Automatic Differentiation for Message-Passing Parallel Programs[*]

Paul Hovland Christian Bischof
Mathematics and Computer Science Division
Argonne National Laboratory
9700 S. Cass Avenue
Argonne, IL 60439

Abstract

Many applications require the derivatives of functions defined by computer programs. Automatic differentiation (AD) is a means of developing code to compute the derivatives of complicated functions accurately and efficiently, without the difficulties associated with developing correct code by hand. We discuss some of the issues involved in developing automatic differentiation tools for parallel programming environments.

1. Introduction

Derivatives of functions are used in a variety of applications, ranging from optimization to sensitivity analysis of computer models. Automatic differentiation (AD) provides a mechanism for computing the derivatives of a complicated function—expressed in the form of a program—accurately and efficiently, without the difficulty of developing correct code by hand or the potentially exponential time and space required by traditional symbolic manipulation. Many programs are being developed on or ported to parallel computing platforms. Thus, there is a need for automatic differentiation tools for parallel programs.

One popular paradigm for parallel programming is message passing. In this paper, we consider two important issues in the AD of message-passing parallel programs. The first issue is maintaining an association between the data structures used for a variable y and the derivative(s) of that variable with respect to the independent variable(s), often denoted ∇y. The second issue discussed is the differentiation of parallel reduction operations.

[*] 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.

The next section provides a brief introduction to automatic differentiation. Section 3 discusses techniques for preserving the association between y and ∇y. Section 4 examines the problem of applying AD to reduction operations, especially the `product` reduction. Section 5 summarizes the recommended strategies for coping with these problems and concludes with a summary of other issues important to the automatic differentiation of parallel programs.

2. Automatic Differentiation

Complex functions are often expressed as algorithms and computed by using computer programs. AD has proven an effective means of developing code to compute the derivatives of such functions. AD relies upon the fact that all programs, no matter how complicated, use a limited set of elementary operations and functions, as defined by the programming 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* [5].

Consider the code for computing the function $y = f(x)$, where $f(x) = (\sin(x)\sqrt{x})/x$, shown in Figure 1(a). Using AD, we can generate code to compute both y and $\mathrm{d}y/\mathrm{d}x$, as shown in Figure 1(b).

This example uses the so-called forward mode of AD. In this mode, we propagate derivatives with respect to the independent variable(s) (in this case x). When there is more than one independent variable, we propagate *derivative vectors*. We use ∇y or `g_y` to denote the derivative vector associated with variable y. This derivative vector contains the derivatives of y with respect to the independent variables. For example, if a, b, and c are all independent variables, AD might produce the following:

$$\begin{aligned} a &= 1 \\ \nabla a &= [1\ \ 0\ \ 0] \end{aligned}$$

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

(a)

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

(b)

Figure 1. Code for computing a simple function (a) and code for computing its derivatives generated by AD (b).

$$\begin{aligned} b &= 2 \\ \nabla b &= [0\ \ 1\ \ 0] \\ c &= 4 \\ \nabla c &= [0\ \ 0\ \ 1] \\ x &= ab \\ \nabla x &= a\nabla b + b\nabla a \\ y &= c/x \\ \nabla y &= -y/x\nabla x + 1/x\nabla c \end{aligned}$$

A Fortran implementation of these equations appears in Figure 2 (to conserve space, we use ; as a line separator). On exit, ∇y contains $[\partial y/\partial a\ \ \partial y/\partial b\ \ \partial y/\partial c] = [-1\ \ -1\ \ 1]$.

```
a = 1.0; b = 2.0; c = 4.0
g_a(1)=1.; g_a(2)=0.; g_a(3)=0.
g_b(1)=0.; g_b(2)=1.; g_b(3)=0.
g_c(1)=0.; g_c(2)=0.; g_c(3)=1.
x = a*b
do i=1,3
  g_x(i) = a*g_b(i)+b*g_a(i)
enddo
y = c/x
do i=1,3
  g_y(i) = (-y/x)*g_x(i)+g_c(i)/x
enddo
```

Figure 2. Example showing propagation of derivatives using derivative vectors.

