Automatic Differentiation of a Parallel Molecular Dynamics Application*

P. Hovland† C. Bischof‡ L. Roh‡

December 31, 1996

Abstract

The ADIC and ADIFOR automatic differentiation tools have proven useful for obtaining the derivatives needed in many scientific applications written in Fortran 77 or ANSI C. But many new scientific programs are written for or ported to parallel platforms to achieve maximal performance. We provide an overview of our approach to the complex task of applying automatic differentiation techniques to parallel programming environments, especially as applied to a parallel molecular dynamics application written in C++ with PVM message passing.

1 Introduction

There are many areas of computational science in which it is necessary or desirable to compute derivatives. One important domain is computational molecular dynamics, which can use derivatives in a number of different ways. One common use of derivatives is in the computation of forces, which are the derivatives of energies with respect to position. Molecular dynamics simulations, like other computer models that attempt to simulate some physical phenomenon, can benefit from sensitivity analysis, wherein we compute the derivatives of the model function with respect to various parameters in order to determine the sensitivity of the model to changes in these parameters. Finally, higher order derivatives can improve the accuracy of a numerical method, such as a differential equation solver, enabling, for example, longer time steps.

When computational scientists need derivatives, they usually obtain them through divided difference approximations or by hand-coding derivative code. The former approach suffers from the fact that the values being computed are *approximations*, not true derivatives. If the step-size used for the divided differences is too large or too small, the approximations can be grossly inaccurate. Furthermore, there is no way to assess the accuracy of the approximation. Developing a derivative code by hand provides efficient, accurate derivatives. However, hand-coding can be tedious, error-prone, and extremely time-consuming. Derivative code can also be generated using a symbolic manipulator such

*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, and the National Science Foundation, through the Center for Research on Parallel Computation, under Cooperative Agreement No. CCR-9120008.

†Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 W. Springfield Ave., Urbana, IL 61801, hovland@uiuc.edu.

‡Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439, {bischof,roh}@mcs.anl.gov.

as Mathematica. However, traditional symbolic manipulation is memory intensive, and is often not feasible for large programs or programs with many loops and branches. An alternative to all of these approaches is the technique of automatic differentiation (AD). AD produces derivative code that computes accurate (within the limits of finite precision arithmetic) derivatives, and can be applied to arbitrarily complex programs with minimal effort on the part of the programmer.

In recent years, tools have been developed that enable AD to be applied to programs written in Fortran, C, Ada, and other languages[1] [1, 2, 3, 7, 14]. However, there remains a need for tools for applying AD to parallel languages and programming systems. Without such tools, a substantial amount of hand-coding must be done in order to develop programs for computing derivatives. We are developing tools and techniques for the automatic differentiation of parallel programs written in explicitly parallel languages, like Fortran M [4], or using parallel extensions, such as MPI and PVM [5, 8]. We are applying some of these techniques to NAMD, a parallel molecular dynamics application written in C++ with PVM message passing [12].

This paper describes the automatic differentiation of NAMD. We begin with a brief introduction to automatic differentiation in Section 2. Section 3 briefly describes NAMD and explains which derivatives are being computed and how they will be used. Section 4 describes how AD technology is being applied to this program. We conclude with a synopsis of our research and a description of planned future work.

2 Automatic Differentiation

Automatic differentiation can be used to transform a program for computing some mathematical function into a new program capable of computing not only the function, but also the derivatives of that function [2, 3]. Automatic differentiation relies upon the fact that all programs, no matter how complicated, use a limited set of elementary operations and functions, as defined by the language. The function computed by the program is simply the composition of these elementary functions. Thus, we can compute the partial derivatives of the elementary functions using formulas obtained via table lookup, then compute the overall derivatives using the chain rule. This process can be completely automated, and is thus termed *automatic differentiation* [6].

