Optimal Checkpointing for Time-Stepping Procedures in ADOL-C[⋆]

Andreas Kowarz and Andrea Walther

Institute of Scientific Computing, Technische Universität Dresden,
01062 Dresden, Germany
{Andreas.Kowarz, Andrea.Walther}@tu-dresden.de

Abstract. Using the basic reverse mode of automatic differentiation, the memory needed for the computation of discrete adjoints is proportional to the number of operations performed. This behavior is frequently not acceptable, especially for large-scale problems that involve a kind of time-stepping procedure. Therefore, we integrate a checkpointing mechanism into ADOL-C, a tool for the automatic differentiation of C and C++ programs. This checkpointing procedure is optimal for a given number of checkpoints in the sense that it yields the minimal number of recomputations. The resulting effects on the run-time behavior are illustrated by means of the derivative computation for an ODE-based optimization problem.

1 Introduction

The reverse mode of automatic differentiation (AD) provides an efficient method to compute discrete adjoint information. For example, the operation count for computing the gradient of a scalar-valued function is only a small multiple of the operation count needed to evaluate the function [3]. However, the memory needed by the reverse mode in its basic form is proportional to the operation count of the function evaluation. For real-world problems this fact may lead to an unacceptable memory requirement. For that reason, several checkpointing approaches 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 to reduce the overall memory requirement. This simple approach is provided for example by the AD-tool TAF [1]. As alternative, one may exploit the call graph structure of the function evaluation to place checkpoints at the entries of specific subroutines. This so-called joint reversal, see, e.g., [3], then leads to a reduction of the memory requirement. The subroutine-oriented checkpointing is used for example by the AD-tools Tapenade [8] and OpenAD [11]. As soon as one can exploit additional information about the structure of the function evaluation, an appropriate adapted checkpointing strategy can be used. This is in particular the case if a time-stepping procedure is contained in the function evaluation allowing the usage of a time-stepping oriented checkpointing. If the number of

⋆ Partially supported by the DFG grant WA 1607/2-1.

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 was shown in [12] that this approach is not optimal. A more advanced but still not optimal approach is the binary checkpointing used for example in [10]. However, optimal checkpointing schedules can be computed in advance to achieve an optimal, i.e. minimal, run time increase for a given number of checkpoints [2, 5]. In this paper, we present the usage of the optimal, also called binomial, checkpointing within the AD-tool ADOL-C [4] to obtain a drastic decrease in the memory requirement by taping only one instead of l time steps. For the considered numerical example the memory reduction leads even to an overall run-time reduction due to the reduced access cost for the required memory. Hence, we face for the first time the situation where checkpointing leads to a decrease in run-time despite the fact that a considerable amount of intermediate information has to be recomputed.

The structure of this paper is the following: In Sect. 2, we present the adapted implementation of ADOL-C to employ the time-stepping oriented checkpointing. This includes a description of the modified internal function representation based on nested taping. Subsequently, the derivative computation exploiting binomial checkpointing is illustrated. This section ends with a short introduction to the usage of the new checkpointing facility. Section 3 discusses possible run-time effects using as example the derivative computation for an ODE-based optimization problem. Finally, a conclusion and an outlook are given in Sect. 4.

2 Optimal Checkpointing in ADOL-C

The AD-tool ADOL-C uses operator overloading for the automatic differentiation of function evaluations $y = F(x)$ written in C or C++. For this purpose, the new class adouble is introduced by ADOL-C. The user has to declare the independent variables x and all quantities that directly or indirectly depend on them of type adouble. Other variables that do not depend on the independent variables but enter, for example, as parameters, may remain one of the passive types double, float, or int. During the function evaluation with adouble variables, ADOL-C stores for each operation the corresponding operator and the variables that are involved into a data structure called *tape*. Once, the tape, i.e., the internal representation is generated, the required derivatives are calculated on the basis of the internal function representation using the elemental differentiation rules. Due to the described approach, the derivative calculations involve a possibly substantial but always predictable amount of data that is accessed strictly sequentially. The size of the much smaller randomly accessed memory can be precalculated using information on the tape.

