Hierarchical Approaches to Automatic Differentiation

Christian Bischof
Mohammad Haghighat

CRPC-TR96647
April 1996

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Hierarchical Approaches to Automatic Differentiation*

Christian H. Bischof† Mohammad R. Haghighat‡

Abstract

A mathematical function, specified by a computer program, can be differentiated efficiently through the exploitation of its program structure. The important properties of a program for the efficient generation of derivative code are the asymmetries between the number of inputs and outputs of program components at various levels of abstraction and the mathematical complexity of the involved operators. Automatic generation of efficient derivative codes thus requires analysis of programs for detection of such properties and systematic methods for their exploitation in composing the derivative codes. We suggest a hierarchical approach based on a partitioning of the computational or program graph as a means to deduce workable solutions to this hard problem. Each partition corresponds to a localized scope for derivative computation, and hierarchical partitions provide a mechanism for exploiting program structure at various levels. As a particular example, we discuss dynamic programming approaches for finding good one-dimensional partitions and generalizations to arbitrary directed acyclic graphs that, by recycling substructure information, allow one to determine the optimal elimination ordering for a graph with n nodes with complexity $O(2^n)$, as compared with the $O(n!)$ complexity of a naive search. Lastly, we give a concrete example illustrating the hierarchical approach on the driven cavity problem from the MINPACK-2 optimization test set collection.

Keywords: differentiation partition, hybrid modes, optimal elimination ordering, higher-level operands, computational graph, program graph, hierarchical differentiation.

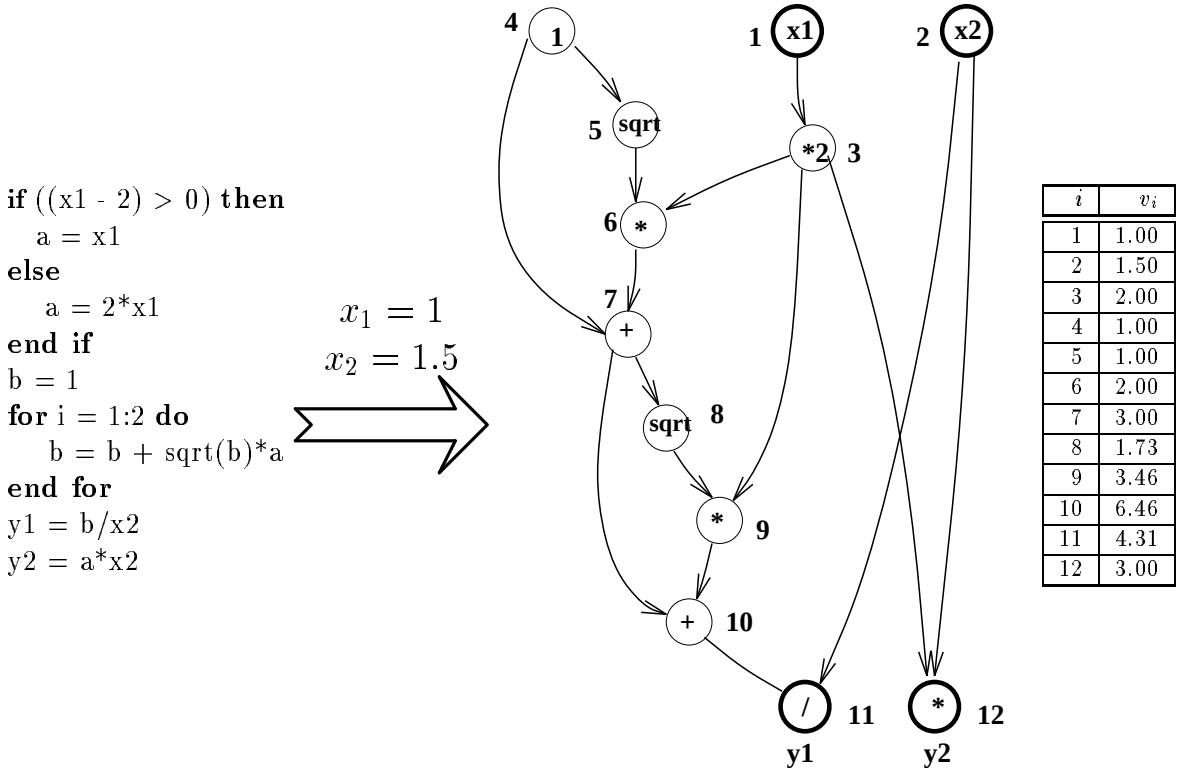
1 Introduction

Traditionally, automatic differentiation of computer programs has been strongly influenced by the view of the program as a computational graph or Kantorovich graph (see, for example, [11, 14, 16, 20]). An example of a computer program and its corresponding computational graph is shown in Figure 1. Note that the computational graph represents the program only when the if-branch evaluates to **false**, as is the case if we instantiate x_1 and x_2 to 1 and 1.5, respectively. Further, note that each node corresponds to a unique *value* v_i, rather than a *storage location* and that loops were unrolled. Given the graph, we can then evaluate the program in a data-flow fashion. The values thus computed are shown

*This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, by the National Aerospace Agency under Purchase Order L25935D and by the National Science Foundation, through the Center for Research on Parallel Computation, under Cooperative Agreement No. CCR-9120008.

†Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439, bischof@mcs.anl.gov.

