Switchback: Profile-Driven Recomputation for Reverse Mode

Mike Fagan and Alan Carle

Department of Computational and Applied Mathematics
Rice University, 6100 Main Street, Houston, TX 77005–1892, USA
{mfagan, carle}@caam.rice.edu
http://www.caam.rice.edu

Abstract. Many reverse mode codes created by Automatic Differentiation (AD) tools use too much storage. To help mitigate the storage requirement, AD users resort to a technique called *recomputation*. Acceptable performance, however, requires judicious use of the recomputation technique. This work describes our approach to constructing good recomputation for Adifor 3.0-generated reverse mode codes by using a standard execution time profiler.

1 Introduction

Modern Automatic Differentiation (AD) tools have enabled scientists and engineers to easily generate adjoint, or reverse mode programs for computation of gradients. Many reverse mode computations suffer from overuse of both primary and file system memory resources. Consequently, further development of reverse mode codes frequently centers on using *recomputation* to reducing the amount of storage required. Recomputation, however, trades time for space, and must be used judiciously, or else the reverse mode computation will become too slow. AD tools implementing the reverse mode include ADOL-C [1], Odyssée [2], and TAMC/TAF [3].

While working on a shape optimization problem, we discovered that we needed recomputation. We were using Adifor 3.0 [4], the successor of [5], to generate reverse mode code for the computation of geometric derivatives in CFL3D [6], a fluid dynamics code. For the problem instances of interest to us, the reverse mode log exceeded the bounds for the local disk storage. Even after using a standard fixed-point technique to drastically reduce the storage, we still could not fit our computation on the local disk. Ergo, we saw that we must use recomputation to compute the derivatives of interest. The recomputation technique we designed for our problem employed the two main ideas: (a) the construction of local recomputation based on subprograms already present in the user's code and (b) the use of a standard program execution profiling tool to indicate subprograms in which recomputation can be effectively applied.

The remainder of this paper describing our technique is organized as follows. Section 2 gives some general background material on the reverse mode of automatic differentiation, including some discussion of the storage problem. Section 3

P.M.A. Sloot et al. (Eds.): ICCS 2002, LNCS 2330, pp. 1029–1038, 2002.

gives a general overview of the recomputation mechanism, and how it is used to reduce reverse mode storage problems. Section 4 gives the relevant details of the specific Adifor 3.0 reverse mode technique, including the Adifor 3.0 "switchback" recomputation mechanism. Section 5 details how we use a standard execution profiler to select appropriate subroutines for recomputation. Section 6 describes the derivative computation problem under study, including systemic constraints. It also details how we used Adifor 3.0 recomputation to render the problem soluble. Finally, Sect. 7 shows the results of our recomputation transformation, and estimates the performance effect.

2 Background Material

The time cost for a (straightforward) reverse mode computation is proportional to the number of "output", or "dependent" variables, and *does not* depend on the "design" or "independent" variables. This property is what makes reverse mode nearly ideal for optimal control or optimal design applications. For example, our shape optimization problem required the derivatives of each grid point with respect to the lift-over-drag ratio. See [7] for a thorough description of reverse mode resource cost, and a complete treatment of reverse mode mathematics.

The following simple example illustrates the reverse mode. Suppose we are given a simple program

$$\begin{aligned} y &= f(x_1, \ldots, x_n) \\ z &= g(y, x_1) \end{aligned}$$

that computes z given $x_1, \ldots, x_n$. Let $\overline{v} := \partial z / \partial v$ denote the derivative, or adjoint, of the dependent variable z with respect to an intermediate variable v. Then the reverse mode computation for the derivatives of z with respect to the independent variables $x_1, \ldots, x_n$ is given by

$$\begin{aligned} &\text{Step 1:} \quad \overline{z} = 1, \quad \overline{y} = 0, \quad \overline{x_1} = 0, \quad \ldots, \quad \overline{x_n} = 0, \\ &\text{Step 2:} \quad \overline{y} = \overline{y} + \frac{\partial g}{\partial y}\overline{z}, \quad \overline{x_1} = \overline{x_1} + \frac{\partial g}{\partial x_1}\overline{z}, \quad \overline{z} = 0, \\ &\text{Step 3:} \quad \overline{x_1} = \overline{x_1} + \frac{\partial f}{\partial x_1}\overline{y}, \quad \ldots, \quad \overline{x_n} = \overline{x_n} + \frac{\partial f}{\partial x_n}\overline{y} \ . \end{aligned}$$