For example, consider the following program to compute the function $y = f(x)$, where $f(x) = (\sin(x)\sqrt{x})/x$.

```
A = sin(X)
B = sqrt(X)
C = A * B
Y = C/X
```

Using automatic differentiation, we can generate code to compute y and dy/dx.

```
A = sin(X)
dAdX = cos(X)                  ! table lookup
B = sqrt(X)
dBdX = 1/(2*B)                 ! table lookup
C = A * B
```

[1] See `http://www.mcs.anl.gov/Projects/autodiff/AD_Tools/` for a survey of AD tools.

```
dCdA = B                          ! table lookup
dCdB = A                          ! table lookup
dCdX = dCdA*dAdX + dCdB*dBdX      ! chain rule
Y = C/X
dYdC = 1/X                        ! table lookup
dYdX = dYdC*dCdX - C/(X*X)        ! chain rule/table lookup
```

This is an example of the so-called forward mode of automatic differentiation. In this mode, we propagate derivatives with respect to the independent variable(s) (in this case x). These *derivative vectors* (in general, there can be more than one independent variable) are often denoted ∇y or g_y. In the event that y itself is a vector, we may refer to ∇y as a *derivative matrix*. While this example is very simple, automatic differentiation can be applied to complex programs of arbitrary length. The ADIC tool has processed programs of over 10,000 lines and the ADIFOR tool has been applied to programs of over 100,000 lines [2].

3 NAMD

NAMD is a parallel, object-oriented molecular dynamics program designed for high performance molecular dynamics simulations of large biomolecular systems [12]. Important features include scalable parallelism, an efficient implementation of full electrostatics, modifiability, portability, and compatibility with X-PLOR (a program for determining three-dimensional structures from crystallographic diffraction or NMR data). Full electrostatics are computed using the Distributed Parallel Multipole Tree Algorithm (DPMTA) developed at Duke University [13]. NAMD is written in C++, using an object-oriented and highly modular design. This design facilitates modification of algorithms and techniques. Communication in NAMD is accomplished via PVM, making it portable across a wide range of computing platforms. The input and output file formats used by NAMD are identical to those used by the program X-PLOR, thus integrating the two tools and the accompanying visualization facilities.

To reduce the cost of the evaluation of long-range electrostatic forces, a multiple time step scheme is combined with the DPMTA method. All but the long-range electrostatics interactions are calculated during every time step. The longer range interactions are computed only every k steps. For appropriate values of k, the error due to holding the forces constant for a few time steps is small compared to the errors incurred from using a finite timestep.

The developers of NAMD hope to improve the integrator using an approach that requires Hessian-vector products [11]. The Hessian required corresponds to the derivatives of forces with respect to position. The availability of these correction terms is expected to increase the smallest time step by a factor of nearly three. Another proposed method would increase performance by decreasing the frequency of long-range force evaluations. It, too, requires the derivatives of the forces. The former method is of interest to researchers because of its improved accuracy, while the latter method would enable the simulation of larger molecules in less time.

While it is possible to develop code to compute the derivatives of the forces by hand, there are several reasons for preferring automatic differentiation. Writing derivative code by hand can be very difficult, and may require a great deal of time for development and debugging. In contrast, automatic differentiation allows us to develop correct and efficient derivative code with very little human effort. In addition, NAMD's modular design

encourages the use of new algorithms to compute forces, or the incorporation of forces that had previously been neglected. Again, automatic differentiation allows us to create derivative code for these new force implementations with very little effort. Finally, the derivative matrices being computed are sparse. Tools such as ADIC and ADIFOR provide support for automatic exploitation of sparsity, without prior knowledge of the sparsity structure of the derivative matrices [2, 3].

4 AD of NAMD

NAMD is written in C++ with PVM message passing. A port to MPI is planned for the future. Thus, in order to apply AD to NAMD, we must be able to apply AD to programs written in C++ and parallel programs written using PVM (MPI).

4.1 AD of C++

ADIC (Automatic Differentiation of C) is an extensible AD tool that produces code for computing first and/or second derivatives. The second derivative capabilities are currently in the prototype stage. ADIC uses a source-to-source program transformation technique to produce the derivative code and provides the following important features: *robustness*, in the form of full support for ANSI C; *flexibility*, provided by simple command-line flags and control files; *portability*, through a careful design that ensures that the code generated by ADIC is portable across different platforms and compilers; and *extensibility*, through the use of a language-independent component architecture.

To accommodate C++, ADIC has been extended to support important language features, such as classes and methods. ADIC also takes advantage of C++ features when generating the derivative code; e.g., new variables may be declared anywhere within a block. Certain aspects of C++ are not yet supported. One unsupported feature is the use of default arguments. Iostream operations are also not supported.