Note that each assignment statement ∇var $=$ ... involves the summation of one of more derivative vectors scaled by a (scalar) partial derivative. This "scale-and-add" operation occurs frequently in linear algebra and appears as the *saxpy* (daxpy, etc.) routine in the Basic Linear Algebra Subprograms (BLAS) library. Thus, efficient implementations exist for most computer architectures.

It is possible to associate a derivative vector with each element of a vector w. In this case, the composite object ∇w may be referred to as a *derivative matrix*. In the general case, we use the term *derivative object*.

While the examples given have been very simple, AD can be applied to complex programs of 100,000 lines or more [2]. Also, because it makes no assumptions about program structure, AD is applicable to applications from a wide range of problem domains [1, 2, 3].

3. Preserving Derivative Object Associations

AD 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`. We call this *association by name*. In C and C++, this is not possible because of the aliasing induced by pointers. Instead, the association may be 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. We refer to both strategies as *association by address*.

In ADIFOR [2], an AD tool for Fortran, the derivative vector associated with the variable `x` is named `g_x`. The ADIC tool [3] for AD of ANSI C, on the other hand, currently replaces the declaration

```
float x;
```

with the declaration

```
DERIV_TYPE x;
```

where `DERIV_TYPE` is defined as in Figure 3. Subsequent uses of `x` are replaced with references to `DERIV_VAL(x)` and the derivative vector associated with `x` is referenced using `DERIV_grad(x)`.

```
typedef struct {
  double value;
  double grad[ad_GRAD_MAX];
} DERIV_TYPE;
#define DERIV_VAL(a) ((a).value)
#define DERIV_grad(a) ((a).grad)
```

Figure 3. Definition of `DERIV_TYPE` object.

In a parallel programming environment with message passing, we must preserve the association between variables and their derivative vectors not only during memory allocation and floating-point operations, but also when data are sent via messages. We describe two strategies for maintaining this association.

The variable and its associated derivative object can be communicated using either one or two messages. If both are to be communicated using a single message, they must be packed together in that message. If they are to be communicated via separate messages, an association between these messages must be maintained by using tags, source identifiers, temporal information, and other data. Pseudocode for these two alternatives is provided in Figures 4 and 5.

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)
```

Figure 4. Pseudocode for packing.

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)
```

Figure 5. Pseudocode for separate messages.

3.1. Packing

The packing of a variable and its derivative object may be explicit or implicit, depending upon the features of the parallel programming environment being used. If the language or system provides support for communicating multiple data structures in the same message, for example through multiple arguments or derived datatypes, the packing may be done implicitly. Otherwise, the packing must be done explicitly. Many parallel programming systems include library routines for packing data, such as the MPI [6] routine `MPI_Pack`. Because the variable and the individual elements of the derivative object are of the same type, an AD tool may pack them into a single message simply by copying them into an array of that type, even if the programming environment being used does not offer support for explicit packing of data.

3.2. Separate Messages

If the variable and its derivative object are communicated via separate messages, an association between these messages must be preserved. In a parallel programming environment, such as MPI, that guarantees in-order delivery of messages, it is sufficient to send the derivative object with the same tag as the variable. The receiving process can then identify the derivative object as the next message to arrive from the same source with the same tag. Additional information, such as message length, can be used as further confirmation that the correct message has been received. If, however, the programming environment does not guarantee in-order delivery of messages, two variables sent at nearly the same time may end up “swapping” derivative objects. In this case, it may be necessary to use the tag or even the message itself to include a unique identifier with each message, so that the relationships among messages can be resolved.

3.3. Comparison

The benefit of using packing is that it is simple to maintain the association between a variable and its derivative object. One disadvantage of packing is the overhead of packing and unpacking the data. However, this overhead can be very small, especially if the packing is implicit. Another disadvantage of packing is that it may be necessary to allocate space for the packed data, especially if the packing is explicit. The benefit of separate messages is that there is no packing overhead and no need to allocate additional space. There is, however, the overhead of sending a second message. Also, it can be difficult to preserve the association between messages, especially in a programming environment that does not guarantee in-order message delivery.

