Adjoining Independant Computations

Laurent Hascoët
Stefka Fidanova
Christophe Held

ABSTRACT The *reverse* or *adjoint* mode of Automatic Differentiation is a software engineering technique that permits efficient computation of gradients. However, this technique requires a lot of temporary memory. In this paper, we present a refinement that reduces memory consumption in the case of parallel loops, and we give a proof of its correctness, based on properties of the *data-dependence graph* of adjoint programs and parallel loops. This technique is particularly suitable for assembly loops that dominate in mesh-based computations. Application is done on the kernel of a realistic Navier-Stokes solver.

1 Introduction

Adjoints are a popular Numerical Analysis tool to compute efficient and accurate gradients [7] [9], for optimisation [1], data assimilation [11], etc... Given an original program that computes a function f, the *reverse* mode of Automatic Differentiation generates an adjoint program that computes the gradient of f. In its basic version, sketched in Figure 1, the adjoint program is composed of two successive phases: the forward sweep, that reproduces the original program, with extra instructions that push intermediate values (the '*tape*") on a stack, and the reverse sweep, that (left-)multiplies the vector of adjoint variables with the local Jacobian of each elementary instruction, in the order *reverse* to that of the original program. These Jacobian matrices use values that are popped from the *tape*.

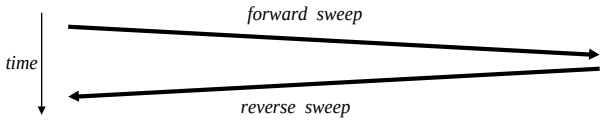

FIGURE 1. *Forward and reverse sweeps of reverse AD*

The tape mechanism is the main drawback of this adjoint program: all intermediate values required by the reverse sweep must be stored *before* any of them is used. In this paper, we prove that parallelisable or reduction loops can be adjoined in a special way, with each loop iteration immediately followed by its adjoint iteration, therefore vastly reducing the total need of

storage. Such loops are frequent and can be detected automatically. See for example *assembly loops* in mesh-based computations.

Our technique is a transformation of the standard adjoint program. Its correctness relies on an isomorphism between the data-dependence graphs of the forward and reverse sweeps, which is proven in section 2. Using this isomorphism, section 3 shows how any parallelisable loop can be transformed. Section 4 gives some experimental results, and we conclude with a comparison with two related approaches.

2 Isomorphism between Data-Dependence Graphs

Classically, a program transformation that only changes the execution order is correct if and only if the new order is compatible with the Data-Dependence Graph (DDG). Actually, the DDG is the sub-order of the original execution order that must be respected by any reordering.

Let us define the nodes of our original program's DDG as the elementary instructions (e.g. assignments). For each original elementary instruction, we define exactly one adjoint DDG node, that contains all instructions achieving the product with the local Jacobian. By definition, we have a bijection between original and adjoint nodes.

The arrows of a DDG reflect *data-dependences*, that relate uses of variables. Dependences may be write-to-read, read-to-overwrite, or write-to-overwrite. We shall set one unique arrow in the DDG from node N_1 to node N_2 iff there is at least one dependence from some variable use in N_1 to a use of the same variable in N_2. In this paper, we refine the usual notion of dependences by making a special case for *increments* (`x += ...`). Successive increments to a variable can be done in any order, because of commutativity-associativity of the sum. Therefore this variable must not generate a dependence between these increments. However, this requires that increments are *atomic*. Two increments on x done exactly in parallel lead to race conditions. There are various ways to ensure atomicity. For example when the program is executed sequentially, atomicity is guaranteed.

Notice that the DDG is different from the *computational graph* [6]. One node of the computational graph represents one run-time instance of a variable, i.e. a value, whereas one node of the DDG represents one instruction in the program, and summarizes all its executions at run-time.

