

Implementation of Partial Separability in a Source-to-Source Transformation AD Tool

Sri Hari Krishna Narayanan¹, Boyana Norris¹, Paul Hovland¹, and Assefaw Gebremedhin²

Abstract A significant number of large optimization problems exhibit structure known as *partial separability*, for example, least squares problems, where element functions are gathered into groups that are then squared. The sparsity of the Jacobian (and Hessian) of a partially separable function can be exploited by computing the smaller Jacobians of the elemental functions and then assembling them into the full Jacobian. We implemented partial separability support in ADIC2 by using pragmas to identify partially separable function values, applying source transformations to subdivide the elemental gradient computations, and using the ColPack coloring toolkit to compress the sparse elemental Jacobians. We present experimental results for an elastic-plastic torsion optimization problem from the MINPACK-2 test suite.

Key words: Forward mode, partial separability, ADIC2, ColPack

1 Introduction

As introduced by Griewank and Toint [14], a function $f : \mathbb{R}^n \mapsto \mathbb{R}$ is considered partially separable if f can be represented in the form

$$f(x) = \sum_{i=1}^m f_i(x), \quad (1)$$

where f_i depends on $p_i \ll n$ variables. Bouaricha and Moré [6] and Bischof and El-Khadiri [4], among others, have explored different ways to exploit the sparsity of the Jacobians and Hessians of partially separable functions. To compute the (usu-

¹ Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA, [snarayan,norris,hovland]@mcs.anl.gov. Corresponding author: Sri Hari Krishna Narayanan.

²Purdue University, West Lafayette, IN, USA, agebre@purdue.edu

ally dense) gradient ∇f of f , one can first compute the much smaller (and possibly sparse) gradients of the f_i elementals, then assemble the full gradient of f . This approach can significantly reduce the memory footprint and floating-point operations for overall gradient computation compared with computing dense gradients.

This paper describes the following new capabilities of the ADIC2 source transformation tool.

- Pragma-guided source transformations to perform scalar expansion of the elemental components of partially separable scalar-valued functions.
- Exposing of the sparsity present in the elemental Jacobians.
- Calculation of compressed elemental Jacobians using ColPack.
- Combining of the elementals into the scalar valued result.

1.1 ADIC2

ADIC2 is a source transformation tool for automatic differentiation of C and C++ codes, with support for both the forward and reverse modes of AD [15]. ADIC2 uses the ROSE compiler framework [17], which relies on the EDG C/C++ parsers [10]. ADIC2 is part of the OpenAD framework at Argonne, whose general structure is illustrated in Figure 1. Briefly, the process of transforming the original source code into code computing the derivatives consists of several steps: (1) canonicalization (semantically equivalent transformations for removing features that hamper analysis or subsequent AD transformations); (2) program analysis (e.g., control flow graph construction, def-use chains); (3) generation of the language independent XAIF intermediate representation; (4) AD transformation of the XAIF representation; (5) conversion of the resulting AD XAIF code back to the ROSE intermediate representation; and (5) generation of C/C++ derivative code. The general differentiation process as implemented by ADIC2 is discussed in detail in [15]. To exploit the sparsity of the gradients of partially separable functions, we have implemented several extensions of the AD process, which are described in Section 3.

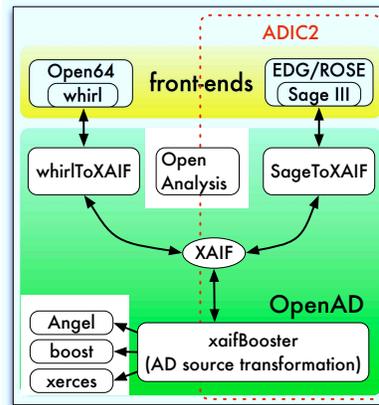


Fig. 1 OpenAD component structure and source transformation workflow.