However, this approach may lead to the storage of a significant amount of redundant information if the function evaluation contains a time-stepping procedure. To overcome this difficulty, the next version of ADOL-C will provide a checkpointing facility based on the binomial checkpointing procedure revolve [5]. To exploit this additional functionality, the number l of time steps in the time-stepping part must be known before this iterative process starts. Furthermore, the user has to provide the number c of checkpoints that can be stored in an internal data structure of ADOL-C. As shown in [5], the applied checkpointing strategy is optimal if the computational costs of the time steps are identical or almost identical. Nevertheless, numerical tests presented for example in [7] show that the checkpointing procedure provided by revolve also yields surprisingly good results for varying computational costs of the time steps. Therefore, it is also feasible to use the provided checkpointing possibility in these cases.

To apply the checkpointing routine revolve inside of ADOL-C, the generation of the internal representation, i.e. the taping mechanism, as well as the derivative evaluation have to be adapted as described in the next paragraphs.

Generating the Internal Function Representation: Nested Taping

For illustrating the modified taping mechanism, we assume that the function evaluation $y = F(x)$ consists of three parts: A start up calculation G, the time-stepping procedure H, and a final calculation evaluating for example a target function J, i.e., $y = J \circ H \circ G(x)$ with

$$u = G(x), \quad v = H(u), \quad y = J(v)$$

and the evaluation of $H(u)$ is given by

$$u_1 = \tilde{H}(u), \; \ldots, \; u_i = \tilde{H}(u_{i-1}), \; \ldots, \; u_l = \tilde{H}(u_{l-1}), \; v = u_l \; .$$

The corresponding consequences for the tape generation using the "basic" taping approach as implemented in ADOL-C so far are shown in the left part of Fig. 1. As can be seen, the iterative process is completely unrolled due to

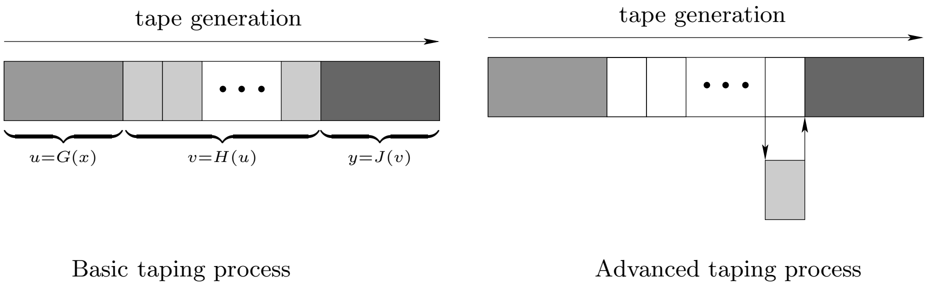

Fig. 1. Different taping approaches

the taping process. That is, the tape contains an internal representation of each time step. Hence, the overall tape comprises a serious amount of redundant information as illustrated by the light gray rectangles in Fig. 1.

To overcome the repeated storage of essentially the same information, recently we incorporated a *nested taping* mechanism into ADOL-C as illustrated on the right-hand side of Fig. 1. This new capability allows the encapsulation of the time-stepping procedure such that only the last time step $u_l = \tilde{H}(u_{l-1})$ is taped as one representative of the time steps. This is illustrated by the white rectangles where only the function is evaluated but no taping is performed. Only the last time step is taped as illustrated by the light gray rectangle. Additionally, a function pointer to the evaluation procedure $\tilde{H}$ is stored for a possibly necessary retaping during the derivative calculation as explained below. Furthermore, c checkpoints are distributed by revolve. It is important to note that the overall tape size is drastically reduced due to the advanced taping strategy. For the implementation of this nested taping we introduced a so-called “differentiating context” that enables ADOL-C to handle different internal function representations during the taping procedure and the derivative calculation. This approach allows the generation of a new tape inside the overall tape, where the coupling of the different tapes is based on the *external differentiated function* concept presented at the AD2004 conference.