3.4. Experimental Results

To study the tradeoff between the overhead of packing and the overhead of sending a second message, we conducted a simple experiment. Using MPI on an ethernet network of various SPARCstations and on an IBM SP at Argonne National Laboratory, 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. The cost of communicating data via a message can be modeled as

$$T_{\text{comm}} = \alpha_{\text{comm}} + \frac{N}{\beta_{\text{comm}}},$$

where α_{comm} is the size-independent startup cost and N/β_{comm} is the size-dependent component, where N is the message size and β_{comm} is the bandwidth. We can model the cost of packing data using a similar equation:

$$T_{\text{pack}} = \alpha_{\text{pack}} + \frac{N}{\beta_{\text{pack}}}.$$

Then, the added cost of packing a message is T_{pack}, while the added cost of sending a second message is α_{comm}. Thus, the cost of packing exceeds that of sending a second message whenever $N > N^* \overset{\text{def}}{=} (\alpha_{\text{comm}} - \alpha_{\text{pack}})\beta_{\text{pack}}$. We estimated the α and β parameters by fitting data gathered with the MPI program to the models using a least squares approximation. These values as well as the crossover value N^* are given in Table 1. From this data, it can be observed that N^* is typically on the order of 10^5. Thus, the overhead of packing will be less than the overhead of a second message whenever the message size is less than 10^5 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 5×10^5 bytes of storage. Nonetheless, for typical problems on typical systems, packing variables and derivative objects together seems preferable to separate messages.

4. Automatic Differentiation of Reduction Operations

In addition to basic features for sending data via messages or defining data layouts, most parallel programming environments provide parallel reduction operations. A *reduction operation* is an operation that reduces N values residing on up to N processors to a single value using an associative, commutative operator, such as addition. These reduction operations are elementary operations, in the sense that we cannot assume anything about the details of the underlying implementation. As such, we cannot apply AD directly to the reduction operation, but instead must provide a rule for computing the partial derivatives and a mechanism for applying the chain rule, just as we do for other elementary operations.

Applying the chain rule turns out to be simpler than computing the partial derivatives. In the forward mode of AD, applying the chain rule amounts to scaling the derivative vectors by the partial derivatives, then adding. Thus, we have in effect a distributed saxpy operation. Scaling by the partial derivatives is a local operation and can be done in parallel. Adding is just a `sum` reduction operation with vector arguments. Thus, for

```
y = OP(x(1:n))
```

where `OP` is a reduction operation, ∇y can be computed by using

```
forall i = 1:n
   g_y_local(i) = {dOP/dx(i)} * g_x(i)
endfor
g_y = SUM(g_y_local(1:n))
```

For simplicity, our example assumes that the parallel programming environment supports reduction over the vector arguments `g_y_local(i)`. If this is not the case, we can implement the sum using a loop over scalar `sum` operations or a user-defined reduction operation.

The most common reduction operations are `sum`, `product`, `max`, and `min`. Computing partial derivatives for the `sum` reduction is trivial, since the partial derivative with respect to each value being added is 1. Thus, for the statement `y = SUM(x[1:N])`, we can compute the derivative vector `g_y` using the reduction `g_y = SUM(g_x[1:N])`. In general, computing the partial derivatives for the `min` and `max` reductions is also simple. The partial derivative with respect to the minimum (maximum) is 1 and with respect to all other variables is 0. The difficulty occurs when the minimum (maximum) is not unique. In this case, the derivative is not defined. See [7] for ideas on how to handle this situation.