‡Microcomputer Research Labs, Intel Corporation, RN6-18, 2200 Mission College Blvd., Santa Clara, CA 95052, mhaghigh@gomez.sc.intel.com. This work was performed while the author was a postdoctoral associate at the Mathematics and Computer Science Division of Argonne National Laboratory.



i	v_i
1	1.00
2	1.50
3	2.00
4	1.00
5	1.00
6	2.00
7	3.00
8	1.73
9	3.46
10	6.46
11	4.31
12	3.00

FIG. 1. *A computer program and its representation as a computational graph*

in the table on the right side of Figure 1, with v_i denoting the value computed at node i. Since the floating-point operations supported by traditional programming languages such as Fortran 77, C, and Pascal are at the scalar level, the elementary operations represented in such computational graphs are typically scalar, with each node having at most two input arcs.

From this graph, we can derive the linearized graph, where we label the arcs corresponding to an elementary operation Φ having inputs u and v with the elementary derivatives $\partial\Phi/\partial u$ and $\partial\Phi/\partial v$. The linearized graph induced by the computational graph in Figure 1 is shown on the left side of Figure 2. Since all operations have been linearized, the operands associated with the nodes can be omitted.

As described by Griewank and Reese [16], the final Jacobian can then be computed by a graph elimination approach. The elimination rule (shown on the right side of Figure 2) involves making the neighbors of an eliminated node a clique, instantiating or augmenting the weights on the clique arcs by the product of the path through the eliminated node. The associativity of the chain rule of differential calculus implies that nodes can be eliminated in arbitrary order, and the final Jacobian can be represented by the bipartite graph containing only the input and output variables of the program.

The forward mode of automatic differentiation generates the linearized graph and eliminates nodes in a fashion that is consistent with the order in which values are computed during program execution. Hence, there is no need to actually build the computational graph. In contrast, the reverse mode of automatic differentiation eliminates the nodes by starting at the output variables of the program, thus requiring storage of the computational or linearized graph in some form. For example, ADOL-C [15] generates a "tape," encoding the operands and operations in the order in which they were encountered during a program

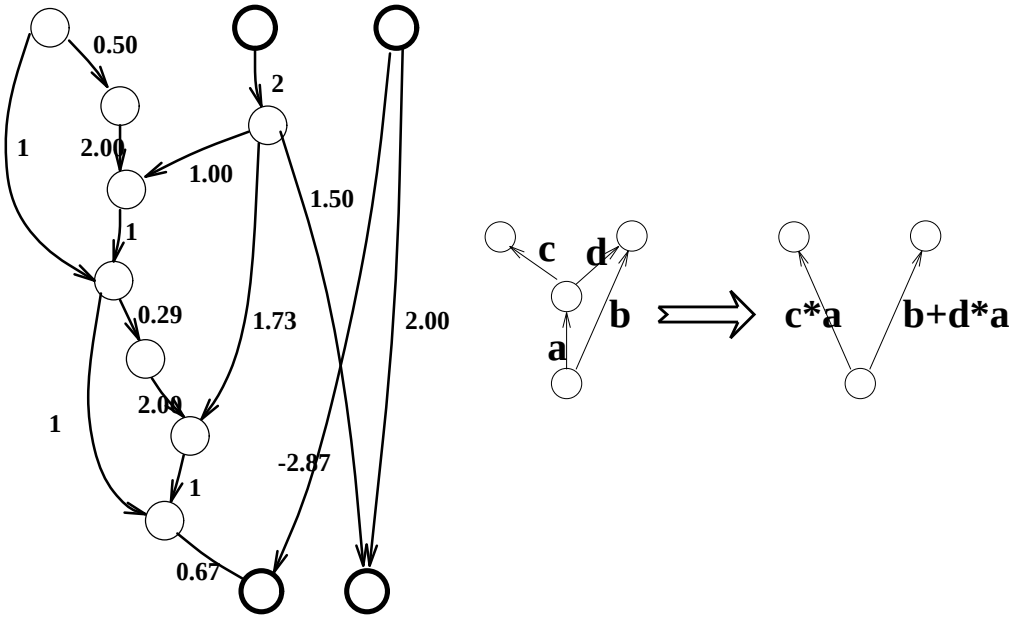


FIG. 2. *Linearized graph induced by computation shown in Figure 1 (left) and example of node elimination rule for linearized graph (right)*

execution; postprocessing utilities can then compute various varieties of derivatives off-line. In contrast, JAKEF [18] stores the linearized graph directly. Note that each of these representations captures only a particular execution path through the program. Once such a graph representation has been built, however, other elimination orderings are possible, such as the Markowitz rule described in [16].

The drawback of this fine-grained graph view of automatic differentiation is that it does not scale. Deriving computational graphs where nodes correspond to scalar additions and multiplications is infeasible for large programs, since such a representation must occupy storage on the order of the number of floating-point operations executed in a program.

