Compiler-based Differentiation of Numerical Simulation Codes

Michel Schanen, Michael Förster, Boris Gendler, Uwe Naumann
LuFG Informatik 12: Software and Tools for Computational Engineering
RWTH Aachen University
Aachen, Germany
{schanen,foerster,bgendler,naumann}@stce.rwth-aachen.de

***Abstract*—Based on algorithmic differentiation, we present a derivative code compiler capable of transforming implementations of multivariate vector functions into a program for computing derivatives. The resulting values are accurate up to machine precision compared to the common numerical approximation by finite differences. This paper gives a short mathematical background of algorithmic differentiation while focusing on the user's perspective of applying derivative generation tools on an already implemented code. This process is illustrated by a one dimensional implementation of Burgers' equation in a generic optimization setting using for example Newton's method. In this implementation, finite differences are replaced by the computation of adjoints, thus saving an order of magnitude in terms of computational complexity.**

***Keywords*-Algorithmic Differentiation; Source Transformation; C/C++; Optimization; Numerical Simulation;**

I. Introduction

A typical problem in fluid dynamics is given by the continuous Burgers equation [1]

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2} \quad , \tag{1}$$

describing shock waves moving through gases. u denotes the velocity field of the fluid with viscosity ν. Similar governing equations represent the core of many numerical simulations. Such simulations are often subject to various optimization techniques involving derivatives. Thus, Burgers' equation will serve as a case study for a compiler-based approach to the accumulation of the required derivatives.

Suppose we solve the differential equation in (1) by discretization using finite differences on a equidistant one-dimensional grid with n_x points. For given initial conditions $u_{i,0}$ with $0 < i \leq n_x$ we simulate a physical process by integrating over n_t time steps according to the leapfrog/DuFort-Frankel scheme presented in [2]. At time step j we compute $u_{i,j+1}$ for time step $j+1$ according to

$$\begin{aligned} u_{i,j+1} = u_{i,j-1} &- \frac{\Delta t}{\Delta x}\left(u_{i,j}\left(u_{i+1,j} - u_{i-1,j}\right)\right) \\ &+ \frac{2\Delta t}{\Delta x^2}\left(u_{i+1,j} - \left(u_{i,j+1} + u_{i,j-1}\right) + u_{i-1,j}\right), \end{aligned} \tag{2}$$

where Δt is the time interval and Δx is the distance between two grid points. In general, if the initial conditions $u_{i,0}$ cannot be accurately measured, they are essentially replaced by approximated values. To improve their accuracy additional observed values $u^{ob} \in \mathbb{R}^{n_x \times n_t}$ are taken into account. The discrepancy between observed values $u^{ob}_{i,j}$ and simulated values $u_{i,j}$ are evaluated by the cost function

$$y = \frac{1}{2}\sum_{i=1}^{n_x}\sum_{j=1}^{n_t}(u_{i,j} - u^{ob}_{i,j})^2 \quad , \tag{3}$$

allows us to obtain improved estimations for the initial conditions by applying, for example, Newton's method [3] to solve the data assimilation problem with Burgers' equation as constraints [4]. The single Newton steps are repeated until the residual cost y undercuts a certain threshold.

In Section II, we introduce algorithmic differentiation as implemented by our derivative code compiler `dcc` covering both the tangent-linear as well as the adjoint model. Section III provides a user's perspective on the application of `dcc`. Higher-order differentiation models are discussed in Section IV. Finally, the results of our case study are discussed in Section V.

II. Algorithmic Differentiation

The minimization of the residual is implemented by resorting to Newton's second-order method for minimization. In general, Newton's method may be applied to arbitrary differentiable multivariate vector functions $\mathbf{y} = F(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^m$. This algorithm heavily depends on the accurate and fast computation of Jacobian and Hessian values, since one iterative step $\mathbf{x}_i \to \mathbf{x}_{i+1}$ is computed by

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \nabla^2 F(\mathbf{x}_i)^{-1} \cdot \nabla F(\mathbf{x}_i) \quad . \tag{4}$$

