* [BNM03, BN03] has been used. VNG was designed at the TU Dresden based on the experiences gained from the performance analysis tool *Vampir* [AHN+96]. It features a client-server architecture and allows to analyze large program traces in parallel. The tracing system recognizes both MPI and OpenMP directives and, therefore, is well-suited to analyze the runtime behavior of the plasma code.

Three different code versions for computing derivatives are analyzed in the remainder of this subsection. Two of them implement the reverse mode of AD including the described checkpointing and parallelization schema. The difference between these two codes emerges from the way that information is addressed within the parallel environment created by ADOL-C. First, a version was implemented that duplicates all data of the serial environment and stores it in appropriate vectors of p times larger size. Each of the p threads that lives in a parallel section may then access its data using its numerical identifier. However, this requires the `OMP_GET_THREAD_NUM` function to be called for each data access. The second approach that is applicable for reverse mode differentiation utilizes the `OMP_THREAD_PRIVATE` directive of OpenMP. Satisfying the required constraints, this directive allows to dedicate specific memory to threads exclusively. Furthermore, threads are not decomposed at the end of a parallel region. Rather, they are deactivated only and are reactivated at the beginning of the next parallel region. Memory that has been dedicated to threads is then accessible again and its contents is guaranteed to not be changed. The principle layout of the data access for these two implementations is depicted in Figure 4.9. There, an integer variable `example` is created for every thread. A unique value – the thread number – is assigned to it in the first parallel region. This value is then written out to the standard output stream in the second parallel region. On the left hand side of of Figure 4.9, the code version is depicted that requires the `omp_get_thread_num` function to be called in each parallel region at least once. In contrast, this function is only used for the creation of the unique value of the variable `example` on the right hand side. The specific implementation of `threadprivate` variables is up to the applied compiler and is hidden from the user. Two separate versions of the reverse mode code have been implemented for reasons of compatibility as not all used compilers handle the `threadprivate` directive correctly.

The third code version that serves the validation of the derivative values has been implemented to use the tapeless forward mode provided by ADOL-C. Thus, derivative values are computed during the evaluation of the function as described in Subsection 3.3.2. Detailed information on the tapeless forward mode provided by ADOL-C can be found in [KW07]. There, runtime observations and usage details are discussed, in addition. The correctness of the derivatives computed by this code has been validated against a code that is based on analytical differentiation. For information regarding the underlying mathematical approach as well as the analysis of the runtime behavior the reader is referred to [Gür07]. To allow a direct comparison between the versions of the plasma code, the forward mode based version applies the vector forward mode of AD. Thus, computing the full gradient requires $3N + 2$ directions to be propagated within the single forward sweep.

Using the thread number

```
int example[NUM_THREADS];

...
#pragma omp parallel
{
   int num = omp_get_thread_num();
   example[num] = num;
}
...
#pragma omp parallel
{
   int num = omp_get_thread_num();
   cout « example[num] « endl;
}
```

Using threadprivate variables

```
int example;
#pragma omp threadprivate(example)
...
#pragma omp parallel
{
   int num = omp_get_thread_num();
   example = num;
}
...
#pragma omp parallel
{
   cout « example « endl;
}
```

Figure 4.9: Layout of the internal data access for the two reverse mode implementations

All three code versions apply the same parallelization approach, i.e., every participating thread of the parallel environment performs all calculations for a specific number of the considered N wave functions. An equal distribution of the workload is tried to be achieved by the program logic. Nevertheless, situations may occur where the number of threads is not a factor of N and where load balance cannot be ensured, accordingly. By careful simulation design, this situation has been avoided. Figure 4.10 depicts the speedup and runtime results measured for the derivation of 24 wave functions. Considering

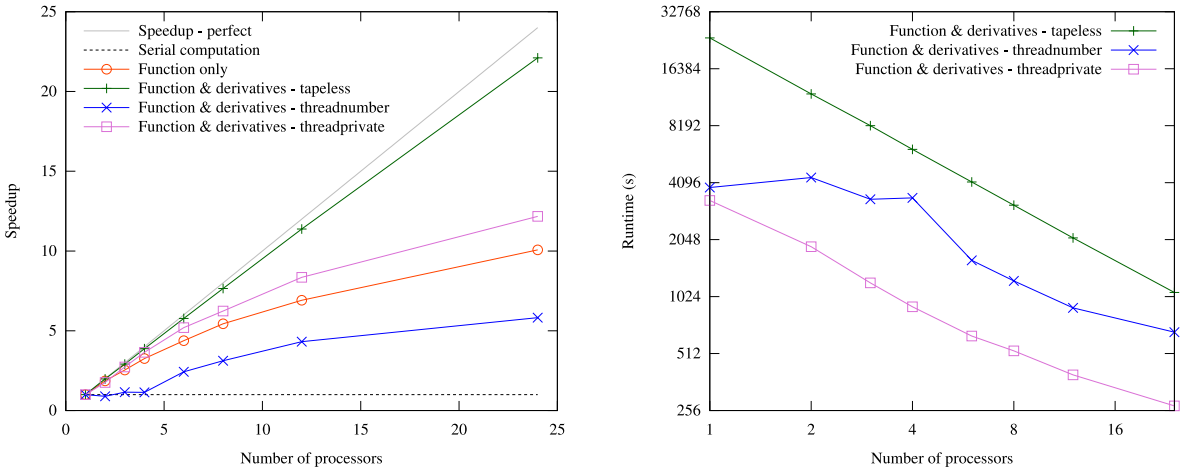


Figure 4.10: Speedups and runtimes for the parallel differentiation of the plasma code for $N = 24$ wave functions

the achieved speedups reveals an interesting result. With exception of the thread number based variant of the code, all derivative calculations achieved a higher speedup than the original code that they are based on. This allows the conclusion that, carefully implemented, the derivative calculations decrease the distracting effect of the necessary synchronization within the parallel environment. However, as can be seen from the code version that extensively uses the `omp_get_thread_num` function, this gain is by no way mandatory. Rather, the mentioned program version suffers from an unfavorable behavior of the considered OpenMP function. This is getting much clearer from the runtime results that reveal a

volatile evolution of the required execution time. Since the difference to the `threadprivate` based code results from the usage of the `omp_get_thread_num` function, the cause of the perturbation is identified.

Although investigating the evolution of speedups allows to predict the program behavior for upscaled tasks and parallel environments, it cannot be the only criteria when rating the applied derivation technique. In terms of speedup, the best results are achieved using the tapeless forward mode. However, the truly superior technique for the differentiation of the plasma code is revealed by the examination of the necessary program runtimes that are depicted in the right part of Figure 4.10. For further clarification, the runtimes of the tapeless forward variant and the threadprivate reverse variant are depicted and rated in Table 4.4. As can be seen, the tapeless version of the code requires significant more execution time.

# procs	Tapeless	Threadprivate	Ratio
1	23816 s	3299 s	7.2
2	12050 s	1878 s	6.4
3	8193 s	1210 s	6.8
4	6138 s	905 s	6.8
6	4128 s	634 s	6.5
8	3112 s	529 s	5.9
12	2029 s	395 s	5.1
24	1077 s	271 s	4.0

Table 4.4: Runtimes of the tapeless forward and the threadprivate reverse version of the plasma code

Hence, reverse mode differentiation is to be preferred for the plasma code. This strongly corresponds to the theoretical assumptions described in Chapter 2, i.e., reverse mode differentiation is suggested if the number of input variables significantly exceeds the number of output variables. Even the application of checkpointing strategies and, thus, the recalculation of function values does not change this fact. Up to the use of 6 processors, the ratio between the two corresponding runtimes is quiet stable and then drops for the benefit of the tapeless code.

