Automatic Sparsity Detection Implemented as a Source-to-Source Transformation

Ralf Giering and Thomas Kaminski

FastOpt, Schanzenstr. 36, 20357 Hamburg, Germany
http://www.FastOpt.com

Abstract. An implementation of Automatic Sparsity Detection (ASD) as a new source-to-source transformation is presented. Given a code for evaluation of a function, ASD generates code to evaluate the sparsity pattern of the function's Jacobian by operations on bit-vectors. Similar to Automatic Differentiation (AD), there are forward and reverse modes of ASD. As ASD code has significantly fewer required variables than AD, ASD should be operated in pure mode, i.e. without an evaluation of the underlying function included in the ASD code. In a performance comparison of ASD to AD on five small test problems, ASD is about two orders of magnitude faster than AD. Hence, for a particular class of sparse Jacobians, it is efficient to determine first the sparsity patten via ASD. In a subsequent AD step, this allows to reduce the effective dimension for the evaluation of the Jacobian by avoiding the evaluation of zero elements via a selection of seed matrices according to the sparsity pattern.

1 Introduction

Automatic Differentiation (AD) generates derivative code for evaluation of the Jacobian matrix that corresponds to a given code for evaluation of a function. Often the Jacobian is sparse, and, if this sparsity information is available, it can be exploited to compute the Jacobian more efficiently. The basic idea is that evaluation of Jacobian entries that are known to be zero is not necessary, which may allow to reduce the effective dimension for the Jacobian evaluation. Algorithms for exploiting Jacobian sparsity have been developed and demonstrated by Curtis Powell Reid (CPR) [6], Newsam and Ramsdell [13, 7], and Coleman and Verma (bi-coloring, [4]), details can be found in the respective references.

In some cases the sparsity structure is not known or changes with the input. Bischof et al. [3] describe a dynamical approach of tracking the sparsity structure (via calls to special bookkeeping routines of an extra library) during the evaluation of forward mode derivative code. Within an operator overloading framework, Geitner et.al. [7] describe how to determine the sparsity pattern by re-executing the tape, a representation of the underlying function, generated in a previous execution of the function code. The tape is built by overloading every operation to store the operands and the operation. The sparsity is represented by bit patterns and combined by logical 'or' operations.

Here we describe the source-to-source equivalent, that is, a semantical transformation of the original function code to a code that evaluates the sparsity

V.N. Alexandrov et al. (Eds.): ICCS 2006, Part IV, LNCS 3994, pp. 591–598, 2006.

structure of the function's Jacobian. This transformation has been implemented in TAF (Transformation of Algorithms in Fortran, [8, 11]). TAF is an AD-tool for Fortran77-95 programs. It normalises the function code and applies a control flow analysis in order to replace old style Fortran constructs and to transform unstructured code to high-level structures. Irreducible control flow graphs are made reducible by node copying [10]. An intra-procedural data dependence analysis is applied to determine loop-carried flow, anti-flow, or output dependences. The following inter-procedural data flow analysis computes the IN and OUT sets [5] of all statements based on the given dependent and independent variables of the top-level routine. A variable is active if it depends on the independent variables and influences the dependent variables [2]. Derivative (AD) or bit-vector (ASD, see below) variables are only built for active variables, and derivative or sparsity code is only generated for active statements, i.e. statements that compute active variables. In reverse modes of AD and ASD, TAF generates recomputations for required variables by an extension of the Efficient Recomputation Algorithm (ERA [9]). ERA uses demand-driven program slicing to generate only a minimum of recomputations.

Depending on the number of independent and dependent variables, ASD is applied in forward or reverse mode to compute the Jacobian's sparsity. In the presence of a priori knowledge about the sparsity structure of a Jacobian on a block level, it is most efficient to restrict ASD to a subset of all blocks.