The easiest method of approximating partial derivatives $\nabla_{x_i} F$ uses the finite difference quotient

$$\nabla_{x_i} F(\mathbf{x}) \approx \frac{F(\mathbf{x} + h \cdot \mathbf{e_i}) - F(\mathbf{x})}{h} \quad , \tag{5}$$

for the Cartesian basis vector $\mathbf{e_i} \in \mathbb{R}^n$ and with $\mathbf{x} \in \mathbb{R}^n$, $h \to 0$. In order to accumulate the Jacobian of a multivariate function the method is rerun n times to perturb each component of the input vector $\mathbf{x}$. The main advantage of this method resides in its straightforward implementation; no additional changes to the code of the function F are necessary. However, the derivatives accumulated through finite differences are only approximations. This represents a major

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-139-7

drawback for codes that simulate highly nonlinear systems, resulting in truncation and cancellation errors or simply providing wrong results. In particular by applying the Taylor expansion to the second-order centered difference quotient we derive a machine precision induced approximation error of $\frac{\epsilon}{h^2}$, with ϵ being the rounding error.

Algorithmic differentiation (AD) [5] solves this problem analytically, changing the underlying code to compute derivatives by applying symbolic differentiation rules to individual assignments and using the chain rule to propagate derivatives along the flow of control. The achieved accuracy only depends on the machine's precision ϵ. There exist two distinct derivative models, differing in the order of application of the associative chain rule. Let ∇F be the Jacobian of F. The *tangent-linear* code

$$\begin{aligned} &F\big(\overset{\downarrow}{\mathbf{x}}, \underset{\downarrow}{\mathbf{y}}\big) \xrightarrow{\texttt{dcc}} \dot{F}\big(\overset{\downarrow}{\mathbf{x}}, \overset{\downarrow}{\dot{\mathbf{x}}}, \underset{\downarrow}{\mathbf{y}}, \underset{\downarrow}{\dot{\mathbf{y}}}\big) \quad , \\ &\text{where} \\ &\qquad \dot{\mathbf{y}} = \nabla F\left(\mathbf{x}\right) \cdot \dot{\mathbf{x}} \\ &\text{and} \quad \mathbf{y} = F(\mathbf{x}) \quad , \end{aligned} \tag{6}$$

of F computes the directional derivative $\dot{\mathbf{y}}$ of the outputs $\mathbf{y}$ with respect to the inputs $\mathbf{x}$ for a given direction $\dot{\mathbf{x}} \in \mathbb{R}^n$, while arrows designate inputs and outputs. By iteratively setting $\dot{\mathbf{x}}$ equal to each of the n Cartesian basis vectors in $\mathbb{R}^n$, we accumulate the entire Jacobian. This leads to a runtime complexity identical to finite differences of $\mathcal{O}(n) \cdot cost(F)$, where $cost(F)$ denotes the computational cost of a single function evaluation.

By exploiting the associativity of the chain rule, the *adjoint* code

$$\begin{aligned} &F\big(\overset{\downarrow}{\mathbf{x}}, \underset{\downarrow}{\mathbf{y}}\big) \xrightarrow{\texttt{dcc}} \bar{F}\big(\overset{\downarrow}{\mathbf{x}}, \overset{\downarrow}{\bar{\mathbf{x}}}, \underset{\downarrow}{\mathbf{y}}, \underset{\downarrow}{\bar{\mathbf{y}}}\big) \quad , \\ &\text{where} \\ &\qquad \mathbf{y} = F(\mathbf{x}) \\ &\text{and} \quad \bar{\mathbf{x}} = \bar{\mathbf{x}} + \nabla F\left(\mathbf{x}\right)^{\intercal} \cdot \bar{\mathbf{y}} \quad , \end{aligned} \tag{7}$$

of F computes *adjoints* $\bar{\mathbf{x}} \in \mathbb{R}^n$ of the inputs $\mathbf{x}$ for given adjoints $\bar{\mathbf{y}} \in \mathbb{R}^m$ of the outputs. To accumulate the entire Jacobian we have to iteratively set $\bar{\mathbf{y}}$ equal to each Cartesian basis vector of $\mathbb{R}^m$ yielding a runtime complexity of $\mathcal{O}(m) \cdot cost(F)$. Note that for scalar functions with $m = 1$ the accumulation of the Jacobian amounts to the computation of one gradient yielding a runtime cost of $\mathcal{O}(1) \cdot cost(F)$ for the adjoint model compared to $\mathcal{O}(n) \cdot cost(F)$ for the tangent-linear model. In this particular case, we are able to compute gradients at a small constant multiple of the cost of a single function evaluation. The reduction of this factor down toward the theoretical minimum of three [5] is one of the major challenges addressed by ongoing research and development in the field of AD [6], [7], [8].