To identify the relevant reasons for these results, more detailed information about the program behavior is required. For this purpose, the computational effort of the simulation has been reduced once more. Only eight particles have been computed within four iterations of the time propagation of the plasma. Furthermore, relevant information describing the runtime behavior has been gathered using the performance analyzing tool *VNG*.

Since automatic differentiation always computes derivatives for a given user code, understanding the properties of this code must be the basis of the analysis of its AD-counterpart. Figure 4.11 depicts the runtime behavior of the plasma code using a specific distribution of the wave functions that is referred to as the original particle sequence. It is characterized by its special starting arrangement of the wave functions that is given in terms of their peak positions.

$$1111 \quad 2222 \quad 3333 \quad 4444 \quad 5555 \quad 6666 \quad 7777 \quad 8888 \tag{4.10}$$

Wave functions and threads are associated from left to right, i.e., thread 0 handles the wave function with peak 1111, thread 1 handles the wave function with peak 2222 and so on. In the resulting *VNG*-images, thread 0 is always denoted by Process 0. Analyzing Figure 4.11, the sectioning of the time propagation loop into *part1* and *part2* is clearly visible. Furthermore, two important observations can be made. Firstly and dedicated to the strongly reduced number of time steps, the computation of the expected value of the particle density requires a significant fraction of the runtime. Even though its influence will decrease as the number of time steps increases, a parallelization of the target function should be

Figure 4.11: Undifferentiated version of the plasma code - original particle sequence

Figure 4.12: Undifferentiated version of the plasma code - particle one and four swapped

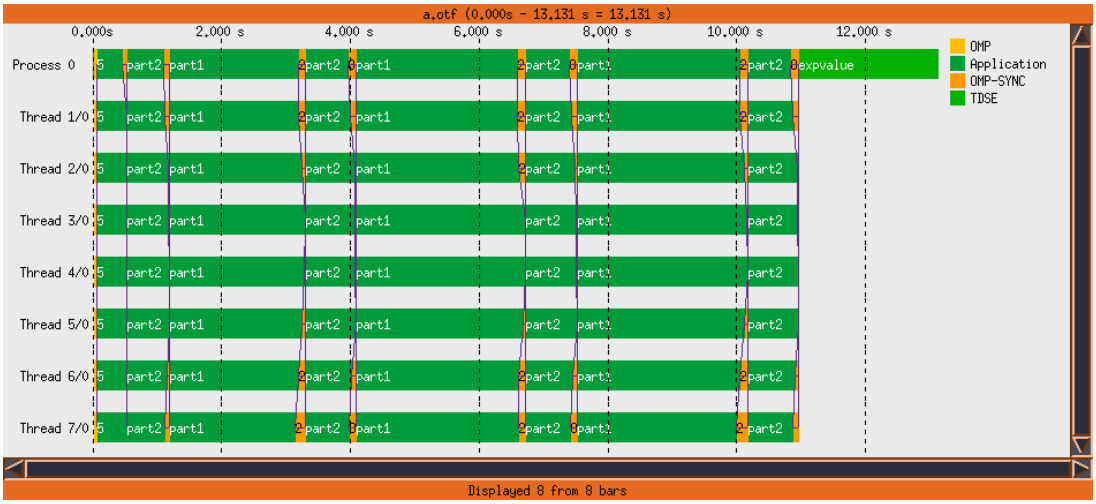


Figure 4.13: Differentiated version of the plasma code based on the tapeless mode of ADOL-C

considered. Thereby, tree-based reduction algorithms may be of interest. Secondly, an imbalance of the work load is observable among the involved processing elements. Even though the equation set depicted in Figure 4.6 suggest the same amount of work to be done for each wave function, the required runtime to perform these computations is different in practice. As a result, different waiting times - orange colored in Figure 4.11 - are required in the synchronization among all threads. These waiting times further reduce the possible speedup and should be minimized. Accordingly, the most important question is to determine if this behavior is algorithm or system dependent. For this purpose, a second particle sequence has been constructed that is given in (4.11). It is derived from the original sequence by exchanging the peak positions of the wave functions one and four.

$$4444 \quad 2222 \quad 3333 \quad 1111 \quad 5555 \quad 6666 \quad 7777 \quad 8888 \tag{4.11}$$

By comparing the Figures 4.11 and 4.12, the influence of the position of the wave function on the computational effort becomes visible. As can be seen, thread 0 (depicted by Process 0) and thread 3 not only swapped their handled particles but also swapped the computational effort for performing the time propagation. Hence, it can be concluded that the effort for the propagation of a specific wave function is mainly bound to its "position" in the simulation interval rather than being bound to the distribution onto the computer hardware. This property must be taken into account for all variants of derivative computations based on the given plasma code.

Regarding to the speedup diagram depicted in Figure 4.10, the tapeless derivation draws the highest benefit from the parallelization. It is therefore a good entry point for the analysis of the runtime behavior of the different differentiation approaches. As for the original function, the tapeless derivation has been performed for eight wave functions and four time steps. A corresponding visualization of the execution created using *VNG* is given in Figure 4.13. Comparing to Figure 4.11, the identical basic layout of the computation is clearly visible. However, a major difference is observable that also give reasons for the higher speedup. This is, a much more favorable ratio between computing time (dark green) and OpenMP synchronization time (orange) could be achieved due to additional effort introduced for the vector forward mode differentiation. Here, 26 directional derivatives have been computed for every operation of the original code that is based on the augmented data type `adouble`. Also clearly observable is the property of the differentiation that the additional effort is not equally distributed. In Figure 4.13, a significant higher ratio between the runtime of `part2` and `part1`, i.e., $ratio = TIME(part2)/TIME(part1)$, is depicted compared to the original function presented in Figure 4.11. This means that more complex computations are performed during `part2`, based on a higher fraction of non-linear operations. All in all, using the tapeless vector-forward mode provided by ADOL-C, the overall program structure is preserved but the parallelization potential is increased.

Reconsidering Figure 4.10, the results of the two reverse mode based versions of the differentiated plasma code remain for analysis. With exception of the variable access in the parallel environment, the two versions apply the same differentiation technique. Therefore, the lower speedup of the threadnumber-based version cannot result from properties of the derivation. For this reason, it is not considered any further and the version of the plasma code using threadprivate variables is analyzed in the remainder of this subsection, exclusively.

Allowing for the advantages of a graphical representation of a programs runtime behavior, the tool *VNG* has been utilized to create the illustration given in Figure 4.14. By comparing the layout of the original function with the tapeless forward differentiation considerable differences are observable. This is, a much higher fraction of the program is executed serially and, moreover, the parallel computations are often interrupted by serial calculations. The impact of these sequential phases on the speedup is the higher the shorter the parallel phases are. Improving the benefit that can be drawn from the parallelization of the code is thus be bounded to the reduction of the serial program parts. For determining relevant locations of the program that should be object to reduction or parallelization, a strict coloring scheme has been applied in Figure 4.14. All program parts that are either responsible for computing function values on standard data type or propagate derivatives are depicted dark green. Thereby, the green blocks

Figure 4.14: Threadprivate version of the differentiated plasma code – overview

at the time positions of about 5, 9, 12 and 15 seconds represent the reverse mode differentiation for a single time step. Further, it should be noted that the time propagation based on standard data types exhibits only subordinate impact. This is due to the quite small fraction of the overall runtime spent with this specific task. The propagation of wave functions based on standard data types is performed three times – three time steps at time position 2s, two time steps at time position 6s and one time step at time position 9s. Program sections that result from the creation of internal function representations by ADOL-C are blue colored. As discussed before, a strict retaping strategy is applied for the derivation of the plasma code. This is visualized by the corresponding four blue phases in the propagation of the wave functions over four time steps. The benefit of avoiding the strict retaping is thus clearly visible, knowing that the creation of the stack of overwritten function values using an existing tape is less expensive than computing derivatives using reverse mode AD. Although reusing tapes will significantly reduce the overall runtime, it does not allow to increase possible program speedups. Hence, a more detailed view on the serial program parts is required that are introduced due to the differentiation. A visualization of one such program section is given in Figure 4.15. There, the serial computations are depicted that are

