A Distributed Application Server for Automatic Differentiation

Boyana Norris and Paul D. Hovland
Mathematics and Computer Science Division
Argonne National Laboratory
9700 S. Cass Avenue, Argonne, IL 60439-4844
norris@mcs.anl.gov, hovland@mcs.anl.gov

Abstract

The ADIC Application Server brings the accuracy and efficiency of automatic differentiation to the World Wide Web. Users of the ADIC Application Server can upload source code written in ANSI-C, manage remote files, differentiate selected functions, and download code augmented with derivative computations. Using a simple driver and linking to the appropriate libraries, the user can compile and run the differentiated code locally. We discuss the unique requirements for an automatic differentiation application server and describe the implementation of the ADIC Application Server.

Keywords : *derivatives, sensitivities, automatic differentiation, ADIC, application server, source transformation*

1. Introduction

Derivatives play an important role in a variety of scientific computing applications, including optimization, solution of nonlinear equations, sensitivity analysis, and nonlinear inverse problems. Automatic, or algorithmic, differentiation (AD) technology provides a mechanism for augmenting computer programs with statements for computing derivatives [11, 12]. In general, given a code C that computes a function $f : x \in \mathbf{R^n} \mapsto \mathbf{y} \in \mathbf{R^m}$ with n inputs and m outputs, an AD tool produces code C' that computes $f' = \partial y / \partial x$, or the derivatives of some of the outputs y with respect to some of the inputs x. In order to produce derivative computations automatically, AD tools systematically apply the chain rule of differential calculus at the elementary operator level.

Compared with other methods, AD offers a number of advantages. The performance of AD-generated code usually exceeds that of divided differences and often rivals that of code developed by hand. Unlike divided difference approximations, derivatives computed via AD exhibit no truncation error. AD also eliminates the time spent developing and debugging derivative code by hand or experimenting with step sizes for finite difference approximations.

To facilitate discussion, we define terms that are commonly used in the context of automatic differentiation:

- *Independent variables* are program input variables with respect to which derivatives are desired.
- *Dependent variables* are program output variables whose derivatives with respect to the independent variables are to be computed.
- A *derivative object* is used to store derivative information, such as a vector of partial derivatives of a variable y with respect to a variable vector x.
- An *active variable* is any program variable with an associated derivative object. Ideally, only variables that may affect a dependent variable and are dependent on an independent variable would be designated as active. A more conservative approach is to consider all floating-point variables active.
- A *driver* routine is used for initializing derivative objects, specifying independent and/or dependent variables, and extracting the computed derivatives.

2. Our approach to AD

ADIC is a source transformation tool for the automatic differentiation of ANSI C code [4, 5]. Source transformation AD tools extend the notion of a compiler by altering the functionality of the original program, augmenting it with derivative computations. Given a set of C source files, ADIC produces a new set of source code files augmented with derivative computations. Some limited C++ support is also available.

ADIC features a design that allows easy expansion of its functionality through plug-in modules. A module specified

at run time interacts with the rest of the system via machine- and language-independent file interfaces. Language independence is achieved through the use of an intermediate representation, known as the AIF (Automatic differentiation Interface Form) [1, 16]. The AIF is designed to abstract AD-relevant information from the more general language features. Although most modules target derivative computation by exploiting the chain rule, modules can be written to perform any language-independent transformation. Each module usually has a set of associated run-time libraries, which must be linked with the differentiated code.

Other AD tools, such as ADOL-C [13], take a different approach and compute derivatives via operator overloading, rather than source transformation. This approach overloads the basic arithmetic operators and intrinsic function calls with special routines that propagate derivatives in addition to performing the original operation [7]. The definition of the class containing derivative computations is hidden from the user, the user's code is not affected by changes in the implementation strategy of the tool, and there is full access to run-time information [3]. However, in order to avoid the generation of temporaries, special care must be taken by using expression templates or a similar mechanism (which may be difficult to implement). Tools based on operator overloading are also more sensitive to the choice of compiler and compiler flags. Most significant for our purposes here is the fact that an AD application server based on operator overloading is not possible, since in this implementation approach the compiler handles the generation of object code for derivative computations.