One way to circumvent this difficulty is to increase the granularity of the elementary operators in the underlying language. For example, the CODE system [21], as well as the class definitions underlying the weather model described in [24], employs 3-D data structures as elementary objects. Vectors are employed as base types in [26], and the elementary objects that can be symbolically differentiated by an algebraic manipulation system are the elementary objects in [22]. The reduction in complexity of the computational graph can be dramatic, as illustrated in Figure 3, which expresses computation at the matrix and vector levels (the matrix M is considered constant with respect to differentiation). Here, up to $O(n^2)$ elementary operations are represented by one node. Note that the elimination rule (Figure 2) still holds, if the multiplication is interpreted as matrix multiply. For example, if we eliminate the node computing α, we obtain $\partial b/\partial x = 2yx^T \in \mathbf{R}^{n\times n}$. If we eliminate the node computing b first, we obtain $\partial \gamma/\partial \alpha = 2(Mb)^T y \in \mathbf{R}$ and $\partial \gamma/\partial y = 2(Mb)^T A \in \mathbf{R}^{1\times n}$. We note that an approach at the matrix level was also used in [13] to optimize derivatives in the context of a symbolic manipulation system.

If matrix and vector operations are represented as subroutine calls, the structure shown in Figure 3 would correspond to the highest level of abstraction of the program graph (or abstract syntax tree), which is the data structure that compilers typically use to represent

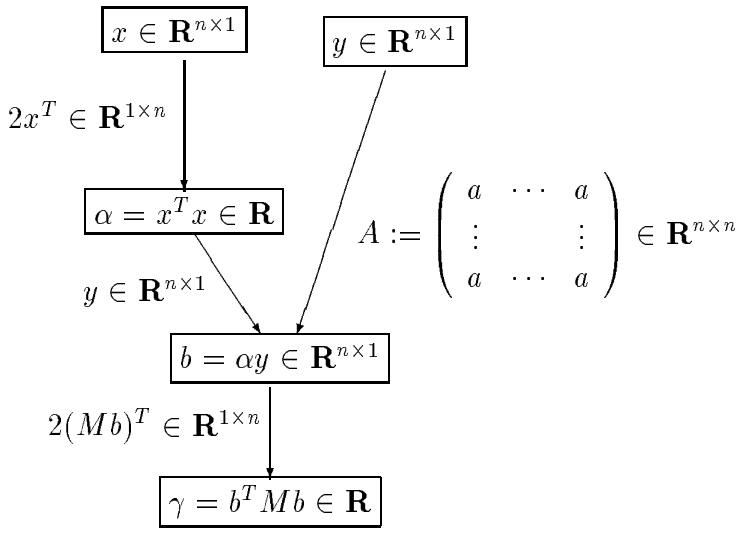


FIG. 3. *Computational graph employing matrix and vector operations and corresponding "elementary" partial derivatives*

and manipulate programs. Recently, automatic differentiation has been approached as a source transformation problem in the ADIFOR [4, 5], ADIC [8], AMC [12], and Odyssée [25] tools. Instead of interpreting the computational graph on the fly or constructing it as a byproduct of program execution, these tools generate a new code that, when executed, computes the desired derivatives. As a result, the adjoint codes generated by Odyssée or AMC can compute derivatives for *all* potential paths through the program, not just a particular path.

2 Hierarchical Derivative Approaches, Graph Partitioning, and Interface Contraction

Opportunities for decreasing the complexity of derivative computation arise when the number of variables passing through a vertex separator (i.e., a group of vertices whose removal from the program graph leads to two disjoint subgraphs) to the computational or program graph is smaller than the number of independent or dependent variables to be computed. Consider Figure 4. If either the forward or the reverse mode is used, node elimination order requires 22 multiplications. On the other hand, if we first eliminate the nodes above node s to compute $\partial s/\partial(a,b,c)$, then eliminate the nodes below node s to compute $\partial(x,y,z)/\partial s$, and finally eliminate s to arrive at $\partial(a,b,c)/\partial(x,y,z)$, only 17 multiplications are required. Iri [20] referred to such a situation as a "vertex cut"; Hovland et al. [19] called it an "interface contraction."

This small example illustrates three important principles.

Graph Partitioning: The identification of program sections whose derivatives with respect to input and output variables should be computed "out of context," that is, ignoring the surrounding computations.

Hierarchical Chain Rule Application: The chain rule is applied at various levels of abstraction.

Interface Contraction: Whenever we can identify a piece of the program whose number of input arguments is smaller than the number of independent variables for differen-

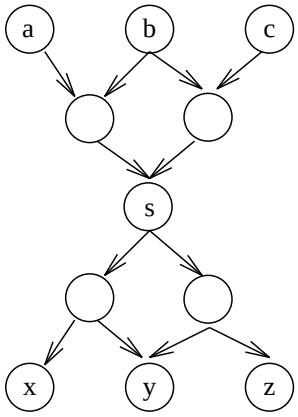


FIG. 4. *Example demonstrating the global Inadequacy of either forward or reverse mode*