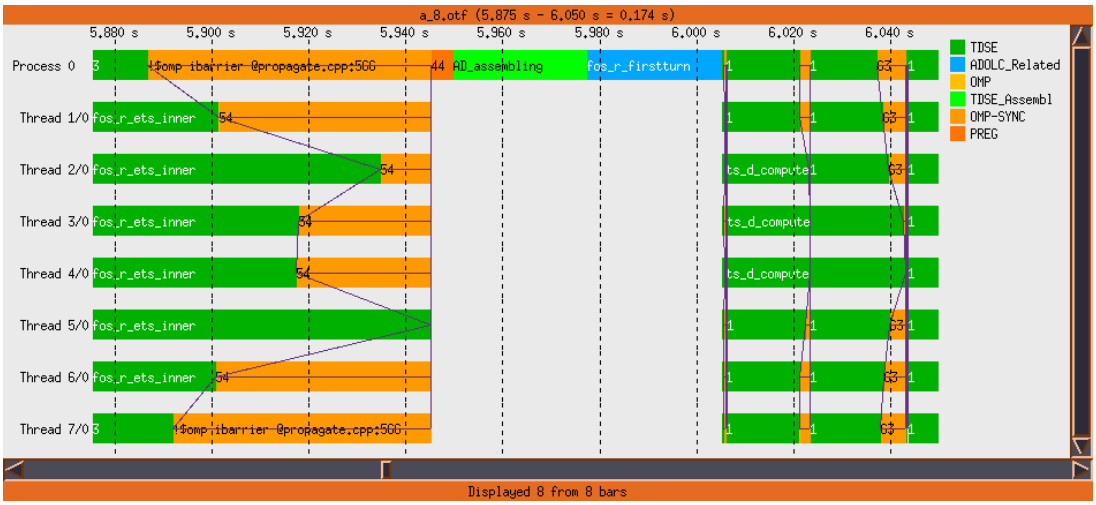


Figure 4.15: Threadprivate version of the differentiated plasma code – zoomed in

performed between the differentiation of the last time step and the reevaluation of the function from the last checkpoint. Comparing the layout of the reverse mode calculation (left part) and the time propagation

using the original user function (right part) reveals an interesting fact: In terms of wave functions, the highest effort for the propagation of function values does not necessarily causes the highest effort for the propagation of derivatives. This gets especially clear when comparing the runtime behavior of the threads three and four with the behavior of the threads 2 and 5. Even though this is an interesting result, it is not the key information that shall be taken from Figure 4.15. Rather, it shall be noted that the serial program part depicted there mainly is composed of two phases, the derivative assembling (light green) and the handling of the external function performing the differentiation of the next time step (light blue). The two of them offer optimization potential.

Initially, the derivative assembling phase has been designed to be performed purely serial. As described in Subsection 3.3.4, each participating thread owns a set of local derivatives that it works with throughout the parallel region. Once the region is completed, the master thread assembles the global derivative values by computing the sum over the corresponding local information of all threads. However, there is no need to perform this task in serial. Rather, the threads can be advised to handle specific, distinct subsets of the output data in parallel. An appropriate algorithm is given in Example 4.4.

EXAMPLE 4.4

```
double *global_der = new double[NUM]();
double *local_der[NUM_THREADS];
#pragma omp parallel
{
   int myID = omp_get_thread_num();
   int chunk_size = NUM / NUM_THREADS;
   int chunk_start = myID * chunk_size;
   int chunk_end = chunk_start + chunk_size;
   local_der[myID] = new double[NUM];
   < compute derivatives >
   double *tmp;
   for (int i = 0; i < NUM_THREADS; ++i) {
      tmp = local_der[i];
      for (int j = chunk_start; j < chunk_end; ++j)
         global_der[j] += tmp[j];
   }
}
```

There, it is assumed that the number of derivative values to be computed is a multiple of the number of threads. Using this kind of algorithm, a near ideal speedup can be expected. This way, the derivative assembling phase cannot prove to be the parallelization bottleneck when increasing the number of simulated wave functions.

Recalling the parallelization approach depicted in Figure 4.8 reveals the reason for the comparable high computational effort for handling the external differentiated function: Before actually performing the parallel taping or the parallel derivation of the provided time step function, the program must pass through two derivative contexts. The first context is given by the checkpointing algorithm itself. From within this context a second one is entered that allows the parallelization of the code. This basically means that the internal checkpointing functions call user wrapper functions that call the actually required work functions. Accordingly, the tape internally created by the checkpointing algorithm basically only contains the operation representing the external differentiated function. Significant effort is required to reevaluate or differentiate the complete tape that actually does nothing more than calling the appropriate parallelized user function. Instead of applying this strategy, the required functions can be called directly by the checkpointing facilities, thus avoiding the extra costs resulting from the internal tape handling.

Figure 4.16 presents the visualization of the runtime behavior of a program version that implements the parallel AD-assembling and features the reduced complexity of handling the parallelization. The most

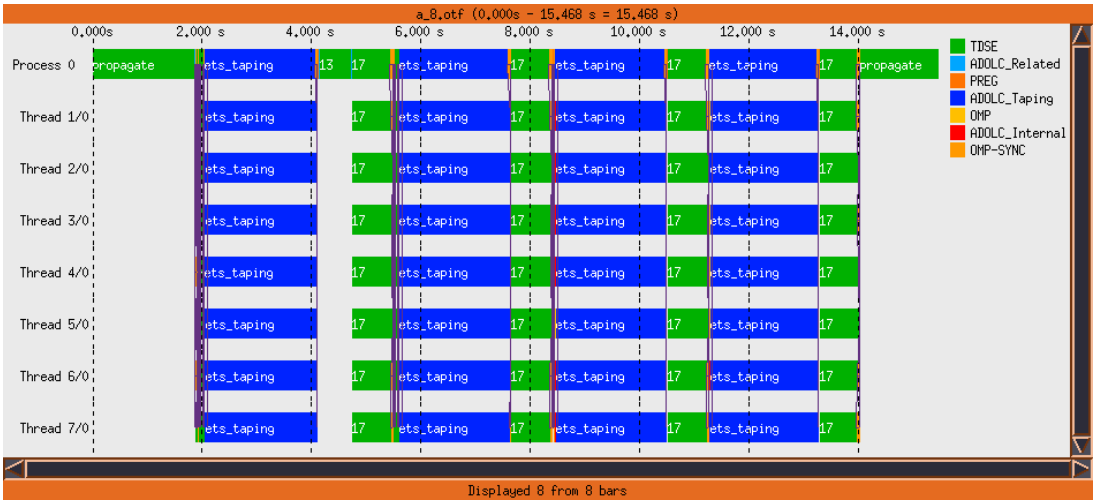


Figure 4.16: Adjusted version of the threadprivate variant of the differentiated plasma code – overview

notable difference to Figure 4.14 is the reduction of the serial program parts interrupting the parallel process. As for the original version of the code, a more detailed view on the change from one time step to another was created. Figure 4.17 provides a graphical representation that depicts the same position of the computation that has also been visualized in Figure 4.15. As can be seen, the derivative assembling phase is now performed in parallel and the effort for handling the external function is reduced considerably. However, the load balance during the assembling phase is not as good as expected. This cannot be