2.1. Example

In this section we present a small example of using ADIC to augment a piece of code with derivative computations. ADIC can be applied to ANSI-C code of arbitrary complexity, but we limit the size of this example for clarity.

The function in Figure 1 takes five double parameters and returns a double value. This function might be called by some other functions that together form the source processed by ADIC.

We apply ADIC to the code in Figure 1 to produce the code augmented with first-derivative computations shown in Figure 2. We have reformatted the code slightly, expanded some macros, and added some comments for better readability.

A *float object* contains a floating-point value that resides at a physical memory address. In C, float objects are created either explicitly (variable declarations, dynamic memory allocation) or implicitly (return values or type casting). Automatic differentiation of C code works by associating a unique derivative object with each float object in a computation. Within each assignment statement

```
double f (double a, double b, double c,
          double d, double z) {
   double y = 0.0;
   if ((y = a * b * c * d) < 0 && z < 0)
      y *= z;
   return y;
}
```

Figure 1. A simple function to be differentiated.

$y = \text{func}(x_1, x_2, \ldots, x_n)$, where func is an expression involving n variables, the partial derivatives $\partial y/\partial x_i$, or adjoints, are computed. The intermediate computations of the expression are saved in temporaries, and then the chain rule is locally applied. The resulting quantities are then used to accumulate the gradient ∇y. In this example, we have added a comment at the top of the code in Figure 2 showing the definition of the derivative object type. Each gradient ∇y is stored as an array of doubles.

The derivative code together with a driver is compiled and linked to ADIC support libraries to produce the final program. A driver for the example code is shown in Figure 3. In many cases, the user can slightly modify an existing routine, for example, the main function, in order to obtain a driver. There are two ways of creating a driver: writing it from scratch or using ADIC to do part of the work. The code in Figure 3 shows the first approach. The driver performs initializations, sets the independent variables, calls the differentiated function, and extracts the results.

3. Objective

The ADIC Application Server aims to provide an easy-to-use, highly accessible interface to ADIC and, potentially, other AD tools. Our goals include developing and implementing mechanisms for remote file management, fast response to user requests, scheduling of requests in a distributed environment, and assistance with tasks the user must perform after downloading differentiated files from the server.

A typical user session with the server comprises the following steps:

- The new user registers and requests an account. A directory for remote storage of user files is created. This step is required only for new users. Existing users may be required to log in if a certain amount of time has elapsed since their last login.
- The user uploads files to his or her account.

```
/* Disclaimer and copyright notice */
#include "ad_deriv.h"
/* ad_deriv.h includes the following definition of DERIV_TYPE:
   typedef struct {
        double value;
        double grad[ad_GRAD_MAX];
   } DERIV_TYPE;
*/
void  ad_f(DERIV_TYPE *ad_var_ret, DERIV_TYPE a, DERIV_TYPE b,
   DERIV_TYPE c, DERIV_TYPE d, DERIV_TYPE z) {
  /* Return value is stored in ad_var_ret */
  DERIV_TYPE  y;
  double  ad_loc_0, ad_loc_1, ad_loc_2;
  double  ad_adj_0, ad_adj_1, ad_adj_2, ad_adj_3;

  {                                          /* Clear the gradient of y */
    int _ad_AD_ctr; double *gz = y.grad;
    for (_ad_AD_ctr = 0 ; _ad_AD_ctr < ad_grad_size ; _ad_AD_ctr++)
      gz[_ad_AD_ctr] = 0.0;
  }
  y.value  = 0.0;                            /* double y = 0.0; */
  /* Compute adjoints corresponding to a * b * c * d */
  ad_loc_0 = a.value  * b.value; ad_loc_1 = ad_loc_0 * c.value;
  ad_loc_2 = ad_loc_1 * d.value; ad_adj_0 = ad_loc_0 * d.value;
  ad_adj_1 = c.value  * d.value;
  ad_adj_2 = a.value  * ad_adj_1;
  ad_adj_3 = b.value  * ad_adj_1;
  {                                          /* Accumulate the gradient of y */
    int _ad_AD_ctr;
    double *gz = y.grad, *ga = a.grad, *gb = b.grad, *gc = c.grad, *gd = d.grad;
    for (_ad_AD_ctr = 0; _ad_AD_ctr < ad_grad_size; _ad_AD_ctr++)
      gz[_ad_AD_ctr] = + ad_adj_3 * ga[_ad_AD_ctr] + ad_adj_2 * gb[_ad_AD_ctr]
               + ad_adj_0 * gc[_ad_AD_ctr] + ad_loc_1 * gd[_ad_AD_ctr];
  }
  y.value = ad_loc_2;
  if (y.value  < 0 && z.value  < 0) {        /* if (y < 0 && z < 0) */
    ad_loc_0 = y.value * z.value;            /* y *= z; */
    {                                        /* Accumulate the gradient of y */
      int _ad_AD_ctr; double *gz = y.grad, *ga = y.grad, *gb = z.grad;
      for (_ad_AD_ctr = 0; _ad_AD_ctr < ad_grad_size; _ad_AD_ctr++)
        gz[_ad_AD_ctr] =  + z.value * ga[_ad_AD_ctr] +  y.value * gb[_ad_AD_ctr];
    }
    y.value  = ad_loc_0;
  }
  {                          /* Accumulate the gradient of the return value */
    int _ad_AD_ctr; double *gz = (*ad_var_ret).grad, *gx = y.grad;
    for (_ad_AD_ctr = 0; _ad_AD_ctr < ad_grad_size; _ad_AD_ctr++)
      gz[_ad_AD_ctr] = gx[_ad_AD_ctr];
  }
  (*ad_var_ret).value = y.value;
  return;
}
```

Figure 2. ADIC-generated code for function f.

```
#include <stdio.h>

#define MAXLEN 4
double f(double, double, double, double, double);
int main() {
  int            i, n;
  InactiveDouble grad[MAXLEN];
  InactiveDouble tmp, val;
  double         x[MAXLEN], y, z = -1.0;

  /* Initialize */
  AD_Init(ad_GRAD_MAX);

  /* Read in values*/
  for (i = 0; i < MAXLEN; i++) {
    scanf("%lf", &tmp);
    x[i]= tmp;
  }

  /* Set independent variables */
  AD_SetIndepArray(x, MAXLEN);
  AD_SetIndepDone();

  /* Invoke the function */
  y = f(x[0], x[1], x[2], x[3], z);

  /* Extract the result */
  AD_ExtractVal(val, y);
  AD_ExtractGrad(grad, y);

  /* Print the result */
  printf ("value = %le\n", val);
  for (i = 0; i < n; i++) {
    printf ("%le\n", grad[i]);   /* Print partials */
  }
}
```

Figure 3. Driver for ADIC-generated code.

- The user selects files among the list of uploaded files, optionally specifies ADIC options, and directs the server to apply ADIC to them.
- The user views or downloads the ADIC-generated files containing derivative computations.
- At any time during the interactive session, the user can manipulate the files in the remote directory (delete, copy, rename, download).

In any given session, the user can submit multiple requests for differentiating different sets of files. Before incorporating the differentiated code into an application, the user must follow the simple steps for downloading and unpacking the ADIC libraries, which are available for a number of Unix platforms. No additional software is required in order to use the code produced by the ADIC Application Server.

4. Related work

The use of the Internet for scientific computations has been increasing rapidly in recent years. Network-enabled servers, such as NEOS [8] and NetSolve [6], aim to provide access to hardware and software computational resources through a variety of interfaces.

Various projects have considered the extension of this paradigm to automatic differentiation. Hovland and Carle developed a remote automatic differentiation tool (RADtool) that provided limited access to ADIFOR [2] via the Web. Roh developed a prototype of the ADIC Application Server, noting the importance of file management capabilities [15]. Recently, several European researchers have begun discussing the development of a Network Enabled Sensitivities System (NESSy) [14].