Note that the function computation order is y, then z, but the reverse mode derivative calculation $\overline{z}$, then $\overline{y}$. This reversal of the function computation ordering is what gives "reverse mode" its name.

An AD tool accepts a user program as input, and outputs an augmented program that computes specified derivatives via reverse mode computation in addition to the original function computation. As illustrated by the preceding example, reverse mode control flow is the reverse of the original function control flow. For simple assignment statements, this control flow reversal is fairly simple. When conditional and unconditional branches, subroutine calls, loops, and other

nontrivial program constructs are added to the statement mix, however, the code implementing the reverse control flow can be quite complex. In most real world codes, implementing the control flow reversal requires saving trace information.

In general, a reverse mode computation consists of two phases: a forward, or *logging* phase, and a reverse, or *derivative* phase. During the forward phase, some program state information is saved, but the control flow of the program is unchanged. The forward phase computes the same function outputs as the original unaugmented program. During the reverse phase, however, the control flow of the program is reversed from the normal forward flow in order to compute the derivative accumulations. Derivative accumulations require the partial derivatives of the left hand side of the assignment statement with respect to each variable used on the right hand side of the assignment statement. The responsibility of the forward phase is to log sufficient information to both correctly reverse the control flow and compute the necessary partial derivatives for the assignment statements.

The quasi-mathematical model of assignment statements given in the preceding example oversimplifies a crucial characteristic of computer program assignment statements. The problem can be seen in another simple example where the program fragment (line numbered for convenience)

```
(1) v1 = a ** 2 + b ** 2
(2) a = sin(z)
```

is given. Forward control flow proceeds from statement (1) to statement (2). So, the reverse pass must assure that the derivative accumulations for (2) occur before the derivative accumulations for (1). Also, note that for statement (1), $\partial \mathtt{v1}/\partial \mathtt{a} = 2\mathtt{a}$, where the value of `a` is the value it had *before* the statement (2) assignment. The forward phase must ensure that the correct value of the partials for statement (1) are computed.

Since assignment statements change the value of a variable, the fundamental problem for reverse mode is saving enough information to compute the partials. There are two primary saving strategies:

1. Compute partials for each assignment in the forward pass, and save them. Use the saved partials during the derivative accumulation of the reverse pass.
2. Save the values of left hand sides of assignments during the forward pass. In the reverse pass, restore the saved values as needed to compute the partials for the given statement.

Clearly, either strategy entails a potentially huge amount of auxiliary storage.

In summary, the *time* cost of reverse mode computation is very favorable for many problems, especially optimization problems. The *space* cost, however, might be problematic.

3 The Recomputation Alternative

To avoid storing values for every single assignment, the reverse mode computation could instead store a *checkpoint*, and when partials are needed, restart

the computation using saved checkpoint, recompute the state of the computation [8]. The obvious drawback to such a scheme is the time cost for doing the recomputation.

There are several technical problems associated with employing a recomputation scheme:

- How do we determine what to save as a checkpoint?
- How do we arrange for derivative computation control flow to use the checkpoint?
- How do we determine where to take the checkpoints?

When considering what information to save for checkpoints, we note that the program itself has a natural level of granularity—the subprogram, i.e., subroutines and functions in Fortran. For a given subprogram call site, the natural checkpoint is the input values for that routine. The input values for a routine consist of the read values for routine arguments, as well as global variables that are read by the routine. Given the input values, the subprogram intermediate state can be restored by simply calling the subprogram with the same inputs Consequently, if we focus on checkpointing at the level of a subprogram call, then we need only save the input values to effectively checkpoint the call.