Computing the Derivative Information

Applying the reverse mode of AD to the function $y = F(x)$ consists of three steps: First the discrete adjoint $\bar{v} = \bar{J}(v, \bar{y})$ of $J(v)$ is calculated, then the discrete adjoint $\bar{u} = \bar{H}(u, \bar{v})$ of $H(u)$, and finally the discrete adjoint $\bar{x} = \bar{G}(x, \bar{u})$ of G. The computation of $\bar{v}$ and $\bar{x}$ is straightforward and implemented in the current version of ADOL-C. The situation changes completely for the computation of $\bar{u}$ due to the usage of a checkpointing approach for the time-stepping procedure and the resulting nested taping. For calculating the discrete adjoint $\bar{u}$ of the time-stepping part, the routine revolve is used to steer the reverse mode differentiation as illustrated in Fig. 2. For this purpose, the nested taping in com-

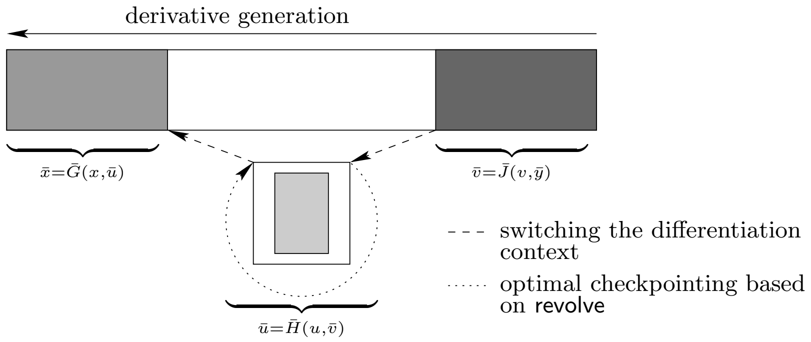
derivative generation
$\bar{x}=\bar{G}(x,\bar{u})$
$\bar{v}=\bar{J}(v,\bar{y})$
$\bar{u}=\bar{H}(u,\bar{v})$
switching the differentiation context
optimal checkpointing based on revolve

Fig. 2. Derivative computation using a checkpointing procedure

bination with the two “differentiating contexts” for the overall function F and the time-stepping part are exploited. This allows a switching from the derivative calculation for F to the more involved derivative computation for the time-stepping procedure based on a checkpointing strategy for the subfunction H. In the ideal case, the adjoint computation for H consists only of recomputations of intermediate states u_i and derivative computations based on the internal representation of one time step according to the reversal schedule provided by revolve for the given number c of checkpoints. This approach yields in the case of constant computational costs for all time steps a minimal run-time for the checkpointing approach. If the control flow changes during the time step evaluation due to a change of an adouble value, a "retaping" of the time-step function $\tilde{H}$ is automatically initiated for a corresponding update of the internal representation. Further information on the validity of tapes can be found in subsection 2.6 of the ADOL-C manual. Alternatively, an additional flag set by the user to the value 1 causes the retaping of each time step to take changes of non-adouble values into account.

The next version of ADOL-C will provide this checkpointing capability also for higher derivative computations and the vector mode although probably the most common usage will be the computation of gradient information using the scalar reverse mode of AD.

Interface of the Checkpointing Facility

Written under the objective of a lean and concise interface, the checkpointing routines of ADOL-C need only very limited information. The user must provide two routines as implementation of the time-stepping function $\tilde{H}$ with the signatures

```
int time_step_function(int n, adouble *u);
int time_step_function(int n, double *u);
```

where the function names can be chosen by the user as long as the names are unique. Furthermore, it is assumed that the result vector of one time step iteration overwrites the argument vector of the same time step. Therefore, no copy operations are required to prepare the next time step.

At first, the adouble version of the time step function has to be *registered* using the C or C++ interface, respectively.

```
C:      CpInfos *cpInfos = reg_timestep_fct(ADOLC_TimeStepFunction);
C++:    CP_Context cpc(ADOLC_TimeStepFunction);
```

Using either cpInfos or cpc and the appropriate interface the user has to provide the remaining checkpointing information:

- a pointer to the double version of the time step routine,
- the number of time steps l,
- the number of checkpoints c,
- the tape number to be used internally for the nested taping,
- the dimension of the argument vector u,
- a pointer to the argument vector storing the initial value of u and
- a pointer to the result vector storing the final value v of the time integration.