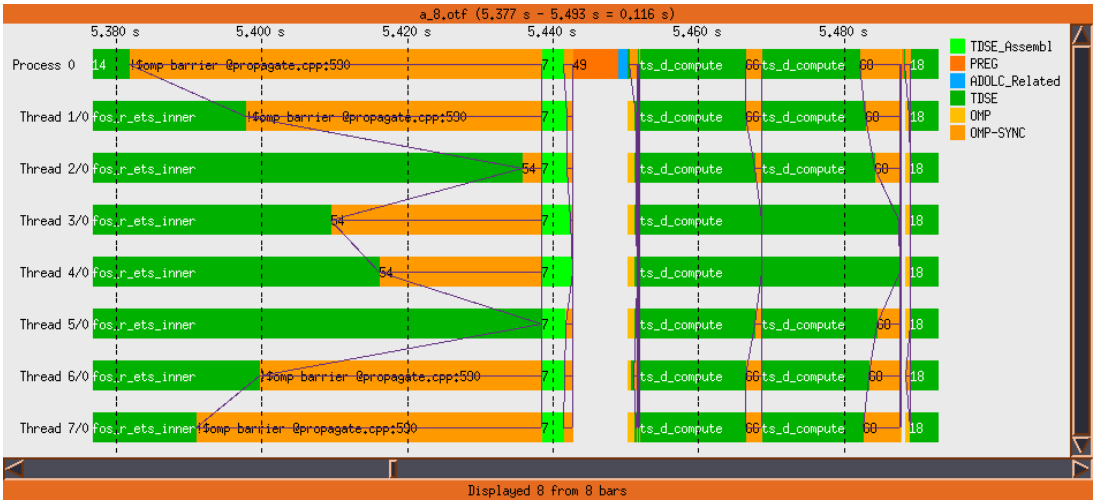


Figure 4.17: Adjusted version of the threadprivate variant of the differentiated plasma code – zoomed in

caused by the algorithm itself that essentially complies to the structure depicted in Example 4.4. Rather, hardware effects must be considered in the search for an answer. Different runtime behaviors of a small test program implementing the assembling algorithm are given in the Figures 4.18, 4.19 and 4.20. The first figure marks the initial situation that is characterized by three properties of the code. Firstly, each thread performs the same number of operations and, secondly, each thread allocates the memory for

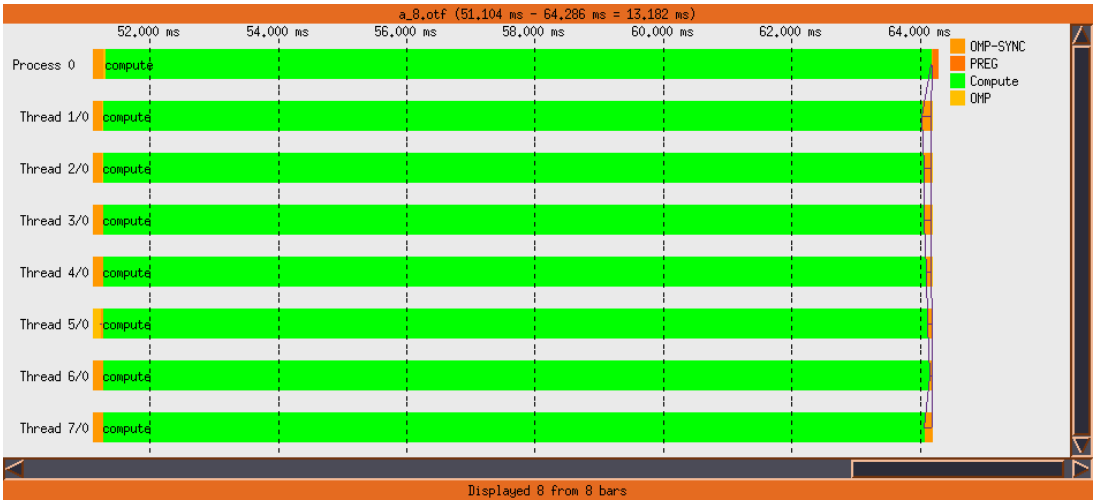


Figure 4.18: AD-assembling example program - all variable hold the numerical value 1.0

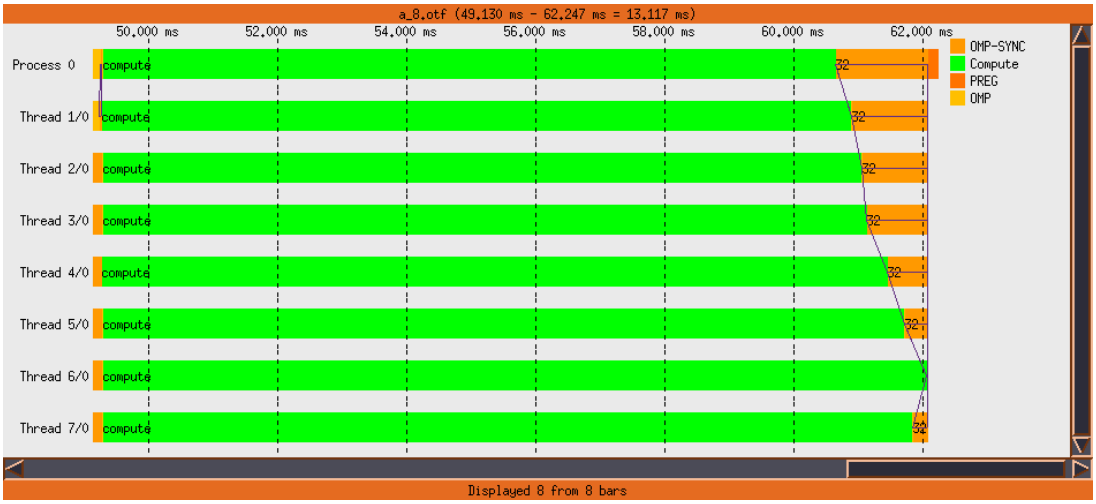


Figure 4.19: AD-assembling example program - increasing number of zeros in the computations

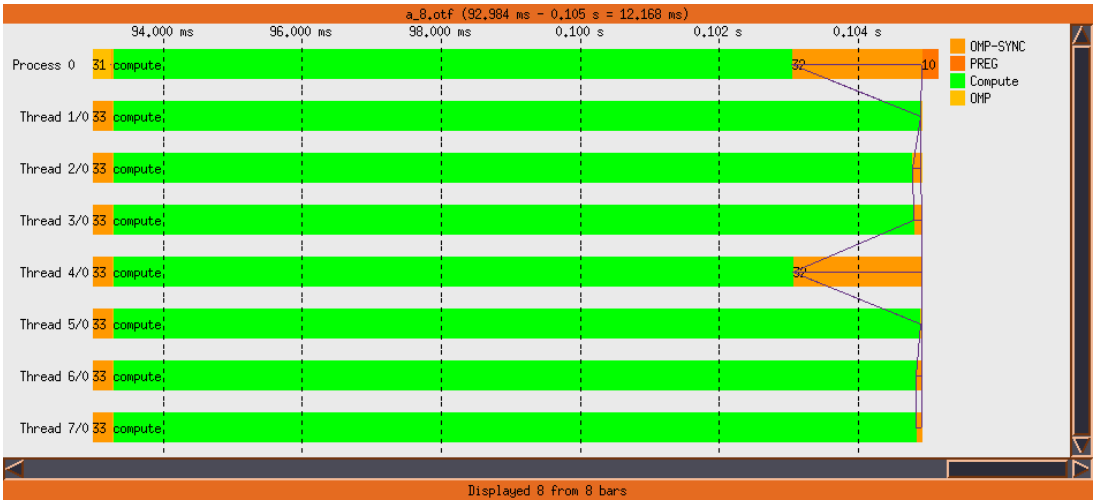


Figure 4.20: AD-assembling example program - different data placement

holding its local derivatives and the chunk of global derivatives it computes. Finally, all local derivative values are initialized with the numerical value 1.0. The test program was compiled without optimization to avoid disturbing changes by the compiler. As expected, providing this special scenario results in the depicted, nearly fully balanced runtime behavior.