For example, Figure 2 shows on the left a small DDG and on the right its adjoint DDG. Only DDG arrows due to variables $\mathbf{x}$ and $\bar{\mathbf{x}}$ (adjoint of $\mathbf{x}$) are shown. Nodes and adjoint nodes face each other: notice therefore that the DDG on the right is executed bottom-up! Observe that there is no dependence between the two increments on $\bar{\mathbf{x}}$. Notice also the two dotted arrows reaching the uses of variables $\mathbf{a}$ and $\mathbf{b}$ on the right: these are *tape dependences*, that denote uses of intermediate values in the reverse sweep.

Each tape dependence requires one value stored into the tape.

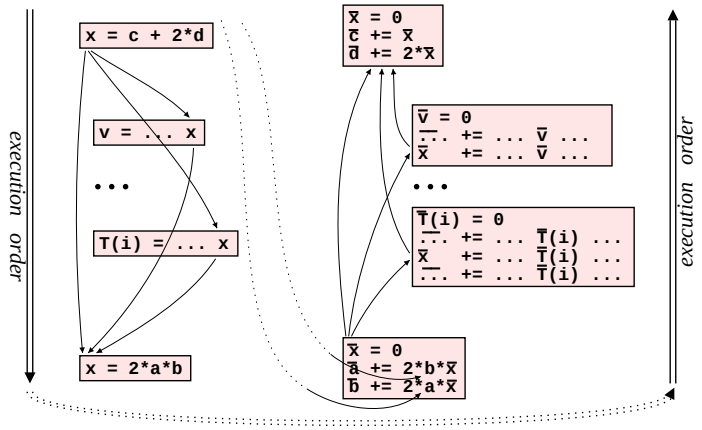

FIGURE 2. *Adjoining a Data-Dependence Graph*

With the above definitions, we prove the following isomorphism:
There exists an arrow from node N_1 to N_2 in the original DDG if and only if there exists an arrow from node adjoint(N_2) to adjoint(N_1) of the adjoint DDG.

Proof sketch: Consider any variable **x** that has an adjoint variable $\bar{\mathbf{x}}$. DDG arrows due to **x** can be deduced from a partition of the DDG nodes in four categories: (n) (no use of **x**), (i) (**x** incremented only), (r) (**x** only read), and (w) (other cases). Variable **x** generates a dependence from a node to another in all cases except when one node is (n), or from (r) to (r), or from (i) to (i). Adjoining operates on these categories. Precisely: (i)↦(r), (r)↦(i), (w)↦(w), and (n)↦(n). Exploring all cases proves that **x** generates an arrow from node N_1 to N_2 iff $\bar{\mathbf{x}}$ generates an arrow from adjoint(N_2) to adjoint(N_1). □

3 Adjoining a Parallelisable Loop

Consider a terminal parallelisable loop. "Terminal" means that the loop's results are not used until the reverse sweep starts. Therefore, all data-dependences leaving the loop go directly to the program's output. "Parallelisable" means that the loop iterations are *independent*, i.e. there are no *loop-carried* data-dependences. Figure 3 shows the standard adjoint. On top is the original loop, with tape storage, and under it is the reverse loop, executing the adjoint of the original instructions in the reverse order. Each vertical grey shape represents a particular loop iteration. Thin arrows are the DDG arrows *plus* tape dependences. Since the original loop is parallel,