Research issues in the design of typical scientific application servers include fault tolerance, load balancing, high-performance computational servers, user interfaces, and network-based computing. AD application servers have somewhat different requirements, with a greater emphasis on account creation, file management, and persistence of state.

4.1. Other scientific application servers

Several principal approaches to network-enabled computing exist. In one approach, the user's program and data are transferred from the user's machine to a server, which then runs the code on the data and transfers back the result. Another approach is to download the program from the server to the user's machine, where it operates on the user's data and generates the result locally. Finally, in a remote computing environment, only the user's data travels to the server, where programs based on numerical libraries operate on it and then return the result to the user. NetSolve uses the third approach, whereas NEOS employs a hybrid of the first and third approaches.

The Grid Portal Toolkit (GridPort) [10, 17] is a collection of services, scripts, and tools that allow developers to connect Web-based interfaces to distributed computational resources. GridPort was originally developed for the informational NPACI HotPage, which allows NPACI users to access resources through a Web interface. This functionality has been extended to enable users to run codes, access data, and communicate with NPACI's Globus-ready systems. These facilities more closely match the requirements for an AD application server.

4.2. AD in the NEOS Server

In addition to providing the basic functionality behind the ADIC Application Server, AD technologies have been used successfully as part of other network-enabled computational servers [9]. The NEOS Server is an Internet-based client-server application that provides access to a number of optimization solvers, eliminating the need for downloading solver software, writing code to call the solver, or computing the derivatives for nonlinear problems. Given an input format and a list of solvers, the NEOS user submits a problem description consisting of the dimensions of the problem, the source code for the function evaluation, bounds and constraints routine (for constrained optimization), and the starting point. NEOS employs automatic differentiation to compute derivatives required by nonlinear solvers. In the case of Fortran user code, the Jacobians or Hessians of nonlinear problems are determined by ADIFOR, and sparsity is handled by the SparsLinC library. For C submissions, the AD tools used are ADOL-C and ADIC.

5. The ADIC Application Server

Like other types of scientific computing server, the ADIC Application Server is concerned with research issues involving user interface design and access to remote hardware and software resources. Unlike conventional computational servers, however, the ADIC Application Server must be concerned also with source transformation issues. The source code supplied by the user is treated as data by the ADIC software. After processing the user-supplied source code remotely, the resulting derivative-augmented source code is downloaded by the user and becomes the new program, to be compiled and executed locally on the user's machine.

Although not usually an important part of scientific servers, the account and file management capabilities of the ADIC Application Server are an essential part of the user

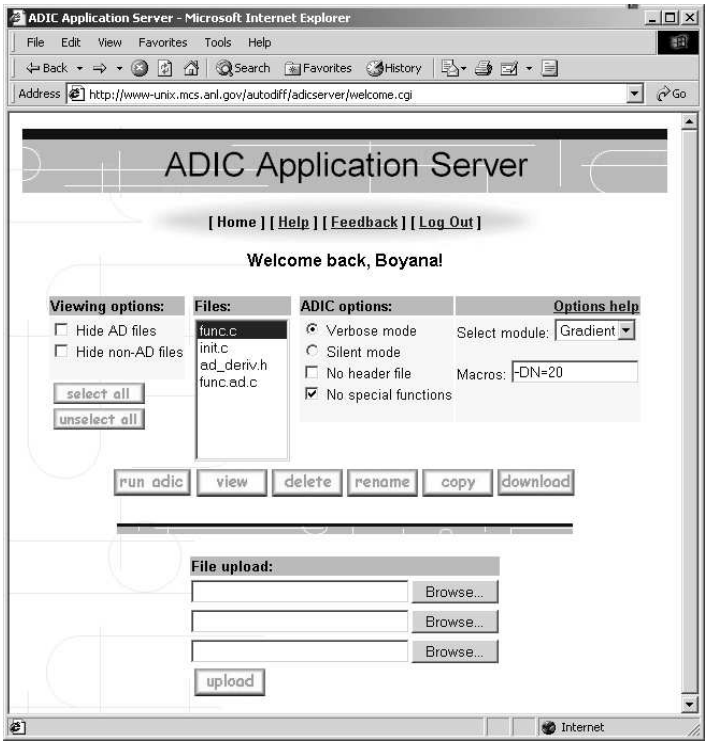

Figure 4. ADIC Application Server main page.