1.2 ColPack

When a Jacobian (or a Hessian) matrix is sparse, the runtime and memory efficiency of its computation can be improved through *compression* by avoiding storing and computing with zeros. Curtis, Powell, and Reid demonstrated that when two or more columns of a Jacobian are structurally orthogonal, they can be approximated simultaneously using finite differences by perturbing the corresponding independent variables simultaneously [9]. Two columns are structurally orthogonal if there is no row in which both columns have a nonzero. Coleman and Moré showed that the problem of partitioning the columns of a Jacobian into the fewest groups, each consisting of structurally orthogonal columns, can be modeled as a graph coloring problem [8]. The methods developed for finite-difference approximations are readily adapted to automatic differentiation with appropriate initialization of the seed matrix [2]. A survey of the use of graph coloring in derivative computation is available in [12].

ColPack is a software package containing algorithms for various kinds of graph coloring and related problems arising in compression-based computation of sparse Jacobians and Hessians [13]. The coloring problems vary depending on whether the derivative matrix of interest is a Jacobian (nonsymmetric) or a Hessian (symmetric) and whether the derivative matrix is compressed such that the nonzero entries are to be recovered directly (with no additional arithmetic work) or indirectly (by substitution). In ColPack, a nonsymmetric matrix is represented using a *bipartite graph*, and a symmetric matrix is represented using an *adjacency graph*. Thus, a partitioning of the columns of a nonsymmetric matrix into groups of structurally orthogonal columns is obtained using a *distance-2 coloring* of the column vertices of the bipartite graph.

The distance-2 coloring problem, as well as every other coloring problem supported by ColPack, is NP-hard to solve optimally [11, 12]. The corresponding algorithms in ColPack are fast, yet effective, greedy heuristics [13]. They are greedy in the sense that vertices are colored sequentially one at a time and the color assigned to a vertex is never changed. The number of colors used by the heuristic depends on the *order* in which vertices are processed. Hence, ColPack contains implementations of various effective ordering techniques for each of the coloring problems it supports. ColPack is designed in a modular, object-oriented fashion and implemented in C++ for ease of extensibility and portability.

The rest of this paper is organized as follows. Section 2 contains a brief overview of related work. Section 3 describes our implementation approach. We show experimental results for an optimization application use case in Section 4, and we conclude Section 5 with a brief summary and discussion of future work.

2 Related Work

Bischof and El-Khadiri [4] describe the approach they took in implementing partial separability support in ADIFOR. Our approach, though similar in spirit, has a

number of significant differences. The ADIFOR approach assumed that the elemental functions were encoded in separate loops, while our approach does not rely on this assumption and supports partial separability when multiple element functions are computed in the same loop nest. To determine the sparsity pattern automatically when the Jacobian structure is unknown, both ADIFOR and ADIC2 use runtime detection through different versions of the SparsLinC library; in addition however ADIC2 also relies on ColPack to compute a coloring, which is used to initialize the seed matrix for computing a compressed Jacobian (or Hessian) using the forward mode of AD.

To exploit the sparsity in Jacobians of the element functions, we perform *scalar expansion*, which is the conversion of a scalar value into a temporary array. For example, scalar expansion can convert a scalar variable with a value of 1 to a vector or matrix where all the elements are 1. Typically scalar expansion is used in compiler optimizations to remove scalar data dependences across loop iterations to enable vectorization or automated parallelization. This transformation is usually limited to countable loops without control flow or function calls. The size of the temporary arrays is typically determined through polyhedral analysis of the iteration space of the loops containing the scalar operations that are candidates for expansion. Polyhedral analysis is implemented in a number of compilers and analysis tools, including Omega [16], CHiLL [7, 18], and PLuTo [3, 5]. Our current approach to the implementation of scalar expansion does not use polyhedral analysis. We describe our approach in more detail in Section 3.