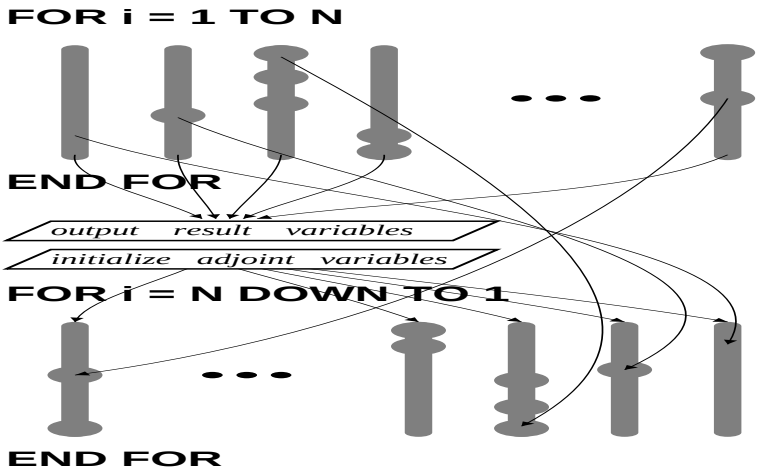

FIGURE 3. *The adjoint for a terminal parallel loop, and its dependences*

there are no loop carried-dependences. Because of DDG isomorphism, there are no loop-carried dependences on the second loop either. Furthermore, there are no dependences from the first loop to the initial adjoint values, nor from the original results to the second loop. We can thus inverse the reverse loop's iterations, then merge the two loops, yielding figure 4. Since

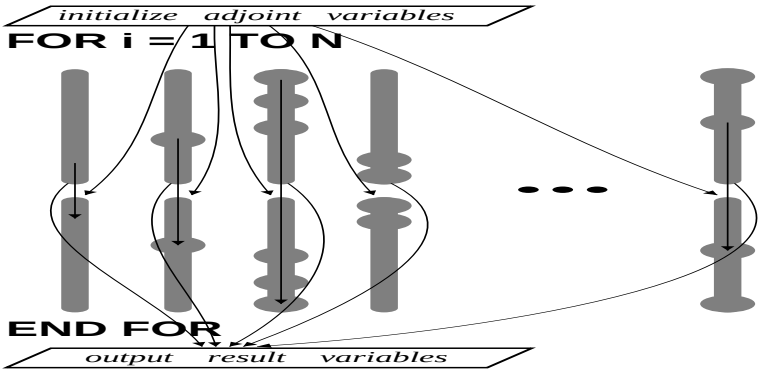

FIGURE 4. *Transformed adjoint equivalent to figure 3*

the tape dependences become local to each iteration, the total memory required for tape storage is divided by **N**.

This can be generalised to loops containing "additive reductions" i.e. global sums, because the adjoint of a global sum simply requires the "spread" of a unique adjoint value to each iteration.

We must finally lift the restriction to *terminal* loops. Figure 5 illustrates our method, called *dry-running*, a sort of *checkpointing* [5]: to make a program *fragment* terminal, first execute it without tape storage (thin arrow

on figure). Then, when the reverse sweep reaches the end of the fragment, execute it again *with* tape storage, and continue immediately with its reverse sweep. Of course this requires that we save and restore enough data for recomputation (black and white dots on figure).

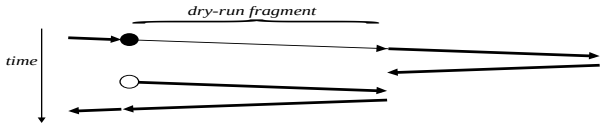

FIGURE 5. *Dry-running a selected fragment of a program*

4 Experimental results

We took the kernel of a real-size Navier-Stokes solver (7 parallel loops) on an unstructured triangular mesh ($\simeq$ 1000 elements). The standard adjoint was built by ODYSSÉE [3], then we transformed parallel loops by hand. Dry-running was necessary because not all 7 loops were "terminal". Of course, the gradients found are identical. Let us compare memory usage:

	reference	**transformed**
memory for original computation	171,000	171,000
memory for adjoint variables	297,000	297,000
memory for dry-running	0	47,000
memory for tape storage	2,400,000	800
total	2,868,000	515,800

The saving factor (3000 on tape) corresponds to what could be expected, i.e. the product of the number of parallel loops by the number of parallel iterations. On the other hand, the dry-running mechanism used 47 kilobytes of memory. Rather surprisingly, the execution time of the transformed version is only slightly longer (3%). The increase due to dry-running was probably partly compensated by the economy in memory allocation time and a better data locality.