4.2 AD of Parallel Programs

Automatic differentiation requires that we associate a derivative vector (or matrix) with each variable. In Fortran, this can be accomplished via a naming scheme, such as using the variable name `g_var` for the derivative vector associated with the variable `var`. In C and C++, this is not possible, because of the aliasing induced by pointers. Instead, the association is accomplished either by creating a structure containing the variable and its associated derivative vector or by applying a hash function to the address of the variable.

In a parallel programming environment with message passing, we must preserve the association between variables and their derivative vectors when data is sent via a message. One approach is to pack the variable and its gradient vector next in the same message. The packing and unpacking may incur some overhead, but guarantees the correct association between a variable and its derivative vector. This method is illustrated in Figure 1. Another option is to send two messages, one containing the variable and one containing the derivative vector. In this case, we must use tags and source identifiers to ensure that the association is preserved, and additional latency overhead may be incurred if we cannot use computation to mask the communication time. This method is illustrated in Figure 2. Note that this implementation assumes that messages from the same source arrive in order. For parallel programming environments where this is not necessarily true, a more sophisticated tagging scheme is needed.

To study the tradeoff between latency and packing overhead, we conducted some simple

sender:

```
pack(x,msg)
pack(g_x,msg)
send(msg,dest,tag)
```

receiver:

```
recv(msg,source,tag,info)
x = unpack(msg)
g_x = unpack(msg)
```

FIG. 1. *Pseudocode for the packing method*

sender:

```
send(x,dest,tag)
send(g_x,dest,tag)
```

receiver:

```
recv(x,ANY_SOURCE,ANY_TAG,info)
source = info.source
tag = info.tag
recv(g_x,source,tag,info)
```

FIG. 2. *Pseudocode for the separate messages method*

experiments. Using MPI on a network of SPARCstations and on an IBM SP, we measured the time to pack and unpack vectors of varying lengths into a message buffer. We also measured the time to send messages of varying lengths. Using this data, we used a least squares fit to find the length-dependent and -independent components of the cost. Table 1 summarizes our results. We use α to denote the latency, β to denote the bandwidth, α_p to denote the length-independent component of packing and unpacking a vector, and β_p to denote the number of bytes that can be packed and unpacked per second. Thus, the time to send a vector of length n is approximately $\alpha + n/\beta$, while the time to pack and unpack the same vector is $\alpha_p + n/\beta_p$. The value $\gamma = (\alpha - \alpha_p)\beta_p$ provides a measure of the minimum number of bytes that must be packed in order to exceed the latency. Therefore, on these systems, packing variables and their associated derivative matrices together is preferable, as long as their combined size does not exceed about 200–600 thousand bytes. This limit is not so large as it may seem. A vector of 250 double precision values, with an associated derivative matrix of size 250×250, requires over 500 thousand bytes of storage. Nonetheless, for typical problems on typical systems, packing variables and derivative matrices together seems preferable to separate messages.

In general, there are other issues that may need to be addressed in applying AD to parallel programs. In addition to preserving variable-derivative matrix associations, we should correctly differentiate reduction operations and attempt to avoid unnecessary derivative computations. These issues are discussed elsewhere [9, 10].

4.3 AD of NAMD

Due to some of the limitations mentioned in Section 4.1, we were unable to process NAMD in its entirety. Instead, we chose to process the class responsible for computing the bonded

System	α (s)	β (B/s)	α_p (s)	β_p (B/s)	γ (B)
SPARCstations	9.22×10^{-3}	1.35×10^{1}	8.18×10^{-4}	2.14×10^{7}	1.96×10^{5}
IBM SP	8.91×10^{-3}	3.76×10^{6}	2.94×10^{-5}	7.54×10^{7}	6.69×10^{5}

TABLE 1

Parameters for communication and packing times (s = seconds, B = bytes)

forces (`BondForce`) separately. This class uses the `Vector` class, so we needed to process this class, too. The `Vector` class uses output streams and default parameters, so we were forced to make some modifications before processing it with ADIC. After the necessary changes had been made, the `BondForce` and `Vector` classes were processed with ADIC, resulting in two new classes, `ad_BondForce` and `ad_Vector`, that were integrated with the unprocessed portion of NAMD.