To gain some insight into the partial derivatives of the `product` reduction operation, we first consider the sequential product

$$y = x_1 * x_2 * x_3 * x_4 * x_5.$$

The partial derivatives of y with respect to x can be expressed as

$$\begin{aligned}
\frac{\partial y}{\partial x_1} &= x_2 * x_3 * x_4 * x_5, \\
\frac{\partial y}{\partial x_2} &= x_1 * x_3 * x_4 * x_5, \\
\frac{\partial y}{\partial x_3} &= x_1 * x_2 * x_4 * x_5,
\end{aligned}$$

Table 1. Parameters for communication and packing times (s = seconds, B = bytes)

System	αcomm (s)	βcomm (B/s)	α_{pack} (s)	β_{pack} (B/s)	N^* (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}

$$\begin{aligned} \frac{\partial y}{\partial x_4} &= x_1 * x_2 * x_3 * x_5, \\ \frac{\partial y}{\partial x_5} &= x_1 * x_2 * x_3 * x_4. \end{aligned}$$

That is,

$$\frac{\partial y}{\partial x_i} = \prod_{j \neq i} x_j.$$

We could compute partial derivatives for the `product` reduction in the same manner. However, this calculation would require that we compute N products of $N-1$ values over N processors, which could be very expensive in terms of both computation and communication. It should be obvious from inspection that there are many redundant subexpressions in the expressions above. We describe two techniques for computing partial derivatives for the `product` reduction that exploit these common subexpressions, modeled on the two standard modes of automatic differentiation.

4.1 Reverse mode

ADIFOR and ADIC use the forward mode globally, but use the reverse mode for computing the local partial derivatives of statements like the one above. The reverse mode avoids recomputing common subexpressions in the partial derivatives.

$$\begin{aligned} t_1 &= x_1 * x_2 \\ t_2 &= t_1 * x_3 \\ t_3 &= t_2 * x_4 \\ t_4 &= x_5 * x_4 \\ t_5 &= t_4 * x_3 \\ y &= t_3 * x_5 \\ \frac{\partial y}{\partial x_1} &= t_5 * x_2 \\ \frac{\partial y}{\partial x_2} &= t_5 * x_1 \\ \frac{\partial y}{\partial x_3} &= t_1 * t_4 \\ \frac{\partial y}{\partial x_4} &= t_2 * x_5 \\ \frac{\partial y}{\partial x_5} &= t_3 \end{aligned}$$

The computation of t_1 through t_3 is a prefix computation, while the computation of t_4 and t_5 is a reverse prefix, or suffix, computation. Thus, in order to use the same technique for computing partial derivatives for a `product` reduction, we need a parallel prefix operation and a reverse parallel prefix operation. Many parallel programming environments provide a built-in parallel prefix operation, and an efficient implementation is available for most others. Thus, we can compute partial derivatives using a sequence of data exchanges (to facilitate the reverse parallel prefix, since this operation is often not provided) and parallel prefix operations. After the partial derivatives have been computed, they are used to scale the local derivative, and a `sum` reduction is used to compute the total derivative.

4.2 Forward mode

Partial derivatives can also be computed by using the forward mode of AD. The forward mode of AD performs a scaling and summation of derivative vectors at every step.

$$\begin{aligned} t_1 &= x_1 * x_2, \\ \nabla t_1 &= x_2 * \nabla x_1 + x_1 * \nabla x_2, \\ t_2 &= t_1 * x_3, \\ \nabla t_2 &= x_3 * \nabla t_1 + t_1 * \nabla x_3, \\ t_3 &= t_2 * x_4, \\ \nabla t_3 &= x_4 * \nabla t_2 + t_2 * \nabla x_4, \\ y &= t_3 * x_5, \\ \nabla y &= x_5 * \nabla t_3 + t_3 * \nabla x_5. \end{aligned}$$

Note that each multiplication is a scalar-vector product and could be very expensive for long derivative vectors.