By investigation of the intermediate results produced by the plasma code, a significant number of zero elements in the computed local derivatives was observable. Furthermore, different number of zero elements are produced by different threads. The second test case has been adjusted to widely reflect this situation by partially initializing the local derivatives to zero. Thread zero encounters $7/8$ zero elements, thread one $6/8$ zero elements and so on. By analyzing the runtime behavior depicted in Figure 4.19, it becomes clear that the higher is the number of zero elements, the lower is the required runtime for the computation. The only exception to this observation is marked by thread seven that performs all computations on non-zero local derivatives. Due to the missing information about the underlying processor internal optimizations, this exception cannot be further analyzed. However, it can be concluded that the number of zero elements in the derivative assembling phase has a considerable influence on the per-thread performance.

The optimal memory placement with respect to the algorithm and the involved threads that was applied so far currently does not represent the layout of the original plasma code. There, memory for global and local derivatives is allocated by the master thread. To reflect this setting, the initial version of the test program has been adjusted such that the master thread allocates all memory and initialized it to 1.0. Figure 4.20 depicts the runtime behavior of this new test program. As can be seen, two threads complete their work significantly earlier than all others. With exception of thread four, the runtime spent for the computations by all threads complies to the expectations that can be made from the properties of the test case. Therefore, the reduced computation time of thread four must be hardware dependent. The *SGI ALTIX 4700* installed at the TU Dresden features Intel Itanium II Montecito dual core processors. Each processor is accompanied by local memory and communication components. Globally, every processor may access its local memory as well as the remote memory of all other processors. Fastest access is bound to the local memory, naturally. In the considered test case of eight threads, four dual core processors are involved in the computations. All data is allocated in the local memory of the processor handling the master thread. Due to the dual core architecture, two cores and, thus, two thread can benefit from the faster access to the local memory. All other threads must access their data using the communication components what results in a higher effort. Hence, six of eight threads require a significant higher runtime for performing their computations.

Taking the much higher complexity of the plasma code into account, the derivative assembling phase is not only sensitive to the number of zero elements and the data allocation scheme but is also influenced by the actual cache layout of the involved data. This causes the runtime layout depicted in Figure 4.17 instead of the expected behavior visualized in Figure 4.18. Nevertheless, compared to the overall differentiation effort, the derivative assembling phases cover only a minor fraction of the runtime. A not fully balanced parallel computation in these special program parts is not relevant, therefore.

In the global scheme, the scalability of the differentiated plasma code is bound to the load balance in the program parts handling the differentiation of the code and the interruption time of these parts. The load balance of the derivative computation was very good from the beginning, compare Figure 4.14. Due to the code adjustments regarding the removal of the external differentiated function context, a considerable reduction of the interruption times could be achieved (Figure 4.16). This suggests preferable runtime results compared to the unadjusted version of the plasma code based on threadprivate variables. To verify this assumption, the runtime measurements were repeated using the new code version. The corresponding speedups and runtimes are presented in Figure 4.21. As can be seen, the runtime results confirm the better scalability of the new code. This allows to perform the computation of derivatives for the plasma code using a much higher number of processing elements without sacrificing a significant fraction of the efficiency. It shall be noted that, once performing the number of time steps targeted

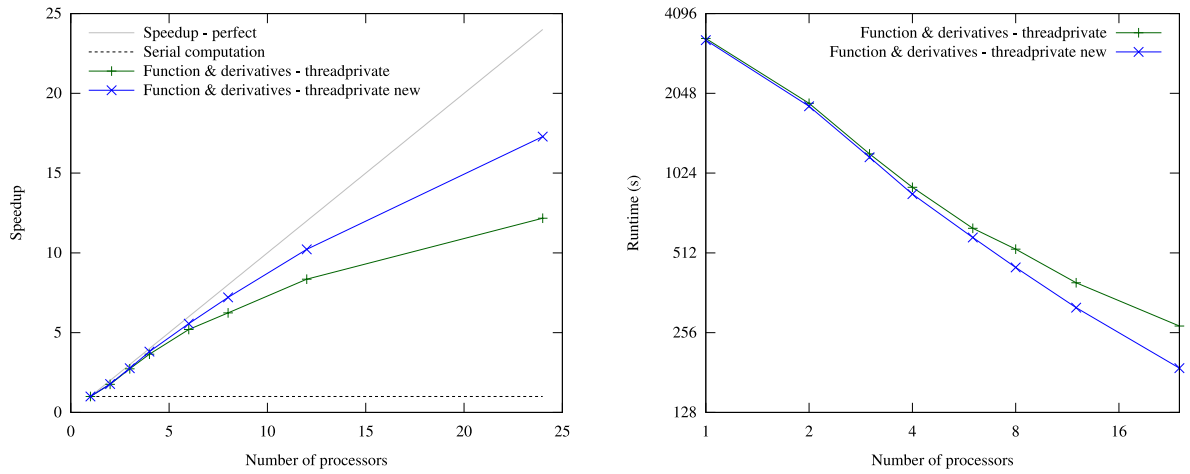


Figure 4.21: Speedups and runtimes for the adjusted threadprivate version of the plasma code for $N = 24$ wave functions

originally, the parallel fraction of the code will be quite near to 100 percent. Speedup expectations for the full simulation can be scaled up accordingly and will be near to the level called “perfect” in theory.

The results attained through the parallelization of the differentiation of the plasma code clarify that operator overloading based AD is prepared to meet the challenges that are brought up by the most complex codes applied in science and engineering. Thus, for the parallelization of derivative calculations using operator overloading AD, an answer has been found to a question that is still open for many other applications.

5 Conclusions & outlook

Automatic differentiation based on operator overloading features a long history and has shown to be highly valuable for most derivation tasks. Especially for programming languages for which AD-enabled compilers are not available or miss a critical feature, the operator overloading based approach often presents the only reasonable technique. However, the need to create an internal function representation to allow reverse mode differentiation, the introduced interpretive overhead when evaluating it, and the lower optimization level are often mentioned against its application. With the presented thesis, the author develops new techniques in the field of operator overloading based AD that improve the usability of the approach, reduce computational effort, and open up new application areas.

When applying operator overloading based AD, the dominating factor in terms of achievable runtime is given by the size of the created internal function representation. Therefore, reducing the size of the tape offers the best way of increasing the efficiency of the differentiation. Two different techniques were developed in this thesis that, independent of each other, serve this purpose. Firstly, introducing separate derivative contexts, the derivation procedure can be divided into subtasks. Each of these subtasks can thus be handled optimally using an accordingly adjusted approach. Based on this technique, the black box AD model can be displaced in favor of more sophisticated solutions that often also reduce the demands on the user. This was demonstrated using the robot application, which serves as a representative example.

Although the exploitation of specific structures allows significant reductions in the computational effort for many function, it is not the only technique that can be used. Independent of this approach, significant benefit can be drawn from taking activity information of the function into account. Then, operations only need to be represented in the tape if the propagated information influences derivative values. With this thesis, a technique was developed that allows to benefit from an activity analysis during the tape creation and evaluation phase, respectively. Reducing the differentiation effort this way was dedicated to the source-to-source approach, so far. As could be verified by use of the airfoil optimization, the state-tracking approach results in the creation of a smaller internal function representation, which can be further evaluated with reduced effort. In this special case, a reduction could be achieved even though the initial code was based on an already highly optimized AD-version.