In order to integrate the new classes into NAMD, we needed to write methods for sending and receiving `ad_Vector` objects, which contain 3 objects of type `DERIV_TYPE`. A `DERIV_TYPE` object may be viewed as containing a variable (denoted `DERIV_VAL`) and its associated derivative vector (denoted `DERIV_grad`). Because NAMD already packs multiple objects into a single message, we chose to pack variables and derivatives together in order to preserve the association between the two objects. The following method is used to pack a variable and its associated derivative vector into a message.

```
Message& put(int n, DERIV_TYPE *d,int copy=TRUE, int delstor=FALSE) {
  int i;

  for (i=0;i<n;i++){
  /* Add the value of the variable d[i] to the message */
  putmsg((void *)(&DERIV_VAL(d[i])), DOUBLE, 0, sizeof(double), copy,
         delstor);
  /* Add the gradient vector of variable d[i] to the message */
  putmsg((void *)(&DERIV_grad(d[i])), DOUBLE, ad_GRAD_MAX,
         sizeof(double), copy, delstor);
  }
  return *this;
}
```

5 Conclusions

We have applied automatic differentiation to the class in NAMD responsible for computing the bonded forces. This required modifying the ADIC tool so that it could handle C++, adding support for the communication of variables and their associated derivative vectors to NAMD, and incorporating the AD-generated class into NAMD. In the future, we intend to apply AD to additional classes, such as that responsible for computing angle forces. After the correctness of the derivatives has been verified, they can be used to improve the integration scheme. We will also continue our work on the development of tools for the automatic differentiation of parallel programs. We have addressed several of the important issues in this task, including maintaining the association between derivative vectors and variables, improving efficiency through intertask dependence analysis, properly differentiating reduction operations, and utilizing the added potential for parallelism created by the automatic differentiation process [9, 10]. Based on the experience gained from the development of prototype AD tools for Fortran M and Fortran with MPI message passing as well as the application of AD to NAMD, we plan to build a tool for the automatic differentiation of C/C++ with PVM/MPI message passing.

Acknowledgements

We thank Bob Skeel, Klaus Schulten, and all of the Theoretical Biophysics group at the Beckman Institute at the University of Illinois for their assistance in working with NAMD.

We also thank Mike Heath for his comments on an earlier version of this paper.

References

[1] 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.

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

[3] C. Bischof, L. Roh, and A. Mauer, *ADIC — An extensible automatic differentiation tool for ANSI-C*, Preprint ANL/MCS-P626-1196, 1996.

[4] I. Foster, R. Olson, and S. Tuecke, *Programming in Fortran M*, Tech. Rep. ANL–93/26, Rev. 1, Mathematics and Computer Science Division, Argonne National Laboratory, October 1993.

[5] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM - Parallel Virtual Machine: A Users' Guide and Tutorial for Network Parallel Computing*, MIT Press, Cambridge, 1994.

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

[7] A. Griewank, D. Juedes, and J. Utke, *ADOL-C, a package for the automatic differentiation of algorithms written in C/C++*, ACM Transactions on Mathematical Software, 22 (1996), pp. 131–167.

[8] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI – Portable Parallel Programming with the Message Passing Interface*, MIT Press, Cambridge, 1994.

[9] P. Hovland, C. Bischof, and L. Roh, *Automatic differentiation of parallel reduction operations*, Preprint ANL/MCS-P632-1296, Mathematics and Computer Science Division, Argonne National Laboratory, 1996.

[10] P. D. Hovland, *Automatic Differentiation of Parallel Programs*, PhD thesis, University of Illinois at Urbana-Champaign. In preparation.

[11] M. López-Marcos, J. M. Sanz-Serna, and R. D. Skeel, *Explicit symplectic integrators using Hessian–vector products*, SIAM J. Sci. Comput., 18 (1997). To appear.

[12] M. Nelson, W. Humphrey, A. Gursoy, A. Dalke, L. Kale, R. D. Skeel, and K. Schulten, *NAMD - a parallel, object-oriented molecular dynamics program*, Journal of Supercomputing Applications and High Performance Computing. In Press.

[13] W. T. Rankin and J. A. Board Jr., *A portable distributed implementation of the parallel multipole tree algorithm*, in Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing, Los Alamitos, CA, 1995, IEEE Computer Society Press, pp. 17–22.

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