Similarly, the restart mechanism for a subprogram is relatively simple: just call it. The differentiated subprogram components can also be called as needed. An example of this will appear in the following section.

4 Adifor 3.0 Reverse Mode Particulars

In order to ensure reasonable computational *time* performance, Adifor 3.0 reverse mode *by default* stores all left hand sides of relevant assignments, and restores the prior values during reverse mode partial computation. More precisely, given an assignment statement of the form

```
foo = bar * baz
```

Adifor 3.0 generates both forward phase code and reverse phase code. The forward phase code is given by

```
call STORE_r(foo)
foo = bar * baz
```

and the reverse phase code is generated in the form

```
a_bar = a_bar + baz * a_foo
a_baz = a_baz + bar * a_foo
a_foo = 0.0
call LOAD_r(foo)
```

where the prefix `a_` is used to denote an adjoint of the corresponding variable.

Adifor 3.0 handles subroutine call control flow by using a switch to select either forward pass or reverse pass computation. To illustrate, suppose we have the following routine with two inputs and two outputs:

```
subroutine SS(in1,in2,out1,out2)
  ! compute stuff
end
```

Then Adifor 3.0 generates an associated reverse mode subroutine that looks like this:

```
subroutine a_SS(dir,in1,a_in1,in2,a_in2,out1,a_out1,
+               out2,a_out2)
   if (dir .eq. FWD) then
     ! compute stuff and log intermediate stuff
   endif
   if (dir .eq. REV) then
     ! restore intermediates, compute partials,
     ! accumulate derivatives.
   endif
end
```

Here, the parameter `dir` controls the forward phase, `FWD`, or the reverse phase, `REV`, of the standard reverse mode computation which is driven by

```
call a_SS(FWD,...)
call a_SS(REV,...)
```

The Adifor 3.0 recomputation technique is called the "switchback" transform. A routine that uses switchback uses the forward and reverse components from conventional Adifor processing, but has a different schematic. The switchback version of the preceding `SS` routine would be:

```
subroutine sb_a_SS(dir,in1,a_in1,in2,a_in2,
+                  out1,a_out1,out2,a_out2)
   if (dir .eq. FWD) then
     call STORE_r(in1) ! save input arg 1
     call STORE_r(in2) ! save input arg 2
     call SS(in1,in2,out1,out2) ! make the (undiff) call
   endif
   if (dir .eq. REV) then
     call LOAD_r(in2) ! restore arguments
     call LOAD_r(in1)
     call a_SS(FWD,...) ! invoke normal fwd pass behavior
     call a_SS(REV,...) ! invoke normal reverse pass behavior
   endif
end
```

Note that constructing a switchback routine uses both the undifferentiated subroutine as well as the differentiated routine.

The user selects routines to which the switchback transform will be applied, and specifies the names in the Adifor control file. In addition, for each selected switchback routine, the *user* must nominate the input values to be saved. This user-based nomination of input values is admittedly crude and tedious, but it will be automated in future Adifor releases.

An actual example of using the switchback mechanism is as follows. Suppose that the following subroutine

```
subroutine flx(a, n, x, flxout)
double precision a, x(n,n), flxout(n), tmp, tmp1
integer n
integer ix(100),iy(100)
double precision z(100),zz(100)
common /flxcomm/ ix,iy,z,zz

tmp = flxx(a,x)
tmp1 = flxz(a,z)

call flxupd(flxout,tmp,tmp1,a,n,x,z)

end
```

is given as part of a larger program where the semantics of the routines `flxx`, `flxz`, and `flxupd` is not known a priori.

After inspecting this routine, and the various calls made by it, we determine that subroutine arguments `a`, `x`, and `n` are read by `flx`. The variable `z` is *not* read prior to assignment by `flxx` or `flxz`, it is an *output* variable. Similarly, `flxout` is an output variable, and `zz` is not used at all. Hence, the switchback mechanism needs only save `a`, `n`, and `x`. This analysis is encoded in the Adifor control script these statements:

```
Switchback
 flx:
  a,n,x
```

Adifor will then ensure that `flx` uses recomputation rather than logging to compute the necessary partial derivatives for the `flx` routine.