2 Transformation Rules

In this section we present the rules of transforming the function code into both types of ASD codes, the forward and the reverse one. To keep the notation simple, the rules are shown for computing the sparsity structure of a boolean vector times Jacobian product in forward mode and a boolean Jacobian times vector product in reverse mode. In the following f, x, y are active variables with corresponding boolean variables $\hat{f}, \hat{x}, \hat{y}$, which hold the sparsity structure that is propagated by the ASD code. In order to compute the full sparsity pattern in the above boolean product the vectors are replaced by the boolean identity matrices (true on the diagonal). In the transformation rules given below, the boolean variables are then to be replaced by boolean vectors.

For a binary operation $\circ \in \{+, -, *, /, **\}$, the assignment

$$f = x \circ y$$

is transformed by forward mode ASD to:

$$\hat{f} = \hat{x} \vee \hat{y}$$

and by reverse mode ASD to:

$$\begin{aligned} \hat{x} &= \hat{x} \vee \hat{f} \\ \hat{y} &= \hat{y} \vee \hat{f} \\ \hat{f} &= false\ , \end{aligned}$$

where $\vee$ denotes the logical 'or'. For a unary operation $\circ \in \{-, +\}$ the assignment

$$f = \circ x$$

is transformed by forward mode ASD to:

$$\hat{f} = \hat{x}$$

and by reverse mode ASD to:

$$\hat{x} = \hat{x} \vee \hat{f}$$
$$\hat{f} = false\ .$$

Similar rules apply for a function invocation. For a function of one argument (e.g. sin, cos, ...), the assignment:

$$f = func(x)$$

is transformed by forward mode ASD to:

$$\hat{f} = \hat{x}$$

and by reverse mode ASD to:

$$\hat{x} = \hat{x} \vee \hat{f}$$
$$\hat{f} = false$$

It is evident that in ASD, unlike AD, the transformed statements do not require any values from the original statements. Only to follow (forward mode) or to reverse (reverse mode) the control flow, values may be required (if-then-else and case constructs). They are provided in the same fashion as for AD (see [8]). Owing to the reduced number of required values, the pure forward and pure reverse modes, which compute only sparsity and do not evaluate the function itself, have a considerable advantage in efficiency. In TAF both pure modes are implemented for AD and ASD. A command line option triggers generation of the corresponding codes.

3 Implementation

The ASD implementation in TAF propagates the sparsity structure in bit-vectors. Depending on the platform used, a Fortran-90 bit-vector is an integer variable that holds 32 bits, if the kind parameter is 4 (byte), or 64 bits, if the kind parameter is 8 (byte). [1]

This way several matrix vector products are computed simultaneously. The logical operation 'or' is implemented by the IOR intrinsic function. It has two bit-vector arguments and a bit-vector result. Inside the bit-vector the value 'false' is

[1] For most efficient code the bit-vector should be as long as a word of the processor.

represented by a zero bit. A false bit-vector (all bits are false) is represented by 0 and a true one by NOT(0), where NOT is the Fortran-90 intrinsic function. For initialisation individual bits are set by IBSET and for interpretation of the results they are tested by BTEST, both of which are Fortran-90 intrinsic functions.

As an example we use the single assignment

$$f = a * x + y * sin(z)$$

which computes a new value for the variable z. It is assumed that only the variables x,y,z, and f are active.

In forward mode, ASD generates the assignment

$$sf = IOR(sx, IOR(sy, sz)) \ ,$$

where sf is the bit-vector corresponding to the active variable f. Other variable names are generated accordingly. In some cases the RHS expression can be simplified by applying the rules of boolean operations. Here one bit-vector corresponds to one active variable and the code computes 32 (64) matrix times vector products simultaneously. In order to compute more vectors, a bit-vector array is generated. The statements remain unchanged, since Fortran-90 elemental intrinsic functions[2] operate on scalars and arrays.

In reverse mode ASD generates the sequence of assignments

$$\begin{aligned} sx &= IOR(sx, sf) \\ sy &= IOR(sy, sf) \\ sz &= IOR(sz, sf) \\ sf &= 0 \end{aligned}$$

Similar to AD, the bit-vector to the LHS variable f is reset after the bit-vectors of all RHS variables have been updated.