3 Implementation

The changes required to support partial separability were implemented in our ADIC2 source-to-source transformation infrastructure introduced in Section 1.1. While this source translation can be performed in a standalone manner, it was convenient to implement it as a precanonicalization step in ADIC2. The ROSE compiler framework, on which ADIC2 is based, parses the input code and generates an abstract syntax tree (AST). ROSE provides APIs to traverse and modify the AST through the addition and deletion of AST nodes representing data structures and program statements.

The translation is implemented as the following three traversals of the nodes representing the function definitions within the AST.

1. Identification of partial separability (traversal T_1)
2. Promotion of elementals within loops and creation of a summation function (traversal T_2)
3. Generation of elemental initialization loop (traversal T_3)

In the first traversal, (T_1), the statements within each function definition are examined. If the pragma `$adic _partially_separable` is found, then the statement immediately following the pragma is an assignment statement whose left-hand side is

the dependent variable and the scalar-valued result of a partially-separable function computation. The right-hand side of the assignment statement is an expression involving the results of the element function computations. The names of the variables representing the elementals are specified in the pragma.

If the pragma `$adic_partially_separable` is found, the function definition is traversed again in the *TopDownBottomUp* fashion. This second traversal visits the nodes of the AST starting at the node representing the function definition and proceeding down to the leaf nodes. During this phase, which we will refer to as T_2P_1 (for second traversal, first phase), inherited attributes can be passed down to child nodes from parent nodes in the AST. Next, the nodes are visited one by one starting at the leaf nodes all the way to the function definition node. In this phase, which we will refer to as T_2P_2 (for second traversal, second phase), synthesized attributed are passed from child nodes to their parent nodes.

In phase T_2P_1 , we perform the following transformations.

1. *Scalar expansion of elementals.* In this transformation, the declaration of each of the scalar elementals is changed into a dynamically allocated array. The size of the array is determined to be the maximum number of updates to the value inside any loop nest within the function body. Each reference to the scalar variable is modified into a reference to the array elemental. The index of each array reference is determined by the bounds of the surrounding loops. For example,

```
double elemental;
    for(j = lb0; j < ub0; j++) {
        for(i = lb1; i < ub1; i++) {
            elemental = ...
        }
    }
```

is transformed to

```
double *elemental;
ADIC_SPARSE_Create1DimArray(&elemental, (ub1-lb1) * (ub0-lb0));
    for(j = lb0; j < ub0; j++) {
        for(i = lb1; i < ub1; i++) {
            elemental = ...
            temp0 = j * (ub0 - lb0) + i;
            elemental[temp0] = ...
        }
    }
```

This transformation is made possible by passing the bounds of outer loops in the T_2P_1 pass to the inner loops. In the T_2P_2 phase, if a reference is to an elemental is encountered, it is converted into an array reference, and an appropriate assignment to the array index variable is inserted at the beginning of the loop body.

In phase T_2P_2 (BottomUp pass), when an innermost loop is visited, we create a parametrized expression whose value will be the number of times that loop executes (based only on its own loop bounds). This expression is passed to its parent

as a synthesized attribute. All parent loops multiply their own local expression by the maximum of the synthesized attributes received from its children. When phase T_2P_2 is concluded, the function definition node will contain the maximum number of updates to the elementals. This value is used to allocate memory for the type-promoted elementals.

2. *Creation of the result vector*: In this transformation, the assignment statement immediately following the pragma *\$adic_partially_separable* is modified. This assignment statement is not affected by the previous transformation. A for loop is created to replace the assignment statement. The assignment statement itself is inserted into the body of the for loop. The for loop iterates as many times as the maximum number of updates to the elementals, which was determined in the previous transformation. For example,

```
#pragma $adic_partiallyseparable , elemental
* scalar = temp * (elemental);
```

is transformed to

```
#pragma $adic_partiallyseparable , elemental
for(k = 0; k < (ub1-lb1) * (ub0-lb0); k++) {
    * scalar = temp * (elemental);
}
```