5 Profiling for Recomputation

Selecting *all* routines for recomputation would result in exceptionally poor performance. In addition, not all routines are good candidates for recomputation. If the checkpoint is nearly as large as the amount of storage used, then nothing has been gained. The ideal candidate for recomputation would be a routine that uses a small number of inputs, but computes a large number of intermediate results, resulting in a large amount of storage used.

To select candidate routines for conversion to switchback, we follow the time-honored computer science paradigm of profiling a code to determine which routines are consuming the majority of the resource of interest. One then focuses intellectual effort on the routines that would benefit the most. For Adifor-generated derivative code, this is especially easy. Since Adifor-generated code stores by calling subroutines, e.g., `STORE_r`, a standard execution profiler such as Unix `gprof` will supply information on which routines make the most calls to the storage routine. Sample `gprof` output for our problem appears in Sect. 6. By focusing the inspection effort on the top "few" routines that use the most storage, some good recomputation candidates can likely be found.

To use profile-driven recomputation for an Adifor-generated code, the following steps are recommended:

1. Run Adifor in reverse mode to obtain default logging-based derivative code
2. Compile the derivative code with profiling support enabled (`-pg` option on most Unix systems).
3. Run the profile-enabled derivative code on a sample problem.
4. Run the profile post processor to discover which routines call the storage routines the most.
5. Inspect the top 10, say, of these to see if they use a "small" number of input values.
6. Assuming you find some, apply the switchback transform to the candidate routines.

The following section details our use of this recipe for our problem.

6 Reverse Mode Computation for CFL3D

The problem under study was an aerodynamic shape optimization for a wing under steady state flow conditions. The entire application consisted of a custom grid generator, a flow solver, and a single scalar objective function computation, the lift-over-drag ratio. Of these components, only the flow solver presented any difficulty in reverse mode gradient computation. The flow solver was CFL3D version 4. CFL3D [6] is a thin-layer Navier-Stokes solver that supports multiple zones, and uses MPI-based parallelism. CFL3D is approximately 126,000 lines of Fortran 77 code. The code is maintained by NASA Langley.

The grid for this problem was divided into 10 zones of size $65 \times 17 \times 17$. The execution platform was an Origin 2000, with 32 processors, and a relatively small local disk. The I/O system, however, required that all of the processor specific I/O be written on the local disk. In particular, that meant that the logs for the reverse mode gradient computation had to fit onto a small local disk of 500 megabytes. In addition, the disk did not perform well when it was almost full. So, ideally, we would like to keep the logs well under the 500 megabyte limit.

The flow solution ran for 1000 steps, and, using the default reverse mode computation of Adifor 3.0, our logging instrumentation revealed that the log for all 1000 steps would be on the order of 500 Gigabytes *per processor*.

Our first move to reduce the log size was to take advantage of the steady state nature of the computation, and reduce our logging requirements to a single step of the flow solver. This derivative accumulation is iterated over this single step until convergence. This technique is well known to AD users; see [9] for mathematical details. Even when reduced to a single step, our logging instrumentation revealed the following information:

```
blocks = 4129180
integers = 29942967
logicals = 232048
reals =  105
doubles = 46865895
```

This makes the total log size 512, 144, 360 bytes which is still too large to fit into memory.

A sample portion of the profiling run for the CFL3D application shows:

```
-----------------------------------------------
                              20572643/46865895    a_flxt_[15]
                              18484228/46865895    a_fhat_ [22]
                              16314714/46865895    a_diagj_ [59]
                              16314714/46865895    a_diagk_ [60]
                              16314714/46865895    a_diagl_ [61]
                              14901631/46865895    vec_v_ [91]
[6]     78.2   39.15    0.00  46865895         store_d_ [6]
```