4 Performance

The Minpack-2 collection [1] provides several test function codes based on real physical problems. Codes to evaluate their Jacobian, Jacobian vector products, and Jacobian sparsity are also provided. We have selected five problems of this collection to compare the performance of automatically generated ASD and AD codes:

FDC	flow in a driven cavity
FIC	flow in a channel
IER	incompressible elastic rods
SFD	swirling flow between disks
SFI	solid fuel ignition

[2] For transformational intrinsic functions such as MATMUL more complex statements must be generated.

Each function code solves a differential equation on a grid of variable size. Both the numbers of input and output variables are set equal to the number of grid points (N), i.e. the Jacobian is quadratic. The number of floating point operations in the function code and both dimensions of the Jacobian scale with N.

For each test code, TAF was first used in AD mode to generate code evaluating the full Jacobian (without any seeding) in forward and reverse modes. Next TAF's ASD mode was applied to generate forward and reverse mode codes evaluating directly the Jacobians' sparsity patterns. The codes were compiled with -O3 by the Lahey Fortran-95 compiler and run on an Athlon Linux PC for different values of N. In the code bit-vectors were represented by default integer variables (kind=4).

Fig. 1 shows the relative run-times for these cases. The runtime of ASD code is about two orders of magnitude faster than than that of AD code evaluating the full Jacobians. This is not only a consequence of the simultaneous propagation of 32 logical values by the operations on bit-vectors and of ASD's lower number of operations. An additional performance gain is achieved by the capability of TAF to generate code operating in pure forward and reverse modes. It is worth noting that the advantage of ASD over AD is almost always bigger in forward mode. Presumably this is owing to the smaller number of IOR operations required by forward model ASD as compard to reverse mode ASD, which is also illustrated by the example of section 3. Fig. 2 shows the ratio of run times for reverse and forward modes of AD, which don't change much with problem size.

The Carbon Cycle Data Assimilation System (CCDAS, http://CCDAS.org, [15, 14]) provides an example of a large-scale ASD application. CCDAS uses observed atmospheric carbon dioxide to constrain parameters in a global model of the terrestrial biosphere. A subset of the parameters are chosen to be specific to the model's plant functional types (PFTs), e.g. tundra. Uncertainties in the observations are back-propagated via the model's Hessian to parameter uncertainties. To map these parameter uncertainties onto uncertainties of target quantities diagnosed from or prognosed by the model, e.g. the land sink over a particular region, the corresponding Jacobian is needed. Now, a given region will typically only host a subset of the global model's PFTs. Consequently the sensitivity of the land sink over that region to parameters specific to PFTs not represented in that region must be zero, i.e. this Jacobian will be sparse. [12] demonstrate the use of TAF's ASD mode for the efficient computation of the sparsity structure of the CCDAS Jacobian that quantifies the sensitivity of the mean fluxes over latitudinal bands with respect the the model's parameters. While the cost of a reverse mode AD run increases by about the cost of 0.25 function evaluations per additional target quantity (from about 3.5 function evaluations for 1 target quantity to about 26 function evaluations for 96 target quantities), the cost of the ASD run remains almost constant (at about 2.5 function evaluations).