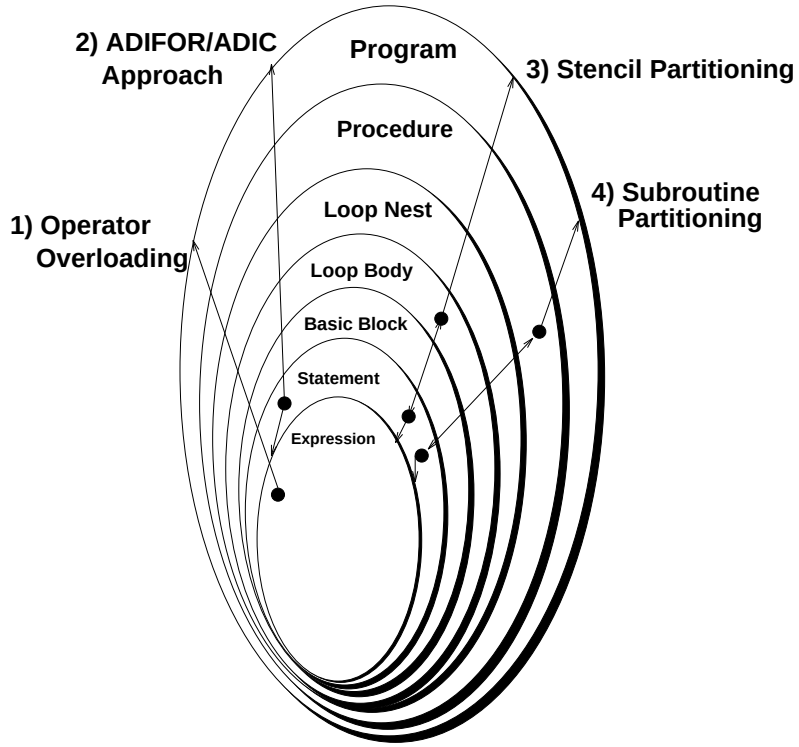


FIG. 5. *Program structure and hierarchical differentiation approaches*

tiation, the independent differentiation of these program pieces is likely to decrease the derivative complexity in a global forward-mode approach. The same holds for the output variables in a reverse-mode framework.

A hierarchical view of a program as well as a visualization of various approaches for computing derivatives is shown in Figure 5. We make the following observations:

1. The structure of Figure 5 is, for the most part, invisible to programming systems that perform automatic differentiation by redefining the elementary operators. Such systems can be viewed as propagating at every step the derivatives of the value of an expression with respect to the global independent variables (in the forward mode) or the derivatives of the global dependent variables with respect to the value returned

by an expression (in the reverse mode).

2. Automatic differentiation (AD) systems employing compiler infrastructure can go beyond this myopic view. This ability to redefine the scope of differentiation is the biggest advantage of the source transformation approach to AD. For example, the approach currently employed in ADIFOR and ADIC can be interpreted as a uniform statement-level partitioning of the program graph. That is, for each assignment statement, the derivative of the result with respect to the variables appearing on the right-hand side is computed by using the reverse mode, and then these derivatives are chained together by using the forward mode, ignoring the remainder of the program structure. Note that, while the reverse mode is the better design choice for computing the little statement gradients, the use of the forward mode would in all likelyhood also have improved performance as long as the number of variables on the right-hand side is smaller than the number of independent variables to be computed globally.

3. Moving up in the program structure, we note that many solution approaches for partial differential equations based on finite-difference approaches employ so-called stencils to update the value at a particular grid point by computing a function of a limited number of neighbors. For example, the seven-point stencil computes the new value of $x(i,j,k)$ from $x(i,j,k), x(i-1,j,k), x(i+1,j,k), x(i,j-1,k), x(i,j+1,k), x(i,j,k-1)$, and $x(i,j,k+1)$. Such a stencil typically is the body of a multidimensional loop, and it may well be profitable to declare it as its own differentiation partition, perhaps by using statement-level partitioning in the ADIFOR/ADIC fashion at a lower level. A concrete example of stencil-level partitioning is given in the next section.

4. Moving up further in the program structure, we observe that subroutine boundaries may be likely places for interface contraction. For example, in the application described in [19], a subroutine that computed a scalar output value from two scalar input values was called in a loop. By considering this subroutine as its own derivative partition, computing its associated two-element gradient using ADIFOR, and then chaining this resulting derivative back into the remainder of the program, the authors obtained a derivative code whose performance was close to a hand-coded one. This approach is the default used for Jacobian generation in Odyssée [10].

To reiterate, these examples achieved improvements by defining a portion of the program (a "differentiation partition"), computing the derivatives of this program fragment in a fashion that was oblivious of context, and applying the chain rule at this level of granularity. In the resulting hierarchical program graph, this partitioning approach can be repeated at many levels of abstraction. To find suitable partitions, we need to know the following information.

Information bandwidth in and out of a candidate partition: The amount of information passing in and out of a program fragment determines the cost of integrating the derivatives determined by this program fragment in a forward- or backward-oriented approach. Data flow and dependence analysis techniques, developed in the context of parallelizing compilers (see, for example, [17, 23]), provide a strong foundation from which such information can be derived.

Estimate of complexity of computing the derivatives of a candidate partition: The cost of derivatives of a program fragment depends on many factors, such as

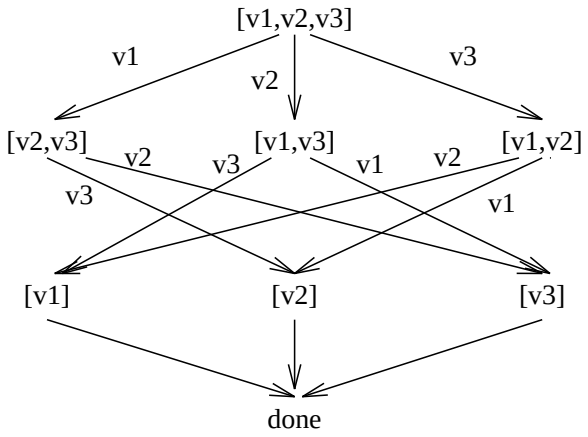


FIG. 6. *Illustration of information reuse in node elimination*

- the memory and floating-point resources,
- the degree of nonlinearity in the computation, and
- the degree of derivative sparsity (see, for example, [7, 3]).