The elemental variable references within the assignment undergo scalar expansion, and the LHS of the assignment statement is replaced by an array reference. The dimensions of this array are the maximum number of updates to the elementals, which was determined in the previous transformation, and the loop becomes

```
#pragma $adic_partiallyseparable , elemental
for(k = 0; k < (ub1-lb1) * (ub0-lb0); k++) {
    temp_vector[k] = temp*(elemental)[k];
}
```

3. *Summation of the result vector*: Last, a call to a summation function is added to the code. The arguments to the summation function are the scalar dependent variable and the temporary array reference that forms the left-hand side of the modified assignment statement. For example,

```
ADIC_SPARSE_Summation(scalar, temp_vector);
```

is a call that can result from this transformation.

In the third traversal (T_3), the statements within each function definition are examined again. If the pragma *\$adic_partialelemental* is found, then the statement immediately following the pragma is an assignment statement whose left-hand side is an elemental and whose right-hand side is an initialization value. Such an assignment statement is not modified by any earlier transformation. Similar to the creation of a result vector, a for loop is created that iterates as many times as the maximum

number of updates to the elementals, which was determined in the previous transformation (T_2). The assignment statement itself is inserted into the body of the for loop. Finally the loop replaces the annotated assignment statements. For example,

```
#pragma $adic_partialelemental
    elemental = 0.0;
```

is transformed to

```
#pragma $adic_partiallyseparable , elemental
for (k = 0; k < (ub1-lb1) * (ub0-lb0); k++) {
    elemental[k] = 0.0;
}
```

4 Experimental Results

We evaluated the performance of the partial separability detection implementation in ADIC2 by using a two-dimensional elastic-plastic torsion model from the MINPACK-2 test problem collection [1]. This model uses a finite-element discretization to compute the stress field on an infinitely long cylindrical bar to which a fixed angle of twist per unit length has been applied. The resulting unconstrained minimization problem can be expressed as $\min f(u)$, where $f : \mathfrak{R}^n \rightarrow \mathfrak{R}$, where $f(u)$ is the quadratic

$$f(u) = \int_{\mathfrak{D}} \left\{ \frac{1}{2} \|\nabla u(x)\|^2 - cu(x) \right\} dx \quad (2)$$

where c is a constant, and \mathfrak{D} is a bounded domain with a smooth boundary.

In our experiments, we applied ADIC2 to the C version of the function implementation after the original Fortran code was manually translated into C. As described in more detail in Section 3, we insert two types of pragmas to define (i) the elementals of the partially separable function and (ii) the initialization of the elementals.

```
...
#pragma $adic_partialelemental
    fquad = zero;
#pragma $adic_partialelemental
    flin = zero;
...
/* computation of the elementals fquad and flin of the
   function f and their sparse Jacobians */
...
#pragma $adic_partiallyseparable , fquad, flin
    *f = area*(p5*fquad+flin);
```

```
...
```

In the initialization portion, the value of the double constant *zero* is 0.0. After the T_2P_1 transformation pass (described in Section 3), the last portion of the code above (the function computation) is transformed to

```
...
#pragma $adic_partiallyseparable , fquad, flin
ADIC_SPARSE_Create1DimArray(&__adic_temp_f, ad_var_max);
for (__adic_temp0 = 0; __adic_temp0 < ad_var_max; __adic_temp0++) {
    __adic_temp_f[ __adic_temp0 ] = (area * ((p5 * fquad[ __adic_temp0 ])
                                        + flin [ __adic_temp0 ]));
}
ADIC_SPARSE_Summation(f, __adic_temp_f, ad_var_max);
...
```

where *ad_var_max* is the size of the gradient vector array for the full Jacobian.

We validated the correctness of the sparse computation by comparing it with the values produced by the analytical version. For example, for an input array size 100, the error of norm of the difference was approximately $1.5e-16$, or near the limit of machine precision for floating-point computations.