III. dcc - A Derivative Code Compiler

Numerical optimization problems are commonly implemented as multivariate scalar functions $y = F(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$, describing some residual y of a numerical model. We assume that the goal is to minimize a norm of this residual y by adapting the inputs $\mathbf{x}$. Therefore, for better readability and without the loss of generality, in this paper, we will only cover multivariate scalar functions.

The main link between dcc and the mathematical models of AD is the ability to decompose each function implementation into single assignment code (SAC) as follows:

$$\begin{aligned} &\text{for } j = n, \ldots, n+p \\ &\qquad v_j = \varphi_j(v_i)_{i \prec j} \quad . \end{aligned} \tag{8}$$

The entire program is regarded as a sequence of $p+1$ elemental statements. In each statement an elemental function φ_j is applied to a set of variables $(v_i)_{i \prec j}$ yielding the unique *intermediate* variable v_j with $i \prec j$ denoting a dependence of v_j on v_i. The *independent* inputs are given by $v_i = x_i$ for $i = 0, \ldots, n-1$ while the *dependent* output of F is the final value $y = v_{n+p}$. When dcc applies the tangent-linear model to each of the $p+1$ assignments, we obtain

$$\begin{aligned} &\text{for } j = n, \ldots, n+p \\ &\qquad \dot{v}_j = \sum_{i \prec j} \frac{\partial \varphi_j}{\partial v_i} \cdot \dot{v}_i \\ &\qquad v_j = \varphi_j(v_i)_{i \prec j} \quad . \end{aligned} \tag{9}$$

Considering the j-th assignment in (9), the local k-th entry of the gradient $(\frac{\partial \varphi_j}{\partial v_k})_{k \prec j}$ is provided in $\dot{v}_j$ by setting $\dot{v}_k$ to one and all $(\dot{v}_i)_{k \neq i \prec j}$ to zero. The gradient component $(\frac{\partial y}{\partial x_k})_{k \in \{0,\ldots,n-1\}}$ is obtained by evaluating (9) and setting $\dot{x}_k$ to one and all other $(\dot{x}_i)_{k \neq i \in \{0,\ldots,n-1\}}$ to zero. To get the whole gradient we have to evaluate (9) n times letting $\dot{\mathbf{x}}$ range over the Cartesian basis vectors in $\mathbb{R}^n$. The adjoint model is acquired by transforming (8) into:

$$\begin{aligned} &\text{for } j = n, \ldots, n+p \\ &\qquad v_j = \varphi_j(v_i)_{i \prec j} \\ &\text{for } i \prec j \text{ and } j = n+p, \ldots, n \\ &\qquad \bar{v}_i = \bar{v}_i + \frac{\partial \varphi_j}{\partial v_i}(v_k)_{k \prec j} \cdot \bar{v}_j \quad . \end{aligned} \tag{10}$$

The first part consists of the original assignments $j = n, \ldots, n+p$ and is called *forward section*. The *reverse section* follows with the computation of the adjoint variables in the order $j = n+p, \ldots, n$. Note the reversed order of the assignments as well as the changed data flow of the left and right-hand sides compared with the original assignments. To compute the local gradient $(\frac{\partial \varphi_j}{\partial v_k})_{k \prec j}$ we have to initialize $(\bar{v}_i)_{i \prec j}$ with zero and $\bar{v}_j$ with one. The initialization with zero is mandatory because $(\bar{v}_i)_{i \prec j}$ occurs in (10) on both sides of the adjoint assignment. According to (7), the adjoint variable $\bar{v}_j$ is an input variable. Therefore it is initialized with the "Cartesian basis vector" in $\mathbb{R}$.

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-139-7

The important advantage of the adjoint model is that by evaluating (10) only once we obtain the full gradient $\frac{\partial y}{\partial \mathbf{x}}$ in $\bar{x}_i = \bar{v}_i$ for $i = 0, \ldots, n-1$. To achieve this we have to initialize $(\bar{x}_i)_{i=0,\ldots,n-1}$ with zero and $\bar{y}$ with one. As mentioned above $\bar{\mathbf{x}}$ must be zero because it occurs not only on the left-hand side in (7) and y is initialized with the value of the Cartesian basis vector in $\mathbb{R}$.