Data flow and dependence analysis tools, static performance analyzers, runtime tools for gathering program statistics, and user interaction may all contribute to this phase.

While we could conceivably partition a computer program in an arbitrary fashion, we believe that the program structure shown in Figure 5 provides useful guidance for choosing candidate partitions and greatly restricts the potential partitions to be considered.

3 Generalizations of Dynamic Programming for Determining Accumulation Orderings

The discussion in the preceding section leads, at every level of partitioning, to the following subproblem: Given a directed acyclic graph (DAG) G with n nodes v_i, say, where the derivatives of each node with respect to any of its predecessors are known, derive the optimal ordering for accumulating the derivatives defined by this DAG. The total number of different elimination orderings is $n!$ the enumeration of which is clearly impractical unless n is small. However, as illustrated in Figure 3, a hierarchical view of the program may well lead to small graphs.

For linear DAGs, dynamic programming [1, pp. 67–69] arrives at an $O(n^3)$ algorithm for finding the minimum-cost node elimination ordering by realizing that the optimal solution is composed of optimal solutions to subproblems. That is, if we define $C(k:j)$ to be the minimum cost of computing the derivatives associated with the DAG consisting of v_k to v_j, and b_i to be the number of values passed from v_i to v_{i+1} (b_0 is the number of inputs to v_1), we have

$$C(k:j) = \min_{k<l<j}(C(k:l-1) + C(l+1:j) + 2b_k b_l b_j).$$

The $O(n^3)$ complexity is achieved by tabulating the solutions in the order of increasing $k-j$.

This approach can be applied to the general case. To illustrate, consider a graph with three nodes, and Figure 6. We can eliminate an arbitrary node first, leading to the three subproblems shown at the next level. Eliminating the next node leads to three one-node subproblems, each of which appears twice. If we tabulate in this fashion for n nodes,

```
      do i = 1, ny
        do j = 1, nx
           k = (i-1)*nx + j

           pbl = x(k-nx-1)
           pb  = x(k-nx)
           pbb = x(k-2*nx)
           pbr = x(k-nx+1)
           pl  = x(k-1)
           pll = x(k-2)
           p   = x(k)
           pr  = x(k+1)
           prr = x(k+2)
           ptl = x(k+nx-1)
           pt  = x(k+nx)
           ptt = x(k+2*nx)
           ptr = x(k+nx+1)

           dpdy  = (pt-pb)/(two*hy)
           dpdx  = (pr-pl)/(two*hx)
           pblap = (pbr-two*pb+pbl)/hx2 + (p-two*pb+pbb)/hy2
           pllap = (p-two*pl+pll)/hx2 + (ptl-two*pl+pbl)/hy2
           plap  = (pr-two*p+pl)/hx2 + (pt-two*p+pb)/hy2
           prlap = (prr-two*pr+p)/hx2 + (ptr-two*pr+pbr)/hy2
           ptlap = (ptr-two*pt+ptl)/hx2 + (ptt-two*pt+p)/hy2
           fvec(k) = (prlap-two*plap+pllap)/hx2 +
     +               (ptlap-two*plap+pblap)/hy2 -
     +               r*(dpdy*(prlap-pllap)/(two*hx)-
     +               dpdx*(ptlap-pblap)/(two*hy))

        enddo
      enddo
```

FIG. 7. *Main loop inside of MINPACK-2's driven cavity problem*

the number of nodes of the enumeration graph in Figure 6 will adhere to the binomial sequence, resulting in 2^n nodes and $n2^{n-1}$ edges overall. Thus, we can arrive at the optimal solution strategy far more rapidly than with the naive approach. For example, for $n = 10$, $n! = 3,628,800$, whereas $2^n = 1,024$.

Although a considerable improvement over the naive search, this exponential enumeration may still require significant effort, especially for large n. Thus, we expect that heuristics will continue to play an important role. For example, we can reduce the general DAG case to the linear DAG case by ordering DAG nodes in a topological fashion and considering all nodes at a given level as a "supernode." Since nodes in a supernode are independent, they can actually be eliminated in parallel, an approach that was explored at the elementary operation level in [9].

4 A Case Study

We now provide a concrete example illustrating that hierarchical approaches to derivative generation can do considerably better than either of the "monolithic" approaches largely underlying current AD tools. To this end, we employ the driven cavity problem in the MINPACK-2 test set collection [2]. Its main computational kernel (ignoring boundary conditions) is the loop shown in Figure 7, which computes an `n = nx * ny` vector `fvec`

from an `n`-vector `x`.

Let us assume that we are interested in computing $D := d\,\texttt{fvec}/d\,\texttt{x} * S$, where S is an $n \times p$ matrix with $p \leq n$ matrix. If we apply a mainly forward-mode tool such as ADIFOR 2.0 [5] to compute D, the derivative cost will be linear in p. Applying a reverse-mode tool such as Odyssée [25] overall does not make sense, since its complexity would be linear in $n \geq p$.

To do better, we realize that the loop body can be viewed as a mapping of 13 distinct elements of the array `x` to `fvec(k)`. Thus, as long as $p < 13$, we have an instance of interface contraction. We can isolate the stencil update (the lines of code from the definition of `dpdy` through the update of `fvec(k)`) in a subroutine `loopbody`, say, and apply an automatic differentiation tool to generate a code to compute the derivatives $g := d\,\texttt{fvec(k)}/d\,\texttt{p}*$, where `p`$*$ is shorthand for "all variables starting with p."