The `gprof` output is dense, but contains the data we want. Line 7, on which `store_d_ [6]` appears indicates the total number of calls, in this case 46, 865, 895. The lines *above* the `store_d_ [6]` line indicate routines that call `store_d_`, and indicate what fraction of the total calls to `store_d_` are made by the given routine. For example, line 1 of the sample output shows that routine `a_flxt_` called `store_d_` 20, 572, 643 times out of the 46, 865, 895 times that `store_d_` was called. Furthermore, `gprof` sorts this output so that the most frequent callers of a given routine appear at the top of the list. In the sample output, the top 6 callers of `store_d_` are `a_flxt_`, `a_fhat_`, `a_diagj_`, `a_diagk_`, `a_diagl_`, and `vec_v_`. Note that only the top 6 are listed *in the sample*. The full `gprof` output lists all the routines, and took several pages of output, as `store_d_` is called by 247 routines.

By focusing our attention on the top 10 users of the `STORE_d` routine as determined by `gprof`, we found 4 routines that had relatively small number of input values, but consumed a relatively large amount of log. Specifically, `a_flxt_`,`a_diagj_`, `a_diagk_`, and `a_diagl_` as shown in the sample output. The `a_fhat_`routine used a fair amount of common block reads, so it could not be used in our recomputation scheme. We applied the switchback transform to these 4 routines.

7 Results

By applying switchback to 4 routines, we were able to reduce the log size to these values:

```
blocks = 1694684
integers = 15975367
logicals = 223940
reals = 105
doubles = 29520599
```

with total log requirements now 307,741,176 bytes, which fit comfortably on the local disk. This reduction enabled us to compute our desired gradients using the small local disk for log storage.

Since we could not actually compute derivatives for the given problem without switchback, we cannot say what the performance cost of switchback was. To estimate the performance cost of switchback for this code, we used a much smaller problem, and compared the switchback version to the black box version. NASA researchers generated a $17 \times 9 \times 9$ grid and a $33 \times 9 \times 9$ grid for our experiment. We were able to run both grids on our platform. Table 1 shows that the time penalty for switchback, on this particular problem appears to be about 37%.

Table 1. Timings (sec) of the two versions using Unix system clock

Grid	Black Box	Switchback
$17 \times 9 \times 9$	262.13	359.14
$33 \times 9 \times 9$	556.63	773.57

8 Concluding Remarks

To eliminate excessive storage requirements in a reverse mode application, application developers often resort to recomputation. Overuse of recomputation, however, damages time performance. Consequently, a reverse mode application developer must choose where to apply the recomputation technique. In the work described here, we showed how the time-honored computer science principle of profiling can be used to focus the developer's efforts on the routines with the most potential improvement. The specific AD tool used in the work was Adifor 3.0, but the profiling idea should be generally applicable.

As a final note, our experience with this technique has guided our Adifor development effort to include automatic detection of "read first" variables so that the switchback mechanism can be applied more easily by users.

Acknowledgments

Thanks to Los Alamos Computer Science Institute, and NASA Langley for their support of this research.

References

1. Griewank, A., Juedes, D., Utke, J.: ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. ACM Transactions on Mathematical Software **22** (1996) 131–167
2. Rostaing, N., Dalmas, S., Galligo, A.: Automatic differentiation in Odyssée. Tellus **45A** (1993)
3. Giering, R., Kaminski, T.: Recipes for adjoint code construction. ACM Transactions on Mathematical Software **24** (1998) 437–474
4. Fagan, M., Carle, A.: Adifor 3.0 overview. Technical Report CAAM–TR00–03, Rice University, Department of Computational and Applied Mathematics (2000)
5. Bischof, C., Carle, A., Khademi, P., Mauer, A.: ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. IEEE Computational Science & Engineering **3** (1996) 18–32
6. Rumsey, C.L., Biedron, R.T., Thomas, J.L.: CFL3D: Its history and some recent applications. Technical Report NASA Technical Memorandum 112861, NASA Langley Research Center (1997)
7. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. SIAM, Philadelphia (2000)
8. Griewank, A.: Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. Optimization Methods and Software **1** (1992) 35–54
9. Christianson, B.: Reverse accumulation and attractive fixed points. Optimization Methods and Software **3** (1994) 311–326