The increasing complexity of the investigated functions more and more requires the application of parallelization techniques. It is obvious that automatic differentiation must face this fact and provide adequate differentiation strategies. Whereas parallelization is comparably simple if the overall computation can be divided into independent subtasks, the challenge is to be found in applications that do not feature this property. An appropriate technique for performing parallel reverse mode differentiation was developed in this thesis. It has been validated using the time propagation of a 1D quantum plasma. The resulting speedups confirm that automatic differentiation based on operator overloading can be applied in high performance computing environments, and that high speedups and drastically reduced runtimes can be gained.

In this thesis, all three techniques were developed and investigated as independently as possible. However, for further improvement, the combination of the approaches will be of interest in the near future. Especially the state-tracking offers a high potential of runtime reductions when being applied to tapes that are frequently reused. Checkpointing strategies and fixed point iterations are two examples of tasks that directly benefit. Furthermore, most improvements done for the serial approach will most probably increase the performance achieved by parallel derivation.

Using the newly developed techniques, operator overloading based AD may be applied to a wider range of applications, or may be applied with higher efficiency. Independent of the considered approach,

the main limitation of AD utilizing the overloading facilities of programming languages is always to be found in the size of created internal function representation. Within this context, the unrolling of loops presents a major drawback that may result in an unfavorable runtime behavior. Hence, one of the main challenges to be answered in the future is the development of techniques that allow a much more compact representation of loops. This not only avoids the expensive storing of every loop iteration, but also presents a major step towards an automatic parallel differentiation of parallelized user functions.

In this thesis, a specific subset of parallel functions has been considered for parallel AD. Utilizing the special properties of the differentiation environment as described in Subsection 3.3.4, an algorithm was developed that allows parallel differentiation. However, the general proof that derivatives can be computed using both the reverse mode of AD and parallelization techniques is not yet found. To the opinion of the author, significant effort must be invested to the analysis of this task. It will be important for the acceptance of AD that parallelism in the derivative calculation cannot only be exploited as widely as possible. Furthermore, the parallel differentiation should also be achieved with minimal user effort.

Many new challenges can be expected in the further development of automatic differentiation. Some known limitations could be removed with this thesis. Many other questions are still unanswered and will require additional effort to be invested in the future. Steady improvement of the technique and its tools will allow AD to preserve its acceptance for most differentiation tasks.

Bibliography

[ADP01] P. Aubert, N. Di Césaré, and O. Pironneau. Automatic differentiation in C++ using expression templates and application to a flow control problem. *Computing and Visualization in Science*, 3:197–208, 2001.

[AHN+96] A. Arnold, H.-C. Hoppe, W. E. Nagel, K. Solchenbach, and M. Weber. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 12(1):69–80, 1996.

[Amd67] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. volume 30 of *AFIPS conference proceedings*, pages 483–485, National Press Building, Washington, D.C. 20004, USA, 1967. Thompson Book Co. Spring joint computer conference, Atlantic City.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bau74] F. L. Bauer. Computational Graphs and Rounding Error. *SIAM Journal on Numerical Analysis*, 11(1):87–96, 1974.

[BBH00] C. H. Bischof, H. M. Bücker, and P. D. Hovland. On Combining Computational Differentiation and Toolkits for Parallel Scientific Computing. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller, editors, *Euro-Par 2000 – Parallel Processing, Proceedings of the 6th International Euro-Par Conference, Munich, Germany, August/September 2000*, volume 1900 of *Lecture Notes in Computer Science*, pages 86–94, Berlin, 2000. Springer.

[BBH02] H. M. Bücker, K. R. Buschelman, and P. D. Hovland. A Matrix-Matrix Multiplication Approach to the Automatic Differentiation and Parallelization of Straight-Line Codes. In U. Brinkschulte, K.-E. Großpietsch, C. Hochberger, and E. W. Mayr, editors, *Workshop Proceedings of the International Conference on Architecture of Computing Systems ARCS 2002, Germany, April 8–12, 2002*, pages 203–210, Berlin, 2002. VDE Verlag.

[BCKM96] C. H. Bischof, A. Carle, P. Khademi, and A. Mauer. ADIFOR 2.0: Automatic Differentiation of Fortran 77 Programs. *IEEE Computational Science & Engineering*, 3(3):18–32, 1996.

[Bel07] B. M. Bell. CppAD: A Package for C++ Algorithmic Differentiation, 2007. http://www.coin-or.org/CppAD.

[Ben96] J. Benary. Parallelism in the Reverse Mode. In M. Berz, C. H. Bischof, G. Corliss, and A. Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 137–147. SIAM, Philadelphia, PA, 1996.

[BFD+05] B. Bull, R. I. C. C. Francis, A. Dunn, A. McKenzie, D. J. Gilbert, and M. H. Smith. CASAL (C++ algorithmic stock assessment laboratory) – User Manual. Technical Report 127, NIWA, Private Bag 14901, Kilbirnie, Wellington, New Zealand, 2005.

[BG05] A. Bhattacharjee and D. A. Gurnett. *Introduction to Plasma Physics - With Space and Laboratory Applications*. Cambridge University Press, The Edinburgh Building, Cambridge, CB2 2RU, UK, 2005.

[BGJ91] C. H. Bischof, A. Griewank, and D. Juedes. Exploiting parallelism in automatic differentiation. In E. Houstis and Y. Muraoka, editors, *Proceedings of the 1991 International Conference on Supercomputing*, pages 146–153. ACM Press, Baltimore, Md., 1991. Also appeared as Preprint MCS–P204–0191, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., January 1991.

[BGP06] R. A. Bartlett, D. M. Gay, and E. T. Phipps. Automatic Differentiation of C++ Codes for Large-Scale Scientific Computing. In V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, editors, *Computational Science – ICCS 2006*, volume 3994 of *Lecture Notes in Computer Science*, pages 525–532, Heidelberg, 2006. Springer.

[Bis91] C. H. Bischof. Issues in Parallel Automatic Differentiation. In A. Griewank and G. F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 100–113. SIAM, Philadelphia, PA, 1991.

[BK03] C. Büskens and M. Knauer. Real-Time Trajectory Planning of the Industrial Robot IRB 6400. *PAMM*, 3:515–516, 2003.

[BLR+02] H. M. Bücker, B. Lang, A. Rasch, C. H. Bischof, and D. an Mey. Explicit Loop Scheduling in OpenMP for Parallel Automatic Differentiation. In J. N. Almhana and V. C. Bhavsar, editors, *Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications, Moncton, NB, Canada, June 16–19, 2002*, pages 121–126, Los Alamitos, CA, 2002. IEEE Computer Society Press.

[BLSS03] H. G. Bock, D. B. Leineweber, A. Schafer, and J. P. Schloder. An Efficient Multiple Shooting Based Reduced SQP Strategy for Large-Scale Dynamic Process Optimization – Part II: Software Aspects and Apllications. *Computers and Chemical Engineering*, 27(2):167–174, 2003.

[BLV03] C. H. Bischof, B. Lang, and A. Vehreschild. Automatic Differentiation for MATLAB Programs. *Proceedings in Applied Mathematics and Mechanics*, 2(1):50–53, 2003.

[BM06] M. Berz and K. Makino. COSY INFINITY Version 9.0 — Programmer's Manual. Technical Report MSUHEP-060803, Department of Physics, Michigan State University, East Lansing, MI 48824, USA, 2006.