If we use ADIFOR to generate code for the computation of g, we obtain the derivative code shown in Figure 8. To fit the code on one page, we have taken some liberty with alignment of Fortran continuation and comment characters. Since neither of the values of the array `x` is overwritten inside the loop, we moved the initialization of the seed matrices for the computation of the 13-element gradient outside of the loop. The variable `p` corresponds to the number of columns of the global seed matrix S. The number 13 in the call to `g_loopbody` is the number of derivatives we wish to compute in this call. For more details on the use of ADIFOR-generated code, see [6].

On the other hand, the code inside `loopbody` is a perfect candidate for the use of the reverse mode. We are interested only in the derivatives with respect to one dependent variable, and no loops or branches complicate the generation of the reverse mode. If we employ Odyssée to generate code for g, we obtain the code shown in Figure 9. Note that while `g_loopbody` and `loopbodyad` require different initializations and use different approaches to computing the gradient g, the chain rule "glue code" remains unchanged.

Executing the code generated by a global application of ADIFOR 2.0, as well as the codes shown in Figure 8 and 9, on a Sun SPARCstation iPX, with $nx = ny = 120$ and p ranging from 5 to 60, we obtain the results shown in Figure 10.

The behavior of the global ADIFOR and the ADIFOR version with interface contraction is as expected, with a crossover around $p = 15$, close to the number of variables (13) involved in the `loopbody` interface. For $p < 13$, our approach actually leads to an interface expansion and performs worse than the global ADIFOR approach.

The performance of the version that uses the Odyssée-generated reverse mode to compute the small gradient g is even better. Because of the simple structure of the underlying code, the reverse mode can be implemented without any overhead for storing (or recomputing) values that are overwritten, or directions of branches taken. Thus, the resulting code outperforms the ADIFOR approach for computing the small gradient g by a considerable margin. Somewhat surprising, the "interface contraction with Odyssée approach" outperforms the global ADIFOR approach even for $p = 5$, illustrating the potential power of hierarchical approaches coupled with context-sensitive differentiation strategies.

5 Conclusions

This paper explored hierarchical approaches to automatic differentiation. We showed how interface asymmetries between the number of derivatives to be computed and the amount of information flowing in or out of a program segment can be used profitably to decrease

```
*  Initialization of ''subroutine seed matrices''

*  NOT SHOWN HERE TO SAVE SPACE: The initialization of
*  lg_pbl(1:13), lg_pb(1:13), lg_pbb(1:13), lg_pbr(1:13), lg_pl(1:13),
*  lg_pll(1:13), lg_p(1:13), lg_pr(1:13), lg_prr(1:13), lg_ptl(1:13),
*  lg_pt(1:13), lg_ptt(1:13), and lg_ptr(1:13) to zero

 lg_pbl(1)  = 1.0d0
 lg_pb(2)   = 1.0d0
 lg_pbb(3)  = 1.0d0
 lg_pbr(4)  = 1.0d0
 lg_pl(5)   = 1.0d0
 lg_pll(6)  = 1.0d0
 lg_p(7)    = 1.0d0
 lg_pr(8)   = 1.0d0
 lg_prr(9)  = 1.0d0
 lg_ptl(10) = 1.0d0
 lg_pt(11)  = 1.0d0
 lg_ptt(12) = 1.0d0
 lg_ptr(13) = 1.0d0

 do 99998 i = 3, ny - 2
     do 99999 j = 3, nx - 2
       k = (i - 1) * nx + j

* Computation of gradient defined by loop body

       call g_loopbody(13,
+           x(k - nx - 1), lg_pbl, x(k - nx),     lg_pb,
+           x(k - 2 * nx), lg_pbb, x(k - nx + 1), lg_pbr,
+           x(k - 1),      lg_pl,  x(k - 2),      lg_pll,
+           x(k),          lg_p,   x(k + 1),      lg_pr,
+           x(k + 2),      lg_prr, x(k + nx - 1), lg_ptl,
+           x(k + nx),     lg_pt,  x(k + 2 * nx), lg_ptt,
+           x(k + nx + 1), lg_ptr, fvec(k),       lg_fval,
+           hx, hy, r)

*   Derivative chain rule to update derivatives of fvec(k) using
*   gradient of loop body and derivatives of variables entering
*   loop body.

       do ii = 1, p
       g_fvec(ii,k) =
+        lg_fval(1) * g_x(ii,k-nx-1)  + lg_fval(2)  * g_x(ii, k-nx)   +
+        lg_fval(3) * g_x(ii, k-2*nx) + lg_fval(4)  * g_x(ii, k-nx+1) +
+        lg_fval(5) * g_x(ii, k - 1)  + lg_fval(6)  * g_x(ii, k-2)    +
+        lg_fval(7) * g_x(ii,k)       + lg_fval(8)  * g_x(ii, k+1)    +
+        lg_fval(9)  * g_x(ii, k + 2) + lg_fval(10) * g_x(ii, k+nx-1) +
+        lg_fval(11) * g_x(ii, k + nx)+ lg_fval(12) * g_x(ii, k+2*nx) +
+        lg_fval(13) * g_x(ii, k + nx + 1)
       enddo

99999     continue
99998   continue
```