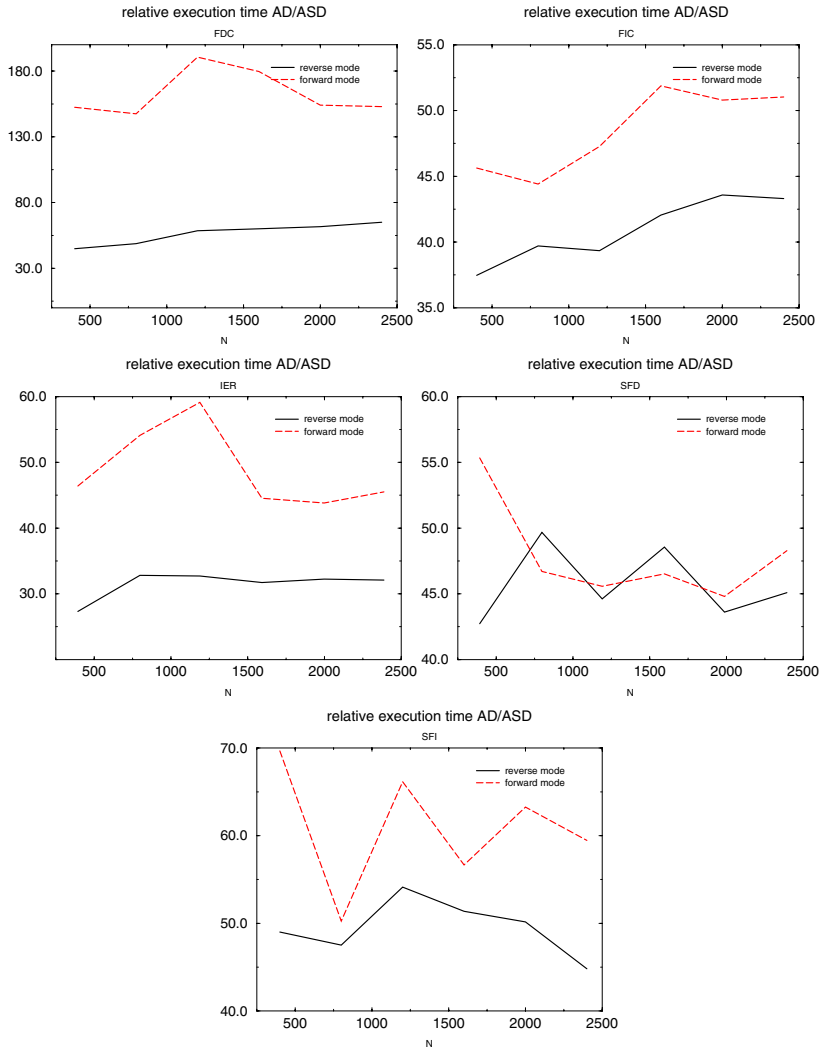

Fig. 1. Ratios CPU time(AD)/CPU time(ASD) for the five test codes over dimension of the problem. Solid lines show reverse mode ratios; dashed lines show forward mode ratios.

5 Conclusions

The rules for the new source transformation Automatic Sparsity Detection (ASD) have been presented. Given an algorithm to evaluate a function an algorithm to evaluate the sparsity pattern of the function's Jacobian is generated. As in Automatic Differentiation (AD) there are two major modes: the forward and the reverse mode. In contrast to AD code, which for its local Jacobians requires values from the function evaluation, ASD code only requires the function's control information.

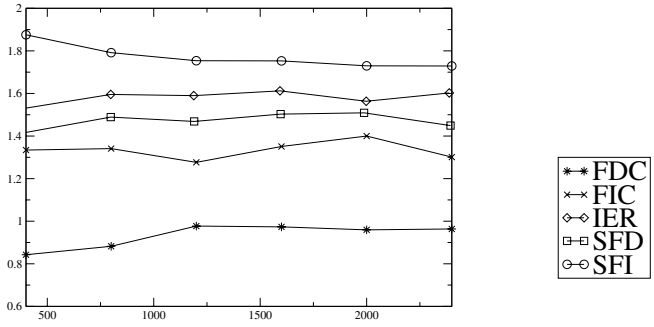

Fig. 2. Ratios CPU time(AD reverse)/CPU time(AD forward) for evaluation of the full Jacobian (without any seeding) of the five test codes over dimension of the problem

The implementation of ASD in Fortran-90 has been described and was explained by a simple example. The run-time of ASD code is about two orders of magnitude faster than that of evaluating the entire Jacobian by AD and checking for sparsity thereafter. The factor is much larger than the expected gain by computing several logical values simultaneously using bit-vectors. The reason for this is the reduced number of operations and the ability of TAF to generate pure mode ASD code, i.e. code that does not include an evaluation of the underlying function itself.

