Automatic Differentation of the TACO2D Finite Element Code Using ADIFOR

Alan Carle
Mike Fagan

CRPC-TR95568
September 1995

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

revised November 1996

Automatic Differentiation of the TACO2D Finite Element Code Using ADIFOR*

CRPC-TR95568

Alan Carle and Mike Fagan

September 29, 1995

Abstract

We use ADIFOR to generate derivative code for **taco2d**, a transient radiative heat transfer code that uses a finite element method. The derivatives produced by the ADIFOR-generated code are accurate, and take less time to produce than finite differences.

1 Introduction

In this report, we describe various aspects of our differentiation project for a radiative heat transfer code. This code, called **taco2d**, models various transient radiative heating effects for two-dimensional geometries using a finite element method [?]. Our primary focus on taco2d was for the design of vlsi heating chambers. Designers at Sandia National Laboratories were using optimization methods to improve their heating chamber designs [?]. Consequently, the designers were interested in obtaining accurate derivatives, and in obtaining them quickly. As we will demonstrate in this report, both objectives are met by using the ADIFOR automatic differentiation tool.

Our report divides into a few sections. First, we summarize our methodology. Second, we describe the derivative generation process. Third, we demonstrate the accuracy of the generated derivative code. Fourth, we discuss the time requirements of the generated derivative code. Fifth, we discuss the space requirements of the generated code. Finally, we close with a summary of results and some speculation about future investigations.

2 Methodology

Whenever feasible, we prefer to study the aspects of ADIFOR-generated derivatives on multiple platforms. For taco2d, we were able to use both the original SGI Power Challenge (SGI/R8000) platform, as well as an IBM RS 6000 Model 390 platform for our study. We used only one node (out of four) on the SGI platform. As an additional point of interest, we processed the source code on Sun workstations. So, our methodology consisted of these steps:

- Obtain the SGI platform source from Sandia, copy it to the Sun platform.
- Port the code to the IBM RS6000 platform.
- Run ADIFOR on the Suns, correcting all inconsistencies in the original code.

*This work was supported by Sandia National Laboratory under contract LC-2683. Additional computing resources provided by NASA Ames High Performance Computing and Communication Program Testbed 1 Cooperative Research Agreement (through IBM P.O. # 835600Z-E1).

- Port the ADIFOR-generated derivative code to the IBM RS6000.
- Port the ADIFOR-generated code to SGI.
- Run accuracy and efficiency tests on both platforms.

Porting taco2d from the SGI to the RS6000 was straightforward. As a byproduct of the porting process, however, we discovered and corrected a minor logic error in the original source. The original source's adaptive convergence test worked properly for non-linear cases, but sometimes resulted in an infinite loop in linear cases. We corrected this error by moving the convergence test code to a more appropriate block.

Sandia provided control and data files for several test cases. The test cases separated into a simple case consisting of 2 heating elements and 800 total finite elements, and a more realistic test consisting of 5 heating elements and 2424 total finite elements. We used the simple case for debugging and development, but we used the more realistic test case for our timing and accuracy tests.

3 Generating Derivatives

To generate derivative code, ADIFOR requires that all source code be type consistent, both within a module, and between modules. Consequently, part of the preparation of source code for ADIFOR processing often requires correcting type inconsistencies, especially in mature codes. In taco2d, the type inconsistencies originated from two main sources: a single input routine that could read real, integer, or character arguments; and routines that reused previously allocated memory.

The input routine followed a hoary Fortran tradition of passing a selection argument to make the same routine perform different actions, depending on the selection argument. Thus, the input routine can read integers in one part of the code, reals in another. This multiple usage, however, is not type consistent (even though it is part of traditional Fortran culture). The technique for handling this kind of inconsistency is simple renaming. Rather than pass a selection argument, we simply cloned copies of the read routine, one uniquely named clone for each different value of the selection argument. We automated this cloning transformation with a short perl script.

The second source of type inconsistencies involved memory management. Consequently, the various inconsistencies were distributed throughout several routines. These inconsistencies arose when the taco2d developer would reuse some real number arrays (and parts of arrays) as integer arrays (and rarely, some integer arrays were reused as real arrays). To handle most of these reuse inconsistencies, we used a technique involving Fortran equivalence. The arrays being reused resided in a several globally accessible (Fortran common block) areas. Thus, for real arrays in the global areas, we simply declared an integer array, and equivalenced the real and integer array. This technique allowed us to change the array name to its alias at call sites where there was an inconsistency. Again, rather than change these cites by hand, we wrote a perl script to alter the appropriate arguments. This technique sufficed for most of the troublesome call sites, but not all of them. Fortran equivalence is limited to common blocks, and some routines reused subroutine arguments, instead of common storage. Consequently, we were forced to add extra arguments to some of these routines. This kind of change requires some non-trivial (human) analysis, and, unfortunately, is not easily automated. In addition, these changes took more than one iteration of consistency checking to produce completely consistent source code for ADIFOR processing.

4 Accuracy

We validated the ADIFOR derivatives by comparing them to a progression of (forward) finite difference runs. We chose three different step sizes: .01% of input value, .001% of input value, and .0001% of input value. The results of the finite difference runs had several notable features. Most importantly, the results indicated the convergence of finite differences to the ADIFOR derivative value. In addition, the results indicated the difficulty of choosing an appropriate step size. An excerpt of our comparison test for the SGI platform appears in Figure 1. We also validated the ADIFOR derivative values across platforms. Our derivative values were identical across platforms. Figure 2 shows some typical examples.