We measured the execution time of the gradient computation on an Intel Xeon workstation with dual quad-core E5462 Xeon processors (8 cores total) running at 2.8 GHz (1600 MHz FSB) with 32 KB L1 cache, 12 MB of L2 cache (6 MB shared per core pair), and 16 GB of DDR2 FBDIMM RAM, running Linux kernel version 2.6.35 (x86-64). All codes were compiled with gcc version 4.4.5 with `-O2` optimizations enabled.

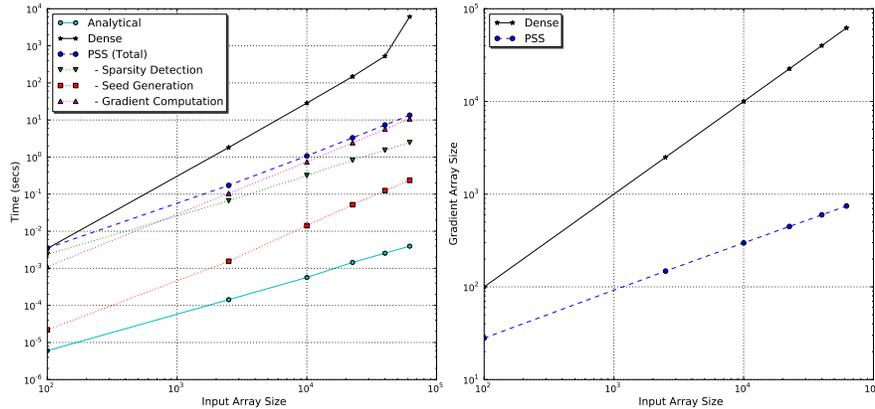


Fig. 2 Left: comparison between the runtimes of the three gradient versions: hand-coded analytical gradients, dense AD, and partially separable sparse AD (PSS). A breakdown of the steps in the PSS version is also shown: sparsity detection, seed generation, and gradient computation and Jacobian recovery. Right: gradient array sizes for the dense and PSS versions.

Figure 2 (left) shows the execution times for computing the Jacobian of the function in Eq. 2 using three approaches: (1) manually implemented analytical derivative computation; (2) dense, forward mode AD (using ADIC2); and (3) sparse, forward mode AD (using ADIC2) with additional sparsity detection (using SparsLinC) and Jacobian compression (using ColPack). The analytic version performs best, as expected. The forward-mode dense AD version is between 500 and 3,400 times slower than the manually optimized analytical derivatives computation, while the partially separable sparse AD version achieves performance within a factor of 6 of the analytic version for small array sizes, and less than a factor of 2 slower for larger array sizes. The right side of Fig. 2 shows the reduction in memory requirements (ranging from 500- to 4000-fold) for storing the gradients using the PSS approach compared with dense gradients.

5 Conclusion

We presented an approach to exploiting sparsity in the computation of gradients of partially separable functions, which are common in large-scale optimization. We identify partially separable computations by using pragmas, which guide our source transformation system to perform scalar expansion and generate efficient forward mode AD code for computing the gradients of the element functions. In addition, we exploit sparsity in these gradients by using the SparsLinC library and the ColPack coloring toolkit to enable efficient forward mode AD by using statically allocated compressed dense vectors for computing the gradients of intermediate active variables. We evaluated the performance of our implementation using a case study of the elastic-plastic torsion problem in the MINPACK-2 test suite, demonstrating that (1) exploiting partial separability and sparsity significantly reduces the memory requirements of the generated code, enabling the solution of larger problems than possible with dense forward mode, and (2) the performance of the best AD version compares favorably with that of the hand-coded gradients. In future work we will extend ADIC2 to remove certain restrictions, for example, the assumption that the gradient vectors of different element functions are of the same size. We also plan to integrate the polyhedral analysis currently being developed in ROSE and add support for exploiting partial separability when using the reverse mode in ADIC2.

Acknowledgments This work was supported by the U.S. Dept. of Energy Office of Science Applied Mathematics Program under Contract No. DE-AC02-06CH11357.