FIG. 8. *Sample derivative code employing ADIFOR-generated code to compute the gradient associated with the loop body*

```
      do 99998 i = 3, ny - 2
        do 99999 j = 3, nx - 2
          k = (i - 1) * nx + j

          fvalad = 1.0d0
          do ii = 1, 13
             lg_fval(ii) = 0.0d0
          enddo
          call loopbodyad(
     +          x(k - nx - 1), x(k - nx), x(k - 2 * nx),
     +          x(k - nx + 1), x(k - 1),  x(k - 2),
     +          x(k), x(k + 1), x(k + 2), x(k + nx - 1),
     +          x(k + nx), x(k + 2 * nx), x(k + nx + 1),
     +          fvec(k), hx, hy, r, lg_fval(1), lg_fval(2),
     +          lg_fval(3), lg_fval(4), lg_fval(5), lg_fval(6),
     +          lg_fval(7), lg_fval(8), lg_fval(9), lg_fval(10),
     +          lg_fval(11), lg_fval(12), lg_fval(13), fvalad)

     *   Derivative chain rule to update derivatives of fvec(k) using
     *   gradient of loop body and derivatives of variables entering
     *   loop body.

          do ii = 1, p
          g_fvec(ii,k) =
     +      lg_fval(1) * g_x(ii,k-nx-1)  + lg_fval(2)  * g_x(ii, k-nx)   +
     +      lg_fval(3) * g_x(ii, k-2*nx) + lg_fval(4)  * g_x(ii, k-nx+1) +
     +      lg_fval(5) * g_x(ii, k - 1)  + lg_fval(6)  * g_x(ii, k-2)    +
     +      lg_fval(7) * g_x(ii,k)       + lg_fval(8)  * g_x(ii, k+1)    +
     +      lg_fval(9)  * g_x(ii, k + 2) + lg_fval(10) * g_x(ii, k+nx-1) +
     +      lg_fval(11) * g_x(ii, k + nx)+ lg_fval(12) * g_x(ii, k+2*nx) +
     +      lg_fval(13) * g_x(ii, k + nx + 1)
          enddo

     99999     continue
     99998   continue
```

FIG. 9. *Sample derivative code employing Odyssée-generated code to compute the gradient associated with the loop body*

derivative complexity. The resulting approach can be viewed as a hierarchical partitioning of the program graph, with individual partitions corresponding to program segments for which derivatives are computed ignoring their computational context and then spliced into the surrounding computation by an application of the chain rule at a higher level of granularity. We suggested approaches for choosing such partitions in computational graphs, and we gave a concrete example illustrating a partitioning at the loop-body level, using different approaches for generating the loop-body derivatives.

Work is under way to provide a largely language-independent intermediate representation of program fragments that would facilitate easy experimentation with different partition strategies and context-sensitivity differentiation approaches, thus allowing the exploration of the combinatorial richness created by the associativity of the derivative chain rule. We believe that, because of the simplicity of the current approaches, the resulting algorithmic improvements will result in order of magnitude saving even for moderate differentiation problems.

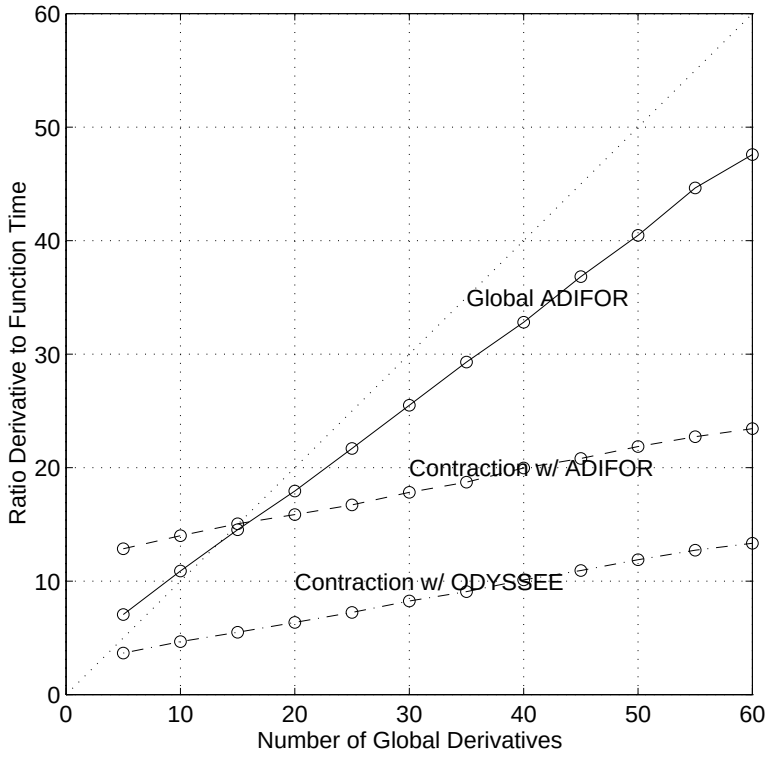


FIG. 10. *Result of Application of Interface Contraction to Driven Cavity Problem*

Acknowledgments

We thank Alan Carle and Lucas Roh for illuminating discussions on the subject, and Nicole Rostaing-Schmidt for processing the example code with Odyssée.

References

[1] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[2] B. M. Averick, R. G. Carter, J. J. Moré, and G. L. Xue, *The MINPACK-2 test problem collection*, Tech. Rep. ANL/MCS-TM-150, Rev. 1, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[3] C. Bischof, A. Bouaricha, P. Khademi, and J. Moré, *Computing gradients in large-scale optimization using automatic differentiation*, Preprint MCS-P488-0195, Mathematics and Computer Science Division, Argonne National Laboratory, 1995.