In (8), we assumed that the input code is given as a SAC. This is an oversimplification in terms of real codes. The adjoint code has to deal with the fact that real code variables are overwritten frequently. One way to simulate the predicate of unique intermediate variables is to store certain left-hand side variables on a stack during the augmented forward section. Candidates for storing on the stack are those variables that are being overwritten and are required for later use during the computation of the local gradients and associated adjoints. Before evaluating the corresponding adjoint assignment in the reverse section the values are restored from the stack.

For illustration purposes we consider Listing 1 showing an implementation of the non-linear reduction $y(\mathbf{x}) = \prod_{i=0}^{n-1} \sin(x_i)$. dcc parses only functions with **void** as a return type (line 1). All inputs and return values are passed through the arguments, which in turn only consist of arrays (called by pointers) and scalar values (called by reference). Additionally we may pass an arbitrary number of integer arguments by value or by reference. We assume that all differentiable functions are implemented using values of type **double**. Therefore, only variables of type **double** are directly affected by the differentiation process.

```
1  void f(int n, double *x, double &y)
2  {
3    int i=0;
4    y=0;
5    for(i=0;i<n;i++) {
6      y=y*sin(x[i]);
7    }
8  }
```

Listing 1: dcc input code.

Using the command line `dcc f.c -t`, we instruct the compiler to use the tangent-linear (`-t`) mode in order to generate the function t1_f (**t**angent-linear, **1**st-order version of f) presented in Listing 2. The original function arguments x and y are augmented with their associated tangent-linear variables t1_x and t1_y. Inside a driver program this code has to be rerun n times letting the input vector t1_x range over the Cartesian basis vectors in $\mathbb{R}^n$ to accumulate the entire gradient. Listing 3 shows how to use the generated code of Listing 2 in a driver program. Lines 2 and 5 let input variable t1_x range over the Cartesian basis vectors. By setting t1_x[i] to 1 the function t1_f (line 3) computes the partial derivative of y with respect to x[i].

The command line `dcc f.c -a` tells dcc to apply the adjoint mode (`-a`) to `f.c`. The result is the function

```
1  void t1_f(int n, double* x, double* t1_x
2                 , double& y, double& t1_y)
3  {
4    ...
5    for(int i=0;i<n;i++) {
6      y=y*sin(x[i]);
7      t1_y=t1_y*sin(x[i])+y*cos(x[i])*t1_x[i];
8    }
9    ...
10 }
```

Listing 2: Tangent-linear version of f as generated by dcc

```
1  for(int i=0; i<n;i++) {
2    t1_x[i]=1;
3    t1_f(n, x, t1_x, y, t1_y);
4    gradient[i]=t1_y;
5    t1_x[i]=0;
6  }
```

Listing 3: Driver for t1_f

a1_f (**a**djoint, **1**st-order version of f) shown in Listing 4. As in the tangent-linear case each function argument is augmented by an associated adjoint component, here a1_x and a1_y. As mentioned above we need a stack in the adjoint code for storing data during the forward section. The *augmented forward section* uses stacks to store values that are being overwritten and to store the control flow. The actual implementation of the stack is not under consideration here; therefore we replaced the calls to the stacks with macro definitions for better readability. By default, dcc generates code that uses static arrays which ensures high runtime performance. There are three different stacks used in the adjoint code. The stack called CS is for storing the control flow, FDS takes floating point values and IDS keeps integer values. The unique identifier of the two basic blocks [9] in the forward section are stored in lines 6 and 9. For example, after evaluating the augmented forward section of Listing 4, the stack CS contains the following sequence

$$0, \underbrace{1, \ldots, 1}_{\text{n times}} \tag{11}$$

In line 10, variable y is stored onto the stack because it is overwritten in each iteration although needed in line 21. Hence, we restore the value of y in line 20. For the same reason we store and restore the value of i in line 11 and 19. The reverse section consist of a loop that processes the control flow stack CS. The basic block identifiers are restored from the stack and depending on the value, the corresponding adjoint basic block is executed. For example, the sequence given in (11) as content in the CS stack leads to a n-times evaluation of the adjoint basic block one and afterward one evaluation of the adjoint basic block zero. The basic block one in line 9 to 11 has the corresponding adjoint basic block in line 19 to 22. In contrast to (7), in

```
1  void a1_f(int n, double* x, double* a1_x,
2                   double& y, double& a1_y)
3  {
4    int i=0;
5    // augmented forward section
6    CS_PUSH(0);
7    y=0;
8    for ( i=0; i<n; i++) {
9           CS_PUSH(1);
10          FDS_PUSH(y); y=y*sin(x[i]);
11          IDS_PUSH(i);
12   }
13   // reverse section
14   while (CS_NON_EMPTY) {
15          if (CS_TOP==0) {
16            a1_y=0;
17          }
18          if (CS_TOP==1) {
19            IDS_POP(i);
20            FDS_POP(y);
21            a1_x[i]+=y*cos(x[i])*a1_y;
22            a1_y=sin(x[i])*a1_y;
23          }
24          CS_POP;
25   }
26 }
```