We can use the forward mode to propagate derivatives through a `product` reduction using reduction on the user-defined sequential function $(y, \nabla y) = F(x[1:N], \nabla x[1:N])$.

$$\begin{aligned} y &= \prod_{i=1}^{N} x_i, \\ \nabla y &= \sum_{i=1}^{N} \frac{\partial y}{\partial x_i} \nabla x_i. \end{aligned}$$

Since y and ∇y can be communicated together, the overall communication requirements for this algorithm are less

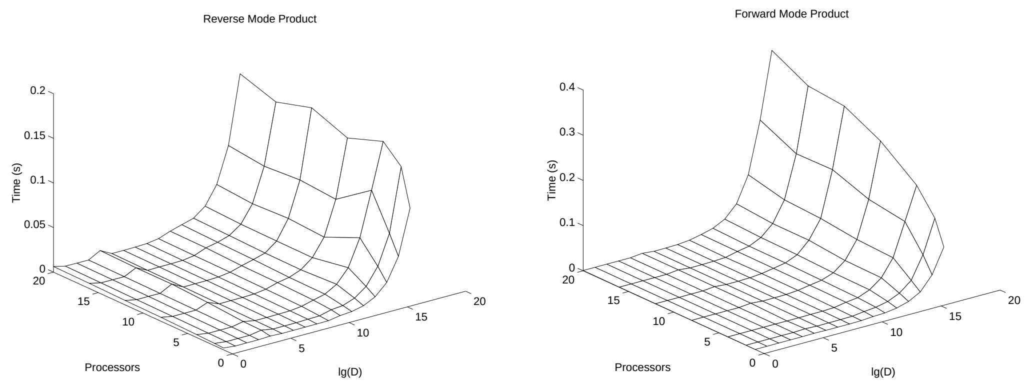

Figure 6. Runtimes for forward and reverse methods for product reduction.

than those for the algorithm based on the reverse mode, especially for systems with high latency. However, the computational cost is higher, and because we are performing a reduction on a user-defined function, compiler optimization may be restricted.

4.3. Comparison of approaches

An analysis of the two proposed rules for differentiating `product` reduction reveals that the reverse mode has a complexity of approximately

$$((4 + D)\log_2 P + D)t_c + (D\log_2 P)t_w + (5\log_2 P)t_s,$$

where D is the length of the derivative vectors, P is the number of processors, t_c is the time to perform a multiplication or an addition, t_s is the startup cost for each message, and t_w is the per-word cost of each message. The forward mode has a complexity of approximately

$$(3D\log_2 P)t_c + (D\log_2 P)t_w + (\log_2 P)t_s.$$

Further simplification indicates that the forward mode will perform better than the reverse mode approximately when

$$D < 2\frac{t_s}{t_c}.$$

On most modern parallel systems, the ratio t_s/t_c is on the order of 10^3 to 10^4. Thus, we expect better performance from the forward mode except when D is very large.

4.4. Experimental results

To compare the two strategies for differentiating the `product` reduction operation, we implemented them in Fortran using MPI message passing. We then timed them for various vector lengths (N) and derivative vector lengths (D) on an Intel Paragon using partitions ranging from 1 to 20 processors (P).

Figures 6–8 provide a comparison of the two methods for computing the derivatives of the product reduction. Figure 6 presents runtime as a function of D and P for a fixed vector length of 1. The graphs in Figure 7 show runtime as a function of D for fixed N and P, aiding in the direct comparison of the two methods. In these graphs, the solid line represents the method based on the reverse mode discussed in Section 4.1, and the dashed line represents the method based on the forward mode discussed in Section 4.2. Figure 8 indicates which method offers the best performance for various values of P and D. Note that the “wrinkle” for moderate D that can be seen in Figures 6 and 7 is even more pronounced in this diagram, because it occurs for different values of D for the two methods. This offset results in the reverse method demonstrating better performance for $D = 32$. The actual crossover occurs around $D = 2000$, which is in keeping with the theoretical crossover point of $D = 2(t_s/t_c)$ given in Section 4.3.

The results obtained on the Paragon are similar to those obtained on a network of SPARCstations and on an IBM SP3. Furthermore, they are consistent with performance predictions based on the abstract algorithms. Therefore, conclusions based on these results should be generally applicable to a wide range of environments.

5. Conclusions and related work

Our experimental results for the IBM SP and a network of SPARCstation indicate that packing a variable and its associated derivative object into a single message is preferable to sending separate messages. We note, though, that on extremely low latency hardware, it may be preferable to use separate messages. Under these circumstances, the cost of packing and unpacking data may exceed the added cost of a second message. Furthermore, separate messages may be easier to implement.