interface. Important issues include account creation and maintenance policies, security, file manipulation options, and persistence of state. In some ways the needs of the ADIC Application Server are more closely related to those of business application service providers (ASPs), including access to server hardware, file management capabilities, and ability to run applications remotely, in this case ADIC.

6. Server implementation

The ADIC Application Server comprises two main components: the Web-based user interface (Figure 4) and the server daemon. The user interface is implemented by using mainly Perl CGI scripts. In order to minimize response time for certain actions, JavaScript is used for some of the file management functionality, as well as for error checking of user input. We use browser cookies to store some user information and eliminate the need for frequent authentication.

The server daemon is separate from the user interface and can run in a distributed environment. At present, the only requirement is that the Web server and the ADIC server daemon reside on a shared file system. When the user selects files and pushes the "Run ADIC" button, a CGI script generates a job file containing user identification and a description of the user's request, including names of source files and ADIC options. The server daemon periodically scans for new job files and processes them in a first-come first-served fashion. The server then assembles the correct call to ADIC and spawns a process to execute the user request. Job files have unique IDs, allowing multiple users to submit requests simultaneously to the same server.

Figure 5 shows the output from a user submission with the options illustrated in Figure 4. The Web page showing the output from the server is dynamically updated as the job execution progresses.

7. Future work

The ADIC Application Server is under active development. In particular, extensions are under way or planned for the near future in the following four areas:

- *File management*. We are working on perfecting and extending the file management capabilities, including adding restrictions to the type and size of files that users can upload. One possibility is to add WebDAV support to the server, allowing third-party tools (e.g., Microsoft Explorer in Windows systems) to be used to transfer files between the server and the user's local

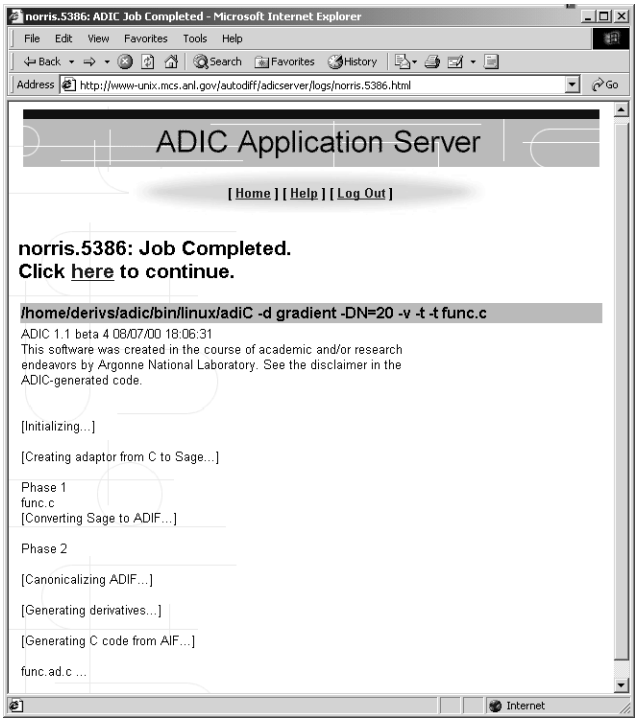

Figure 5. ADIC Application Server job output.

machine.

- *Scheduler*. At present, the ADIC Application Server uses a simple FCFS scheduling strategy. Multiple servers can be started on different machines, but there is no distributed scheduling strategy in place yet. We will implement a scheduler that will effectively utilize a number of networked computers, aiming to maximize response time, while performing some load balancing of requests.