[4] C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland, *ADIFOR: Generating derivative codes from Fortran programs*, Scientific Programming, 1 (1992), pp. 11–29.

[5] C. Bischof, A. Carle, P. Khademi, and A. Mauer, *The ADIFOR 2.0 system for the automatic differentiation of Fortran 77 programs*, 1994. Preprint MCS-P481-1194, Mathematics and Computer Science Division, Argonne National Laboratory, and CRPC-TR94491, Center for Research on Parallel Computation, Rice University. To appear in IEEE Computational Science & Engineering.

[6] C. Bischof, A. Carle, P. Khademi, A. Mauer, and P. Hovland, *ADIFOR 2.0 user's guide*, Technical Memorandum ANL/MCS-TM-192, Mathematics and Computer Science Division, Argonne National Laboratory, 1994. CRPC Technical Report CRPC-95516-S.

[7] C. Bischof, P. Khademi, A. Bouaricha, and A. Carle, *Computation of gradients and Jacobians by transparent exploitation of sparsity in automatic differentiation*, 1995. Preprint MCS-P519-0595, Mathematics and Computer Science Division, Argonne National Laboratory, and

CRPC-TR95583, Center for Research on Parallel Computation, Rice University. Accepted for publication in Optimization Methods and Software.

[8] C. Bischof, L. Roh, and A. Mauer, unpublished information, Argonne National Laboratory, 1996.

[9] C. H. Bischof, *Issues in parallel automatic differentiation*, in Automatic Differentiation of Algorithms, A. Griewank and G. Corliss, eds., Philadelphia, PA, 1991, SIAM, pp. 100–113.

[10] B. Mohammadi, J.-M. Malé, and N. Rostaing-Schmidt, *Automatic Differentiation in Direct and Reverse Modes: Application to Optimum Shape Design in Fluid Mechanics*, in Computational Differentiation: Techniques, Applications, and Tools, M. Berz, C. Bischof, G. Corliss, and A. Griewank, eds., SIAM, Philadelphia, Penn., 1996. To appear.

[11] L. C. W. Dixon, *Use of automatic differentiation for calculating Hessians and Newton steps*, in Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. F. Corliss, eds., SIAM, Philadelphia, Penn., 1991, pp. 114 – 125.

[12] R. Giering, *Adjoint model compiler, manual version 0.2, AMC version 2.04*, tech. rep., Max-Planck Institut für Meteorologie, August 1992.

[13] V. V. Goldman, J. Molenkamp, and J. A. van Hulzen, *Efficient numerical program generation and computer algebra environments*, in Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. F. Corliss, eds., SIAM, Philadelphia, 1991, pp. 74–83.

[14] A. Griewank, *On automatic differentiation*, in Mathematical Programming: Recent Developments and Applications, Amsterdam, 1989, Kluwer Academic Publishers, pp. 83–108.

[15] A. Griewank, D. Juedes, and J. Srinivas. *ADOL-C, a package for the automatic differentiation of algorithms written in C/C++*, Preprint MCS-P180-1190, Mathematics and Computer Science Division, Argonne National Laboratory, 1990.

[16] A. Griewank and S. Reese, *On the calculation of Jacobian matrices by the Markowitz rule*, in Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. F. Corliss, eds., SIAM, Philadelphia, 1991, pp. 126–135.

[17] M. R. Haghighat, *Symbolic Analysis for Parallelizing Compilers*, Kluwer Academic Publishers, Dordrecht, 1995.

[18] K. E. Hillstrom, *JAKEF - a portable symbolic differentiator of functions given by algorithms*, Technical Report ANL/82–48, Mathematics and Computer Science Division, Argonne National Laboratory, 1982.

[19] P. Hovland, C. Bischof, D. Spiegelman, and M. Casella, *Efficient derivative codes through automatic differentiation and interface contraction: An application in biostatistics*, Preprint MCS-P491-0195, Mathematics and Computer Science Division, Argonne National Laboratory, 1995. To appear in SIAM J. Scientific Computing.

[20] M. Iri, *History of automatic differentiation and rounding estimation*, in Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. F. Corliss, eds., SIAM, Philadelphia, Penn., 1991, pp. 1–16.

[21] A. John and J. C. Browne, *A constraint-based parallel programming language*, Tech. Rep. TR95-42, Department of Computer Science, University of Texas at Austin, 1995.

[22] M. Monagan, *An implementation of automatic differentiation in Maple*, 1995. Personal Communication. Software available in the MAPLE Share library.

[23] C. Polychronopoulos, *Parallel Programming and Compilers*, Kluwer Academic Publishers, Boston, Massachusetts, 1988.

[24] A. Rhodin, U. Callies, and D. Eppel, *GESIMA 90 – An object-oriented approach to program a mesoscale model and its adjoint*, 1994. Talk presented at the General Assembly of the European Geophysical Society at Grenoble, April 25-29.

[25] N. Rostaing, S. Dalmas, and A. Galligo, *Automatic differentiation in Odyssee*, Tellus, 45a (1993), pp. 558–568.

[26] D. Shiriaev, *Fast automatic differentiation for vector processors and reduction of the spatial complexity in a source translation environment*, PhD thesis, Department of Mathematics, Universität Karlsruhe, 1993.