Listing 4: Adjoint dcc output

line 22 the adjoint a1_y is not incremented but assigned. This is due to the fact that y is on both hand sides of the original assignment in line 10. This brings an aliasing effect into play. This effect can be avoided with help of intermediate variables; making this code difficult to read. For that reason we show the adjoint assignment without intermediate variables. dcc generates adjoint assignments with intermediate variables and incrementation of the left-hand side as shown in (7). The dcc-generated code and the one shown here are semantically equivalent. To accumulate the gradient using the function a1_f, we again have to write a driver, presented in Listing 5. It is sufficient to initialize the adjoint variable a1_y and call the adjoint function a1_f only once to get the whole gradient (line 2), illustrating the reduced runtime complexity of the adjoint mode.

```
1  a1_y=1;
2  a1_f(n, x, a1_x, y, a1_y);
3  for(int j=0; j<n;j++)
4     gradient[j]=a1_x[j];
```

Listing 5: Driver for a1_f

IV. Higher Order Differentiation

Numerical optimization algorithms often involve higher-order derivative models. Thus, the need for Hessians is imminent. With this in mind, dcc was designed to generate higher-order derivative codes effortlessly using its *reapplication feature*. dcc is able to generate jth-order derivative code by reading $(j-1)$th-order derivative code as the input. In this section we will focus on second-order models.

The tangent-linear mode reapplied to the first-order tangent-linear code (6) with $m = 1$ for scalar functions yields the second-order tangent-linear code

$$\dot{F}\big(\overset{\downarrow}{\mathbf{x}},\overset{\downarrow}{\dot{\mathbf{x}}},y,\dot{y}\big) \xrightarrow{\texttt{dcc}} \tilde{\dot{F}}\big(\overset{\downarrow}{\mathbf{x}},\overset{\downarrow}{\tilde{\mathbf{x}}},\overset{\downarrow}{\dot{\mathbf{x}}},\overset{\downarrow}{\tilde{\dot{\mathbf{x}}}},y,\tilde{y},\dot{y},\tilde{\dot{y}}\big) \quad ,$$

where

$$\begin{aligned} \tilde{\dot{y}} &= \left(\nabla^2 F(\mathbf{x})\cdot\dot{\mathbf{x}}\right)^{\intercal}\cdot\tilde{\mathbf{x}} + \nabla F(\mathbf{x})\cdot\tilde{\dot{\mathbf{x}}} \quad , \\ \dot{y} &= \nabla F(\mathbf{x})\cdot\dot{\mathbf{x}} \quad , \\ \tilde{y} &= \nabla F(\mathbf{x})\cdot\tilde{\mathbf{x}} \quad \text{and} \\ y &= F(\mathbf{x}) \quad . \end{aligned} \tag{12}$$

Again, dcc generates exactly the implementation of the mathematical model. As we see in (12), the term $\nabla F(\mathbf{x})\cdot\tilde{\dot{\mathbf{x}}}$ must be equal to 0 in order to accumulate the entries of the Hessian $\nabla^2 F$. As a consequence, $\tilde{\dot{\mathbf{x}}}$ must be set to 0 on input. The product $\left(\nabla^2 F(\mathbf{x})\cdot\dot{\mathbf{x}}\right)^{\intercal}\cdot\tilde{\mathbf{x}}$ represents a projection of the Hessian, determined by the vectors $\dot{\mathbf{x}}$ and $\tilde{\mathbf{x}}$. In our case with $m = 1$ the Hessian $\nabla^2 F \in \mathbb{R}^{n\times n}$ has n^2 entries.