- *Automated driver generation*. To use the code produced by ADIC, the user must write a driver to perform initialization, call the differentiated routine(s), and extract the gradient. This task can be made easier by supplying the user with a driver prototype routine, which may require only minor adjustments before it can be incorporated into the user's application.

- *ADIFOR Application Server*. We have begun development of the ADIFOR Application Server. Most of the implementation of the ADIC Application Server can be reused, because of the many shared features of the two servers. There is also potential for developing servers for similar types of transformation systems, including Java bytecode optimizers.

Less essential features we are planning to implement include enabling the Web server to determine whether a server daemon and/or scheduler is running somewhere on the network, providing additional formats for archives of files the user wishes to download, and providing locking mechanisms so that multiple requests operating on the same user files can be served.

8. Conclusions

The need for accurate and fast derivatives for models implemented as computer code is ubiquitous in computational science. Automatic differentiation provides a mechanism for computing these derivatives accurately with minimum programming effort. In this article, we introduced the ADIC Application Server, which provides a highly accessible, easy-to-use interface to AD technology. The server eliminates the need to install the ADIC software, configure it correctly, and learn how to use it. The URL for the ADIC Application Server is www.mcs.anl.gov/autodiff/adicserver.

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

We thank Shannon Melfi of the University of Illinois at Urbana-Champaign for her substantial contribution to the initial design and implementation of the ADIC Application Server and Lucas Roh, formerly of Argonne National Laboratory, for his ideas about server features. We also thank Gail Pieper for proofreading an early draft of this manuscript.

References

[1] AIF developer's page. www-unix.mcs.anl.gov/autodiff/AIF.

[2] C. Bischof, A. Carle, P. Khademi, and A. Mauer. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering*, 3(3):18–32, 1996.

[3] C. Bischof and A. Griewank. Tools for the automatic differentiation of computer programs. Technical Report Preprint ANL/MCS-P608-0896, Argonne National Laboratory, January 1997.

[4] C. Bischof and L. Roh. ADIC user guide. Technical Memorandum ANL/MCS-TM-225, Argonne National Laboratory, 1997.

[5] C. Bischof, L. Roh, and A. Mauer. ADIC — An extensible automatic differentiation tool for ANSI-C. *Software–Practice and Experience*, 27(12):1427–1456, 1997.

[6] H. Casanova and J. Dongarra. Applying NetSolve's network-enabled server. *IEEE Computational Science and Engineering*, 5(3):57–67, July–September 1998.

[7] G. F. Corliss and A. Griewank. Operator overloading as an enabling technology for automatic differentiation. Technical Report Preprint ANL/MCS-P358-0493, Argonne National Laboratory, May 1993.

[8] J. Czyzyk, M. P. Mesnier, and J. J. Moré. The NEOS server. *IEEE Computational Science and Engineering*, 5(3):68–74, July–September 1998. Preprint ANL/MCS-P615-0996, Mathematics and Computer Science Division, Argonne National Laboratory.

[9] M. C. Ferris, M. Mesnier, and J. J. Moré. NEOS and Condor: Solving optimization problems over the Internet. Preprint ANL/MCS-P708-0398, MCS Division, Argonne National Laboratory, Argonne, Ill., 1998.

[10] Gridport website. www.gridport.npaci.edu.

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

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

[13] A. Griewank, D. Juedes, H. Mitev, J. Utke, O. Vogel, and A. Walther. ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM TOMS*, 22(2):131–167, June 1996.

[14] U. Naumann. Automated analysis and enhancement of application code. Personal communication, 1999.

[15] L. Roh. Personal communication, 1997.

[16] L. Roh, P. D. Hovland, J. Abate, and C. Bischof. AIF component system.

[17] M. Thomas, S. Mock, and J. Boisseau. Development of toolkits for computational science portals: the NPACI HotPage. www.gridport.npaci.edu/pubs/publications/HPDC-9.pdf.