[BN03] H. Brunst and W. E. Nagel. Scalable Performance Analysis of Parallel Systems: Concepts and Experiences. In G. R. Joubert, W. E. Nagel, F. J. Peters, and W. V. Walter, editors, *PARALLEL COMPUTING: Software Technology, Algorithms, Architectures and Applications*, volume 13 of *Advances in Parallel Computing*, pages 737–744, Dresden, Germany, 2003. Elsevier.

[BNM03] H. Brunst, W. E. Nagel, and A. D. Malony. A Distributed Performance Analysis Architecture for Clusters. In *IEEE International Conference on Cluster Computing, Cluster 2003*, pages 73–81, Hong Kong, China, dec 2003. IEEE Computer Society.

[BRM97] C. H. Bischof, L. Roh, and A. Mauer. ADIC — An Extensible Automatic Differentiation Tool for ANSI-C. *Software–Practice and Experience*, 27(12):1427–1456, 1997.

[BRV06] H. M. BÃ$\frac{1}{4}$cker, A. Rasch, and A. Vehreschild. Automatic Generation of Parallel Code for Hessian Computations. Preprint of the Institute for Scientific Computing RWTH-CS-SC-06-01, 2006.

[BS96] C. Bendtsen and O. Stauning. FADBAD, a Flexible C++ Package for Automatic Differentiation. Technical Report IMM–REP–1996–17, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark, 1996.

[Chr94] B. Christianson. Reverse Accumulation and Attractive Fixed Points. *Optimization Methods and Software*, 3:311–326, 1994.

[CLGM96] D. Conforti, L. De Luca, L. Grandinetti, and R. Musmanno. A parallel implementation of automatic differentiation for partially separable functions using PVM. *Parallel Computing*, 22(5):643–656, 1996.

[CNR03] M. Cohen, U. Naumann, and J. Riehme. Towards Differentiation-Enabled Fortran 95 Compiler Technology. In *Proceedings of the 18th ACM Symposium on Applied Computing, Melbourne, Florida, USA, March 9–12, 2003*, pages 143–147, 2003.

[DFF+03] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, editors. *Sourcebook of Parallel Computing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[DGMS94] J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. The PVM concurrent computing system: evolution, experiences, and trends. *Parallel Computing*, 20(4):531–545, 1994.

[DM98] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 05(1):46–55, 1998.

[FD99a] C. Faure and P. Dutto. Extension of Odyssée to the MPI library - Direct mode. Rapport de recherche 3715, INRIA, Sophia Antipolis, jun 1999.

[FD99b] C. Faure and P. Dutto. Extension of Odyssée to the MPI library - Reverse mode. Rapport de recherche 3774, INRIA, Sophia Antipolis, oct 1999.

[For06] S. A. Forth. An Efficient Overloaded Implementation of Forward Mode Automatic Differentiation in MATLAB. *ACM Transactions on Mathematical Software*, 32(2):195–222, 2006.

[Gay05] D. M. Gay. Semiautomatic Differentiation for Efficient Gradient Computations. In M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*, volume 50 of *Lecture Notes in Computational Science and Engineering*, pages 147–158. Springer, 2005.

[GGKK03] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Pearson Education Limited, Harlow, Essex CM20 2JE, 2. edition, 2003.

[GJU96] A. Griewank, D. Juedes, and J. Utke. Algorithm 755: ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++. *ACM Transactions on Mathematical Software*, 22(2):131–167, 1996.

[GK06] R. Giering and T. Kaminski. Automatic Sparsity Detection Implemented as a Source-to-Source Transformation. In V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, editors, *Computational Science – ICCS 2006*, volume 3994 of *Lecture Notes in Computer Science*, pages 591–598, Heidelberg, 2006. Springer.

[GKS05] R. Giering, T. Kaminski, and T. Slawig. Generating Efficient Derivative Code with TAF: Adjoint and Tangent Linear Euler Flow Around an Airfoil. *Future Generation Computer Systems*, 21(8):1345–1355, 2005.

[GMWW07] N. R. Gauger, C. Moldenhauer, A. Walther, and M. Widhalm. Automatic Differentiation of an Entire Design Chain for Aerodynamic Shape Optimization. Notes on Numerical Fluid Mechanics and Multidisciplinary Design. Springer, 2007. To appear in Vol. 96.

[GR03] S. Gorlatch and F. A. Rabhi, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, London, UK, 2003.

[Gri92] A. Griewank. Achieving Logarithmic Growth of Temporal and Spatial Complexity in Reverse Automatic Differentiation. *Optimization Methods and Software*, 1:35–54, 1992.

[Gri00] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Applied Mathematics. SIAM, Philadelphia, PA, 2000.

[Gür06] N. Gürtler. Simulation eines eindimensionalen idealen Quantenplasmas auf Parallelrechnern, 2006. Diploma thesis in physics, Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen, Germany.

[Gür07] N. Gürtler. Parallel Automatic Differentiation of a Quantum Plasma Code, 2007. Diploma thesis in computer science, Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen, Germany.

[GW00] A. Griewank and A. Walther. Algorithm 799: Revolve: An Implementation of Checkpoint for the Reverse or Adjoint Mode of Computational Differentiation. *ACM Transactions on Mathematical Software*, 26(1):19–45, mar 2000. Also appeared as Technical University of Dresden, Technical Report IOKOMO-04-1997.

[GW03] R. Griesse and A. Walther. Parametric Sensitivities for Optimal Control Problems using Automatic Differentiation. *Optimal Control Applications and Methods*, 24(6):297–314, 2003.

[HB98] P. D. Hovland and C. H. Bischof. Automatic Differentiation of Message-Passing Parallel Programs. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 98–104, Los Alamitos, CA, 1998. IEEE Computer Society Press.

[Hem94] R. Hempel. The MPI Standard for Message Passing. In W. Gentzsch and U. Harms, editors, *High–Performance Computing and Networking, InternationalConference and Exhibition, Proceedings, Volume II: Networking and Tools*, volume 797 of *Lecture notes in computer science*, pages 247–252. Springer, 1994.

[HGP05] L. Hascoët, R.-M. Greborio, and V. Pascual. Computing Adjoints by Automatic Differentiation with TAPENADE. In B. Sportisse and F.-X. Le Dimet, editors, *École INRIA-CEA-EDF "Problèmes non-linéaires appliqués"*. Springer, 2005.

[HHG05] P. Heimbach, C. Hill, and R. Giering. An efficient exact adjoint of the parallel MIT general circulation model, generated via automatic differentiation. *Future Generation Computer Systems*, 21(8):1356–1371, 2005.

[HKL+05] F. P. Hart, N. Kriplani, S. R. Luniya, C. E. Christoffersen, and M. B. Steer. Streamlined Circuit Device Model Development with fREEDA® and ADOL-C. In H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*, Lecture Notes in Computational Science and Engineering, pages 295–307. Springer, 2005.

[HM00] P. D. Hovland and L. C. McInnes. Parallel Simulation of Compressible Flow Using Automatic Differentiation and PETSc. Technical Report ANL/MCS-P796-0200, Mathematics and Computer Science Division, Argonne National Laboratory, 2000. To appear in a special issue of *Parallel Computing* on "Parallel Computing in Aerospace".

[HNP05] L. Hascoët, U. Naumann, and V. Pascual. "To Be Recorded" Analysis in Reverse-Mode Automatic Differentiation. *Future Generation Computer Systems*, 21(8), 2005.

[HNRS99] P. D. Hovland, B. Norris, L. Roh, and B. F. Smith. Developing a Derivative-Enhanced Object-Oriented Toolkit for Scientific Computations. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, pages 129–137, Philadelphia, PA, 1999. SIAM.