To compute the entry $\nabla F_{i,j}$ of the Hessian the vectors $\tilde{\mathbf{x}}$ and $\dot{\mathbf{x}}$ have to be set to the i-th and j-th Cartesian basis vectors, respectively. In order to accumulate the whole Hessian this step has to be repeated for each entry, yielding a computational complexity of $\mathcal{O}(n^2)\cdot cost(F)$. Taking either adjoint or tangent-linear first-order input code, we reapply dcc by invoking dcc -t -d 2 t1_foo.cpp. This tells dcc to generate second-order (-d 2) tangent-linear (-t) derivative code while avoiding internal namespace clashes.

Looking at the possible combinations of the two differentiation models, there exist another three second-order models. We may either apply the adjoint model to the tangent-linear code or apply the adjoint mode to the adjoint code. We will focus on the model where tangent-linear mode is applied to the adjoint code, called *tangent-linear over adjoint* mode.

This time the adjoint code (7) is taken as the input for the reapplication of the tangent-linear mode, obtaining

$$\bar{F}\big(\overset{\downarrow}{\mathbf{x}},\overset{\downarrow}{\bar{\mathbf{x}}},y,\overset{\downarrow}{\bar{y}}\big) \xrightarrow{\texttt{dcc}} \dot{\bar{F}}\big(\overset{\downarrow}{\mathbf{x}},\overset{\downarrow}{\dot{\mathbf{x}}},\overset{\downarrow}{\bar{\mathbf{x}}},\overset{\downarrow}{\dot{\bar{\mathbf{x}}}},y,\dot{y},\overset{\downarrow}{\bar{y}},\overset{\downarrow}{\dot{\bar{y}}}\big) \quad ,$$

where

$$\begin{aligned} \dot{y} &= \nabla F(\mathbf{x})\cdot\dot{\mathbf{x}} \quad , \\ y &= F(\mathbf{x}) \quad , \\ \dot{\bar{\mathbf{x}}} &= \dot{\bar{\mathbf{x}}} + \dot{\mathbf{x}}^{\intercal}\cdot\nabla^2 F(\mathbf{x})\cdot\bar{y} + \nabla F(\mathbf{x})^{\intercal}\cdot\dot{\bar{y}} \quad \text{and} \\ \bar{\mathbf{x}} &= \bar{\mathbf{x}} + \nabla F(\mathbf{x})^{\intercal}\cdot\bar{y} \quad . \end{aligned} \tag{13}$$

The generated implementation computes the term $\dot{\mathbf{x}}^{\intercal}\cdot\nabla^2 F(\mathbf{x})\cdot\bar{y}$. This time we do not end up with one single entry, but we are able to harvest one complete row $\nabla^2 F_i$ of the Hessian in $\dot{\mathbf{x}}$. To achieve this, the term $\nabla F(\mathbf{x})^{\intercal}\cdot\dot{\bar{y}}$ and thus $\dot{\bar{y}}$ must be set to 0 on input. The scalar $\bar{y}$ must be set to 1. Finally to compute a row of the

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-139-7

n	250	500	1000	2000
f (s)	0.03	0.08	0.15	0.32
TLM (s)	33	109	457	1615
ADJ (s)	0.21	0.43	0.85	1.82
TLM-ADJ (s)	150	587	2286	8559
IDS size	7500502	15001002	30002002	60004002
FDS size	5000002	10000002	20000002	40000002
CS size	7500503	15001003	30002003	60004003

Table I: Time and memory requirements for gradient computation

Hessian $\nabla^2 F_i$, $\dot{x}$ must be set to the i-th Cartesian basis vector. As such, we have to rerun this model n times in order to accumulate the whole Hessian, yielding only a linear increase in runtime complexity of $\mathcal{O}(n) \cdot cost\,(F)$.

The desired `dcc` command is `dcc -a -d 2 t1_foo.cpp` resulting in the file `a2_t1_foo.cpp`. The option `-a` instructs `dcc` to generate adjoint code.

V. Case Study

As discussed in Section I, we run a test case on an inverse problem based on Burgers' equation (1). As a start we take the code presented in [2] implementing the original function with the signature of

```
1 void f(int n, int nt, double& cost, double**
      u, double* ui...)
2 {
3 ...
4 }
```

Listing 6: Signature of Burgers' function

Taking `n` grid points of `ui` as the initial conditions we integrate over `nt` timesteps. The values are saved in the two dimensional array `u` for each grid point `i` and time step `j`.

To solve the inverse problem we need the derivatives of `cost` with respect to the initial conditions `ui`.