5 Related work and conclusions

The above technique can be combined profitably with others, such as fine-grain reduction of the tape, as described in [2]. Notice that [4] also provides a special strategy for adjoining parallel loops. But further comparison is difficult because TAMC is based on recomputation rather than tape storage. Let us compare with two very similar techniques that reorder iterations of a

loop and its adjoint. In the general case, the loop is not terminal, and thus we need dry-running, as sketched on Figure 6. In [8], the technique amounts

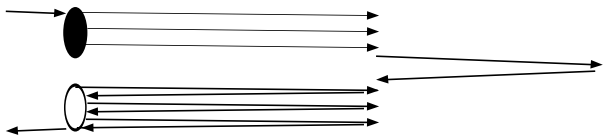

FIGURE 6. *our transformation, using dry-running*

to dry-running each iteration separately (Figure 7). Each iteration stores enough data for its own recomputation, which is generally more expensive than above. The *tape* savings are the same. On the other hand, the loop need not be parallel. Similarly, J.C. Gilbert and F. Eyssette, computed the

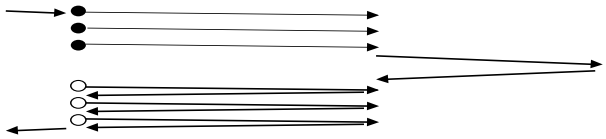

FIGURE 7. *dry-running on each iteration*

explicit Jacobian matrix for each iteration (Figure 8). There is no more *tape* to be stored, nor checkpoints, but the Jacobians may use a considerable memory. Again, the loop need not be parallel. A general strategy

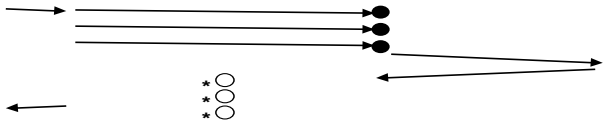

FIGURE 8. *local Jacobian technique*

emerges from the above. If the loop is parallel, we recommend application of our method, especially when the loop is terminal. Otherwise, dry-running each iteration (Figure 7) is a good backup technique. The local Jacobian technique could make sense if loop body is computationally very expensive relative to the number of entries in the Jacobian.

This technique is a very efficient optimisation, already used by people who write adjoint programs by hand [10]. So far, this was done with no guaranty of correctness. Under precise hypotheses, that can be verified mechanically by any parallelisation tool, we proved that this transformation is correct. Thus it can be done automatically by source code analysis and transformation. This makes the link with notions from parallelisation theory such as data-dependences, and this improves our understanding of Automatic Differentiation techniques.

6 References

[1] J.M. Malé B. Mohammadi and N. Rostaing-Schmidt. Automatic differentiation in direct and reverse modes: application to optimum shapes design in fluid mechanics. In M. Berz, C. Bischof, G. Corliss, A. Griewank, eds., SIAM, editor, *Computational Differentiation: Techniques, Applications and Tools*, pages 309–318, 1996.

[2] C. Faure and U. Naumann. The trajectory problem in automatic differentiation. 2000. Proceedings of AD2000, Nice, France.

[3] C. Faure and Y. Papegay. Odyssée user's guide. version 1.7. Rapport technique 224, INRIA, 1998.

[4] R. Giering. Tangent linear and adjoint model compiler, users manual. Technical report, 1997.

[5] A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54, 1992.

[6] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* SIAM, 2000.

[7] J.L. Lions. *Optimal control of systems governed by partial differential equations.* Springer, 1971.

[8] B. Mohammadi P. Hovland and C. Bischof. Automatic differentiation of navier-stokes computations. Technical report, Argonne National Laboratory MCS-P687-0997, 1997.

[9] O. Pironneau. *Optimal shape design for elliptic problems.* Springer, 1982.

[10] C. Sevin. *Optimisation de formes en mécanique des fluides numérique.* PhD thesis, Université Pierre et Marie Curie, 1999.

[11] O. Talagrand. The use of adjoint equations in numerical modelling of the atmospheric circulation. In A. Griewank, G. Corliss, eds., SIAM, editor, *Automatic Differentiation of Algorithms: Theory, Implementation and Application*, pages 169–180, 1991.