References

1. Averick, B.M., Carter, R.G., Moré, J.J., Xue, G.L.: The MINPACK-2 test problem collection. Tech. Rep. Preprint MCS-P152-0694, Mathematics and Computer Science Division, Argonne

- National Laboratory (1992)
2. Averick, B.M., Moré, J.J., Bischof, C.H., Carle, A., Griewank, A.: Computing large sparse Jacobian matrices using automatic differentiation. *SIAM J. Sci. Comput.* **15**(2), 285–294 (1994). DOI 10.1137/0915020. URL <http://link.aip.org/link/?SCE/15/285/1>
 3. Baskaran, M.M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In: Proc. 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, PPOPP '08, pp. 1–10. ACM, New York (2008). DOI <http://doi.acm.org/10.1145/1345206.1345210>. URL <http://doi.acm.org/10.1145/1345206.1345210>
 4. Bischof, C., El-Khadiri, M.: On exploiting partial separability and extending the compile-time reverse mode in ADIFOR. Technical Memorandum ANL/MCS–TM–163, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill. (1992). ADIFOR Working Note # 7.
 5. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: PLuTo: A practical and fully automatic polyhedral program optimization system. In: Proc. ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI) (2008)
 6. Bouaricha, A., Moré, J.J.: Impact of partial separability on large-scale optimization. *Comp. Optim. Appl.* **7**(1), 27–40 (1997)
 7. Chen, C.: Model-guided empirical optimization for memory hierarchy. Ph.D. thesis (2007)
 8. Coleman, T.F., Moré, J.J.: Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM Journal on Numerical Analysis* **20**(1), 187–209 (1983)
 9. Curtis, A.R., Powell, M.J.D., Reid, J.K.: On the estimation of sparse Jacobian matrices. *Journal of the Institute of Mathematics and Applications* **13**, 117–119 (1974)
 10. Edison Design Group C++ Front End. http://www.edg.com/index.php?location=c_frontend
 11. Gebremedhin, A., A.Tarafdar, Manne, F., Pothén, A.: New acyclic and star coloring algorithms with applications to Hessian computation. *SIAM Journal on Scientific Computing* **29**(3), 1042–1072 (2007)
 12. Gebremedhin, A.H., Manne, F., Pothén, A.: What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Review* **47**(4), 629–705 (2005). DOI 10.1137/S0036144504444711. URL <http://link.aip.org/link/?SIR/47/629/1>
 13. Gebremedhin, A.H., Nguyen, D., Patwary, M., Pothén, A.: ColPack: Software for graph coloring and related problems in scientific computing. Tech. rep., Purdue University (2011)
 14. Griewank, A., Toint, P.L.: On the unconstrained optimization of partially separable functions. *Nonlinear Optimization* pp. 119–137 (1981)
 15. Narayanan, S.H.K., Norris, B., Winnicka, B.: ADIC2: Development of a component source transformation system for differentiating C and C++. *Procedia Computer Science* **1**(1), 1845–1853 (2010). DOI DOI:10.1016/j.procs.2010.04.206. URL <http://www.sciencedirect.com/science/article/pii/S1877050910002073>. ICCS 2010
 16. Pugh, W.: The Omega Project web page. <http://www.cs.umd.edu/projects/omega/>
 17. Quinlan, D.: ROSE: Compiler support for object-oriented frameworks. Tech. Rep. UCRL-ID-136515, Lawrence Livermore National Laboratory (1999). URL <http://www.osti.gov/bridge/servlets/purl/793936-hdq2WX/native/793936.PDF>
 18. Tiwari, A., Chen, C., Jacqueline, C., Hall, M., Hollingsworth, J.K.: A scalable auto-tuning framework for compiler optimization. In: Proc. 2009 IEEE International Symposium on Parallel & Distributed Processing, pp. 1–12. Washington, DC (2009)

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.