Based on our results, we conclude that the best algorithm for computing the derivatives of the `product` reduction

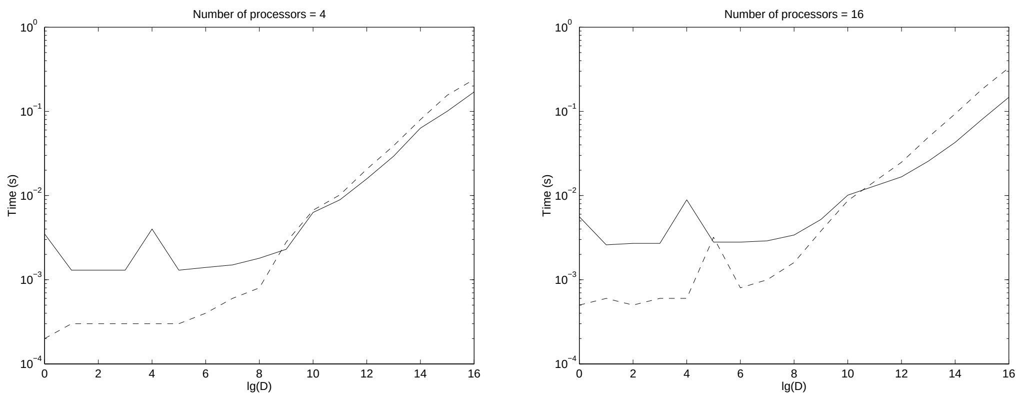

Figure 7. Comparison of forward (dashed) and reverse (solid) methods for fixed number of processors.

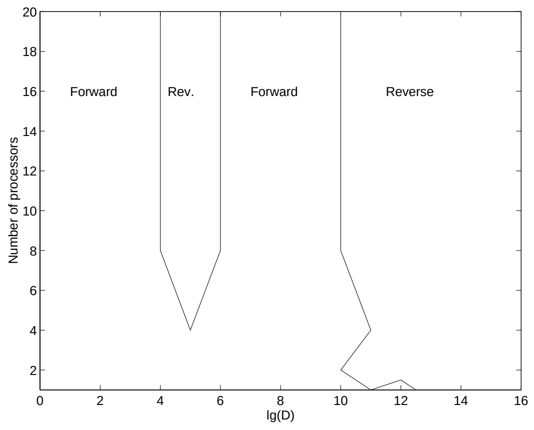

Figure 8. Method for product reduction offering best performance for various values of D and P.

is the one based on the forward mode. For typical problems on typical systems, this method outperforms the reverse mode. It is only when the length of the derivative vector is on the order of 10^3 that the reverse mode offers better performance. Furthermore, a derivative vector length of 10^3 represents a thousandfold increase in work in the derivative computation over the function computation. In this context, a small improvement in the performance of a reduction is not likely to be noticeable. However, for programming environments that do not support reduction of user-defined functions, the reverse mode may be easier to implement.

We have built prototype AD tools for Fortran and C with subsets of MPI, and are in the process of extending the C tool to handle all of MPI. In building these tools, we have needed to address the issues discussed as well as the proper handling of exceptions caused by the evaluation of intrinsic functions at points of nondifferentiability, the use of intertask dataflow analysis to improve efficiency, and whether to utilize the added potential for parallelism created by the automatic differentiation process. We consider these and related issues elsewhere [7]. These issues are also being addressed by other researchers in the area of AD tools for parallel programs [4].

References

[1] M. Berz, C. Bischof, G. Corliss, and A. Griewank. *Computational Differentiation: Techniques, Applications, and Tools*. SIAM, Philadelphia, 1996.

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

[4] A. Carle. Personal communication, 1997.

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

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

[7] P. D. Hovland. *Automatic Differentiation of Parallel Programs*. Ph.D. thesis, University of Illinois at Urbana-Champaign, Urbana, IL, May 1997.