In addition a flag for enforcing the retaping of every timestep might be set. Then, the nested taping and the derivative calculation using binomial checkpointing is initiated by calling the ADOL-C function

```
C:      info = checkpointing(cpInfos);
C++:    info = cpc.checkpointing();
```

at the corresponding point of the function evaluation during the taping process. Subsequently, ADOL-C computes derivative information using the optimal checkpointing strategy provided by revolve internally, i.e., completely hidden from the user.

3 Numerical Example

The numerical example that serves to illustrate the run-time effects of the checkpointing procedure is an industrial robot as depicted in Figure 3 that has to perform a fast turn-around maneuver. We denote by $q = (q_1, q_2, q_3)$ the angular coordinates of the robot's joints, q_1 referring to the angle between the base and the two-arm system. The robot is controlled via three control functions u_1 through u_3, denoting the respective angular momentum applied to the joints (from bottom to top) by electrical motors. The control problem under consideration is to minimize the energy-related objective

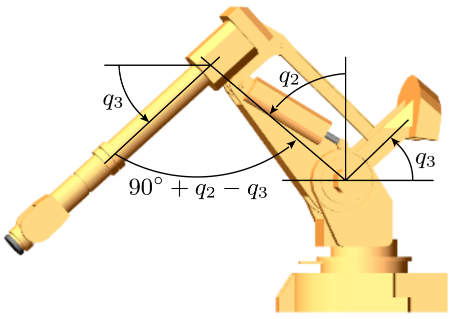

Fig. 3. Industrial robot ABB IRB 6400

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

where the final time t_f is given. The robot's dynamics obeys a system of three differential equations of second order:

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

where $M(q)$ is a 3×3 symmetric positive definite matrix containing moments of inertia, called a generalized mass matrix. The vector v is composed of centrifugal and Coriolis force entries, and w contains the gravitational influence. Finally, we allow for forces induced by dry friction and reset forces by means of τ_{friction} and τ_{reset}, respectively. The complete equations of motion can be found in [9]. The robot's task to perform a turn-around maneuver is expressed by means of initial and terminal conditions as well as control constraints [6]. However, for illustrating the run-time effects of the checkpointing facility integrated in ADOL-C we consider only the gradient computation of $J(q,u)$ with respect to u.

To compute an approximation of the trajectory x, we apply for the integration the standard Runge-Kutta method of order 4 resulting in about 800 lines of code. The integration and derivative computations were computed using an AMD Athlon64 3200+ (512 kB L2-cache) and 1GB main memory. The resulting averaged run-times in seconds for one gradient computation are shown in Fig. 4, where the run-time required for the derivative computation without checkpointing, i.e., the basic approach (BA), is illustrated by a dotted line. The run-time