Often from prior knowledge about the function the Jacobian can be partitioned into blocks such that the sparsity structure on a block level is known but within the block is not. In these cases ASD can be restricted to the evaluation of the sparsity structure within the blocks.

References

1. Brett M. Averick, Richard G. Carter, Jorge J. Moré, and Guo-Liang Xue. The MINPACK-2 test problem collection. Preprint MCS–P153–0692, ANL/MCS–TM–150, Rev. 1, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1992.
2. C. 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.
3. Christian H. Bischof, Peyvand M. Khademi, A. Bouaricha, and Alan Carle. Efficient computation of gradients and Jacobians by dynamic exploitation of sparsity in automatic differentiation. *Optimization Methods and Software*, 7:1–39, 1997.
4. Thomas F. Coleman and Arun Verma. Structure and efficient Jacobian calculation. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 149–159. SIAM, Philadelphia, Penn., 1996.
5. Beatrice Creusillet and F. Irigoin. Interprocedural Array Region Analysis. Rapport CRI, A-282, Ecole des Mines de Paris, FRANCE, January 1996.

6. A. R. Curtis, M. J. D. Powell, and J. K. Reid. On the estimation of sparse Jacobian matrices. *J. Inst. Math. Appl.*, 13:117–119, 1974.
7. Uwe Geitner, Jean Utke, and Andreas Griewank. Automatic Computation of Sparse Jacobians by Applying the Method of Newsam and Ramsdell. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques Applications, and Tools*, pages 161–172. SIAM, Philadelphia, Penn., 1996.
8. R. Giering and T. Kaminski. Recipes for Adjoint Code Construction. *ACM Trans. Math. Software*, 24(4):437–474, 1998.
9. R. Giering and T. Kaminski. Recomputations in reverse mode AD. In George Corliss, Andreas Griewank, Christele Fauré, Laurent Hascoet, and Uwe Naumann, editors, *Automatic Differentiation of Algorithms: From Simulation to Optimization*, chapter 33, pages 283–291. Springer Verlag, Heidelberg, 2002.
10. R. Giering and T. Kaminski. Applying TAF to generate efficient derivative code of Fortran 77-95 programs. *PAMM*, 2(1):54–57, 2003.
11. R. Giering, T. Kaminski, and T. Slawig. Applying TAF to a Navier-Stokes solver that simulates an Euler flow around an airfoil. *Future Generation Computer Systems*, 21(8):1345–1355, 2005.
12. T. Kaminski, R. Giering, M. Scholze, P. Rayner, and W. Knorr. An example of an automatic differentiation-based modelling system. In V. Kumar, L. Gavrilova, C. J. K. Tan, and P. L'Ecuyer, editors, *Computational Science – ICCSA 2003, International Conference Montreal, Canada, May 2003, Proceedings, Part II*, volume 2668 of *Lecture Notes in Computer Science*, pages 95–104, Berlin, 2003. Springer.
13. G. N. Newsam and J. D. Ramsdell. Estimation of sparse Jacobian matrices. *SIAM J. Alg. Disc. Meth.*, 4(3):404–417, 1983.
14. P. Rayner, M. Scholze, W. Knorr, T. Kaminski, R. Giering, and H. Widmann. Two decades of terrestrial Carbon fluxes from a Carbon Cycle Data Assimilation System (CCDAS). *Global Biogeochemical Cycles*, 19:doi:10.1029/2004GB002254, 2005.
15. M. Scholze. *Model studies on the response of the terrestrial carbon cycle on climate change and variability.* Examensarbeit, Max-Planck-Institut für Meteorologie, Hamburg, Germany, 2003.