The results in Table I represent the runtime of one full gradient accumulation as well as the memory requirements in adjoint and tangent-linear mode. Additionally one Hessian accumulation is performed using the tangent-linear over adjoint model (13). Different problem sizes are simulated with varying `n`. We also mention the different stack size shown in Section III.

If we assume four bytes per integer and control stack element plus eight bytes for a floating data stack element we end up with a memory requirement of $\approx$ 610 MB for the Hessian accumulation. The tests were running on a GenuineIntel computer with Intel(R) Core(TM)2 Duo CPU and with 2000.000 MHz cpu.

The execution time of the tangent-linear gradient computation is growing proportionally to the problem size `nx` and the execution time of `f`:

$$FM : \frac{cost(F')}{cost(F)} \sim \mathcal{O}(n).$$

The single execuiton of *t1_f* takes approximately twice so much as the execution of *f*.

The execution time of the adjoint gradient computation is growing only proportional to the execution time of `f`:

$$AM : \frac{cost(F')}{cost(F)} \sim \mathcal{O}(1).$$

Finally we accumulate the Hessian using tangent-linear over adjoint mode. Here, the runtime is growing linearly with respect to `n` as well as `f` since the dimension of the dependent `cost` is equal to 1.

$$FM - AM : \frac{cost(F'')}{cost(F)} \sim \mathcal{O}(n).$$

For scalar functions in particular, the runtime complexity for accumulating the Hessian using AD is the same as the runtime complexity of the gradient accumulation using finite difference. This enables developers to implement a second-order model where a first-order model has been used so far.

VI. Outlook & Conclusion

We have presented a source transformation compiler for a restricted subset of C/C++. As such, `dcc` runs on any system with a valid C/C++ compiler making it a very portable tool. Its unique reapplication feature allows code to be transformed up to any order of differentiation.

Additionally, several extensions were implemented. As these programs run on cluster systems, they often rely on parallelization techniques. The most widely used parallelization method is MPI. Hence, there is a need for adjoint MPI enabled code [10]. This feature has been integrated into `dcc` using an adjoint MPI library [11]. Additionally there are attempts to achieve the same goal with OpenMP [12]. For the sake of brevity we did not mention the program analysis `dcc` performs. For better efficiency, `dcc` uses *activity* and *TBR* analyses [13].

References

[1] D. Zwillinger, *Handbook of Differential Equations, 3rd ed.* Boston, MA: Academic Press, 1997.

[2] E. Kalnay, "Atmospheric modeling, data assimilation and predictability," 2003.

[3] T. Kelley, *Solving nonlinear equations with Newton's method*, ser. Fundamentals of Algorithms. Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM), 2003.

[4] A. Tikhonov, "On the stability of inverse problems," *Dokl. Akad. Nauk SSSR*, vol. 39, no. 5, pp. 195–198, 1943.

[5] A. Griewank and A. Walter, *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation (2nd Edition)*. Philadelphia: SIAM, 2008.

[6] G. Corliss and A. Griewank, Eds., *Automatic Differentiation: Theory, Implementation, and Application*, ser. Proceedings Series. Philadelphia: SIAM, 1991.

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-139-7

[7] G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, Eds., *Automatic Differentiation of Algorithms – From Simulation to Optimization*. New York: Springer, 2002.

[8] U. Naumann, "Dag Reversal is NP-complete," *Journal of Discrete Algorithms*, vol. 7, no. 4, pp. 40–410, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/B758J-4THC1FD-2/2/7ddfc2eab484bbe184d4dcdf16d8e58a

[9] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers. Principles, Techniques, and Tools (Second Edition)*. Reading, MA: Addison-Wesley, 2007.

[10] P. Hovland and C. Bischof, "Automatic differentiation for message-passing parallel programs," in *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International ... and Symposium on Parallel and Distributed Processing 1998*, mar-3 apr 1998, pp. 98 –104.

[11] M. Schanen, U. Naumann, and M. Förster, "Second-order adjoint algorithmic differentiation by source transformation of mpi code," in *Recent Advances in the Message Passing Interface, Lecture Notes in Computer Science*. Springer, 2010, pp. 257–264.

[12] OpenMP Architecture Review Board, "OpenMP Application Program Interface," Specification, 2008. [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf

[13] L. Hascoët, U. Naumann, and V. Pascual, "To-be-recorded analysis in reverse mode automatic differentiation," *Future Generation Computer Systems*, vol. 21, pp. 1401–1417, 2005.

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-139-7