[Hov97] P. D. Hovland. *Automatic differentiation of parallel programs*. PhD thesis, 1997. Advisers: M. T. Heath and C. H. Bischof.

[HW99] R. Hempel and D. W. Walker. The Emergence of the MPI Message Passing Standard for Parallel Computing. *Computer Standards & Interfaces*, 21:51–62, 1999.

[Ins85] The Institute of Electrical and Electronics Engineers, Inc, 345 East 47th Street, New York, NY 10017, USA. *IEEE Standard for Binary Floating-Point Arithmetic*, 1985.

[JG90] J. Juedes and A. Griewank. Implementing Automatic Differentiation Efficiently. Technical Memorandum ANL/MCS–TM–140, Mathematics and Computer Sciences Division, Argonne National Laboratory, Argonne, Ill., 1990.

[KF06] R. V. Kharche and S. A. Forth. Source Transformation for MATLAB Automatic Differentiation. In V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, editors, *Computational Science – ICCS 2006*, volume 3994 of *Lecture Notes in Computer Science*, pages 558–565, Heidelberg, 2006. Springer.

[KRE+06] B. Kreaseck, L. Ramos, S. Easterday, M. Strout, and P. Hovland. Hybrid Static/Dynamic Activity Analysis. In V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, editors, *Computational Science – ICCS 2006*, volume 3994 of *Lecture Notes in Computer Science*, pages 582–590, Heidelberg, 2006. Springer.

[Kub98] K. Kubota. A Fortran77 Preprocessor for Reverse Mode Automatic Differentiation with Recursive Checkpointing. *Optimization Methods and Software*, 10(2):315–335, 1998.

[KW06] A. Kowarz and A. Walther. Optimal Checkpointing for Time-Stepping Procedures in ADOL-C. In V. N. Alexandrov, G. Dick van Albada, P. M. A. Sloot, and J. Dongarra, editors, *Computational Science – ICCS 2006*, volume 3994 of *Lecture Notes in Computer Science*, pages 541–549, Heidelberg, 2006. Springer.

[KW07] A. Kowarz and A. Walther. Efficient Calculation of Sensitivities for Optimization Problems. *Discussiones Mathematicae. Differential Inclusions, Control and Optimization*, 27:119–134, 2007.

[Man02] M. Mancini. A Parallel Hierarchical Approach for Automatic Differentiation. In G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors, *Automatic Differentiation of Algorithms: From Simulation to Optimization*, Computer and Information Science, chapter 27, pages 231–236. Springer, New York, NY, 2002.

[Mat06] The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760-2098. *Getting Started with MATLAB Verison 7*, 2006. http://www.mathworks.com.

[Nau02] U. Naumann. Elimination Techniques for Cheap Jacobians. In G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors, *Automatic Differentiation of Algorithms: From Simulation to Optimization*, Computer and Information Science, chapter 29, pages 247–253. Springer, New York, NY, 2002.

[Nau04] U. Naumann. Optimal Accumulation of Jacobian Matrices by Elimination Methods on the Dual Computational Graph. *Mathematical Programming, Ser. A*, 99(3):399–421, 2004.

[PH05] V. Pascual and L. Hascoët. Extension of TAPENADE toward Fortran 95. In M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*, Lecture Notes in Computational Science and Engineering. Springer, 2005.

[SH06] M. Strout and P. Hovland. Linearity Analysis for Automatic Differentiation. In V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, editors, *Computational Science – ICCS 2006*, volume 3994 of *Lecture Notes in Computer Science*, pages 574–581, Heidelberg, 2006. Springer.

[SKH06] M. Strout, B. Kreaseck, and P. Hovland. Data-Flow Analysis for MPI Programs. *International Conference on Parallel Processing (ICPP)*, 0:175–184, 2006.

[SPF00] S. Stamatiadis, R. Prosmiti, and S. C. Farantos. AUTO_DERIV: Tool for automatic differentiation of a FORTRAN code. *Computer Physics Communications*, 127(2&3):343–355, 2000. Catalog number: ADLS.

[SW89] D. F. Stubbs and N. W. Webre. *Data structures with abstract data types and Pascal*. Brooks/Cole Publ. Co., Pacific Grove, CA, 2. edition, 1989.

[SWGH06] S. Schlenkrich, A. Walther, N. R. Gauger, and R. Heinrich. Differentiating Fixed Point Iterations with ADOL-C: Gradient Calculation for Fluid Dynamics. Technical report, Technische Universität Dresden, Dresden 01062, Germany, 2006.

[TFP03] M. Tadjouddine, S. A. Forth, and J. D. Pryce. Hierarchical Automatic Differentiation by Vertex Elimination and Source Transformation. In V. Kumar, M. L. Gavrilova, C. J. K. Tan, and P. L'Ecuyer, editors, *Computational Science and Its Applications – ICCSA 2003, Proceedings of the International Conference on Computational Science and its Applications, Montreal, Canada, May 18–21, 2003. Part II*, volume 2668 of *Lecture Notes in Computer Science*, pages 115–124, Berlin, 2003. Springer.

[TFPR01] M. Tadjouddine, S. A. Forth, J. D. Pryce, and J. K. Reid. On the Implementation of AD using Elimination Methods via Source Transformation: Derivative Code Generation. Technical Report AMOR Report No 2001/4, Cranfield University (RMCS Shrivenham), Swindon SN6 8LA, UK, 2001. online at www.rmcs.cranfield.ac.uk/esd/amor/filestore/mt_amor0104.pdf.

[Utk04] J. Utke. OpenAD: Algorithm Implementation User Guide. Technical Memorandum ANL/MCS–TM–274, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 2004.

[VGK04] M. Voßbeck, R. Giering, and T. Kaminski. Towards a tool for forward and reverse mode source to source transformation in C++, 2004.

[Wal99] Andrea Walther. *Program Reversal Schedules for Single- and Multi-processor Machines*. PhD thesis, Institute of Scientific Computing, Technical University Dresden, Germany, 1999.

[WG04] A. Walther and A. Griewank. Advantages of binomial checkpointing for memory-reduces adjoint calculations. In M. Feistauer, V. Dolejsi, P. Knobloch, and K. Najzar, editors, *Numerical mathematics and advanced applications*, Proceedings ENUMATH 2003, pages 834–843. Springer, 2004.

Acknowledgments

I would like to express my gratitude to all those who supported my work on this thesis. First of all, I would like to thank my supervising tutors Prof. Andrea Walther and Prof. Wolfgang E. Nagel for many fruitful discussions, suggestions and encouragement. Special thanks to Andrea Walther for helping me putting this thesis into shape. Furthermore, I would like to thank the German Research Foundation (DFG) for the financial support within the project "Effiziente Berechnung von Adjungierten für komplexe, C/C++-basierte Programmpakete".

Moreover I would like to thank the Center for Information Services and High Performance Computing (ZIH), Technische Universität Dresden for providing the computational environment for the expensive numerical simulations and tests. Especially, I am indebted to the administrators Michael Kluge and Guido Juckeland for the technical support. For cooperation on the numerical examples, especially for providing the source codes, I am grateful to the German Aerospace Center (DLR) and the Rheinisch-Westfälische Technische Hochschule Aachen. Special thanks belong to Niels Gürtler, who kept track of the physical background of the plasma simulation and cooperated on the parallelization of the code. Furthermore, I wish to thank my colleagues at the Institute of Scientific Computing, Technische Universität Dresden for providing the stimulating working atmosphere.

Moreover, I am grateful to my parents for moral encouragement, especially in the final phase of the work. Last but not least, I wish to thank the members of the Pillnitzer Reiterhof for guiding the scientist back to reality whenever necessary or appropriate.