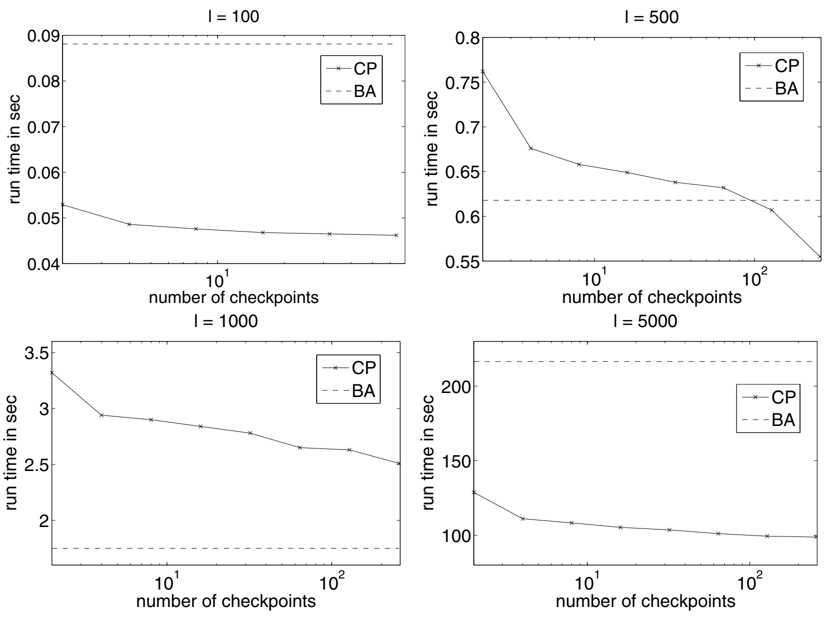

Fig. 4. Comparison of run-times for $l = 100, 500, 1000, 5000$

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 memory requirement, Table 1 shows the tape sizes for the basic approach as well as the tape and checkpoint sizes required by the checkpointing version. The tape size for the later varies since the number of independents is a multiple of the number of time steps l due to the distributed control u. 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 Fig. 4. The smaller runtime for the basic approach completes the setting. However, the more interesting cases from this example are $l = 100$ and $l = 5000$, respectively. In these situations a smaller runtime was achieved even though checkpointing was used. These results are effected 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 mentioned cases of the robot example the computation could

Table 1. Memory requirements for $l = 100, 500, 1000, 5000$

# 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

be redirected from the main memory mainly into the L2-cache of the processor ($l = 100$) and from at least partially hard disk access completely into the main memory ($l = 5000$). The last case from Fig. 4 ($l = 500$) depicts a situation where only the tape and a small number of the most recent checkpoints can be kept within the L2-cache. A well chosen ratio between l and c in this case causes a significantly smaller recomputation rate and results in a decreased overall runtime, making the checkpointing once more preferable.

4 Conclusion and Outlook

In this paper, we present a new nested taping approach incorporated into the AD-tool ADOL-C. This new strategy allows a compressed internal representation for time stepping procedures in combination with a checkpointing approach for the derivative calculation. In addition to the drastic decrease in memory requirement due to the nested taping, the new capability may even lead to a reduction of the overall run-time for such cases where the reduced memory access costs compensate the required recomputations.

We plan to employ the nested taping also for an efficient differentiation of fixpoint iterations, where the computation of gradient information can be based only on the last iteration performed during the function evaluation. Hence, also in this situation a complete unrolling and hence the storage of the full internal representation can be avoided. Therefore, also in these situations the applicability of ADOL-C will be extended.

References

1. Giering, R., Kaminski, T.: Recipes for Adjoint Code Construction. ACM Trans. Math. Software, **24** (1998) 437–474.
2. Griewank, A.: Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. Optim. Methods Softw., **1** (1992) 35–54.
3. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, 2000.
4. Griewank, A., Juedes, D., Utke. J.: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. ACM Trans. Math. Software, **22** (1996) 131–167.
5. Griewank, A., Walther, A.: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. ACM Trans. Math. Software, **26** (2000) 19–45.
6. R. Griesse, A. Walther.: Parametric sensitivities for optimal control problems using automatic differentiation. *Optimal Control Applications and Methods*, Vol. 24, pp. 297–314 (2003).
7. Hinze, M., Sternberg, J.: A-Revolve: An adaptive memory and run-time-reduced procedure for calculating adjoints; with an application to the instationary Navier-Stokes system. Optim. Methods Softw., **20** (2005) 645–663.
8. Hascoët, L., Pascual, V.: Tapenade 2.1 user's guide. Tech. rep. 300, INRIA, 2004.
9. Knauer, M., Büskens, C.,: *Real-Time Trajectory Planning of the Industrial Robot IRB6400.* PAMM. 3, 2003, 515–516.

10. Kubota, K.: A Fortran 77 preprocessor for reverse mode automatic differentiation with recursive checkpointing. Optim. Methods Softw., **10** (1998) 319–335.
11. Naumann, U., Utke, J., Lyons, A., Fagan, M.: Control Flow Reversal for Adjoint Code Generation. Proceed. of SCAM 2004, IEEE Computer Society (2004) 55–64.
12. Walther, A., Griewank, A.: Advantages of binomial checkpointing for memory-reduced adjoint calculations. In: Feistauer, M., et al., (eds.): *Numerical mathematics and advanced applications,* Proc. ENUMATH 2003, Springer (2004) 834–843.