Element Label	**Difference at step .01%**	**Difference at step .001%**	**Difference at step .0001%**	**ADIFOR** Derivative
20	-3.11764706e+00	3.60000000e+00	3.99999999e+00	3.64022605E+00
594	4.04509804e+01	4.27000000e+01	4.30000000e+01	4.25795335E+01
2405	8.61274510e+01	8.00000000e+01	8.00000000e+01	7.97472102E+01

Figure 1 Accuracy of Finite Differences vs ADIFOR-generated derivatives (excerpt)

Element	**RS6000 Derivative**	**SGI Derivative**
51	3.13683799E+01	3.13683799E+01
998	2.42428801E+00	2.42428801E+00

Figure 2 ADIFOR-generated derivative values on RS6000 vs SGI

5 Time Requirements

Typically, a study of the time requirements for analytic derivatives (whether they are generated automatically or not) involves a comparison with finite differences. Consequently, we studied the efficiency of ADIFOR-generated derivatives on both the RS6000 and the SGI platform by comparing the computation time of the ADIFOR-generated code to the computation time of a simple forward difference scheme, using the .001% step size from our accuracy studies. We chose this particular differencing scheme for two reasons:

1. Our simple scheme is one of the more time efficient differencing schemes. It is faster, for example, than central differences or an adaptive step size method.

2. The engineers at Sandia used a similar scheme [?].

Furthermore, on the RS6000 platform we investigated two different variations of ADIFOR exception handling: the report-once mode, and the performance mode. These variations differ in their treatment of the singular points of Fortran intrinsic functions. Report-once mode checks the arguments of the intrinsic functions, and reports if any of the arguments are singular values. This kind of checking has moderate overhead, but is indispensable for diagnosing certain kinds of numerical errors. Performance mode, on the other hand, foregoes this checking process in return for improved performance. More details on report-once mode and performance mode can be found in the ADIFOR manual [?].

Our usual practice is to develop and test ADIFOR code with report-once mode. Once we are satisfied that it does not have singularity problems, we reprocess the code using performance mode. We adhered to this practice for our tests of taco2d on the RS6000. For the SGI platform, we used only performance mode, since the report-once mode tests on the RS6000 verified that the cases under consideration had no singular value problems. For all cases studied, we took the average running time of five separate executions to generate the timing values.

On the RS6000 platform, our simple forward difference (FD) scheme took 1487.64 seconds to compute 5 derivative values for 2424 nodes. By comparison, we found that report-once mode delivered the same output in 1073.87 seconds (72.2% of FD). The performance mode derivatives reduced the time requirements to 776.16 seconds (52.2% of FD). For the SGI platform, the FD scheme required 2160.72 seconds to complete the derivative calculations, whereas the ADIFOR-generated code (using performance mode only) required only 1855.39 (85.8% of FD). We believe that the difference in comparable methods between the two platforms is due to the difference in I/O. In all cases, though, the time requirements of ADIFOR-generated derivatives were superior to finite difference time requirements. Furthermore, as we noted earlier, we used a relatively fast difference scheme. Consequently, we would see an even more favorable comparison with a more sophisticated scheme (like central differences, or adaptive step). Our results are summarized in Figure 3.

Configuration	**AD Time** (sec)	**simple FD Time** (sec)	**AD/FD** (%)
RS6000/Report-once	1073.872	1487.64	72.2
RS6000/Performance	776.163	1487.64	52.2
SGI/Performance	1855.39	2160.72	85.8

Figure 3 Efficiency of ADIFOR-generated derivatives

6 Space Requirements

We analyzed the space requirements for taco2d's ADIFOR-generated derivatives in terms of two parameters: the ADIFOR independent variable upper bound (**pmax**) and the taco2d global node array size (**maxa**). The space requirements for all ADIFOR-generated derivative code depend on pmax, since all derivative quantities computed by the program must have one component for every independent variable. Moreover, if a taco2d problem exceeds the pmax parameter, pmax must be changed and the derivative code recompiled. For taco2d, the independent variables were the heating element parameters.

The dependence of the derivative space requirements on the maxa parameter is a consequence of the original memory management scheme used by taco2d. The original taco2d declared a global array for finite element node values. The size of this array (maxa) was designed to be as large as possible (taking up most of the available machine memory). The ADIFOR-generated code, however, needed space for not only node values, but derivatives of node values as well. To accommodate this extra space requirement, we reduced asize by a factor of (pmax+1). The ADIFOR-generated derivative code declared a derivative matrix of size (maxa * pmax) to hold the derivative values. In other words, to accomodate the extra space needed for derivatives, the largest possible number of nodes was reduced by a factor of (pmax+1). For the taco2d version we processed, the original maxa was 6,000,000. Thus, we reduced maxa to 1,000,000, and ADIFOR automatically sized the derivative values array to 5,000,000 values. For the test cases we studied, the number of node values needed was much lower than the maximum. The largest case (2424 nodes, 5 heating elements) required 201,500 node values, and 1,007,500 derivative values.

7 Summary and Conclusions

Our study of taco2d indicated that ADIFOR-generated derivatives yielded accurate derivatives at a fraction of the time requirements of finite difference approximations, and space requirements proportional to the number of heating elements. While we were satisfied with both the accuracy and efficiency of the ADIFOR-generated derivatives for taco2d, we also noted that we might be able to produce derivative values more efficiently if we use a technique called *simplified recurrence differentiation* [?]. This technique saves work by not computing derivative values during the early iterations of an iterative solver. Simplified recurrence differentiation has yielded efficiency improvements on other types of codes (our own experience has been with computational fluid dynamics), so we would expect efficiency improvement in taco2d as well.