Automatic Generation of Parallel Code for Hessian Computations

H. Martin Bücker, Arno Rasch, and Andre Vehreschild

RWTH Aachen University, Institute for Scientific Computing,
Seffenter Weg 23, D-52074 Aachen, Germany,
{buecker,rasch,vehreschild}@sc.rwth-aachen.de

Abstract. Given a program to compute some function, automatic differentiation can be used to mechanically generate another program capable of evaluating first- and higher-order derivatives of that function. A new strategy for the computation of Hessians by automatic differentiation is proposed where the generated code is automatically parallelized using OpenMP. The approach is applied to compute second-order derivatives of an atmospheric reference model and performance results on a Sun Fire E6900 system are reported.

1 Introduction

First- and second-order derivatives are required in various areas of scientific computing, for instance in algorithms for nonlinear optimization and nonlinear equations [21, 17, 20, 28, 27] or optimal experimental design [2]. Sometimes these derivatives are easy to calculate by hand. However, in a growing number of cases arising from real-world applications in science and engineering, the underlying functions are represented by large programs written in C, C++, Fortran or MATLAB and are too complicated. That is, it is no longer reasonable to expect the user to provide code to compute the corresponding Jacobians or Hessians by hand. Instead, one is often relying on numerical approximations by divided differences. While this approach based on numerical differentiation is easy to implement by calling the program multiple times with perturbed input values, its significant drawback is the presence of truncation errors. Fortunately, automatic differentiation remedies this issue by transforming a given computer program to a new program capable of evaluating (higher-order) derivatives without truncation error. Compared to the original program, the number of floating point operations of the corresponding program generated by this technique is increased, sometimes significantly for higher-order derivatives. Therefore, there is a need for parallelism in derivative computations [13, 3, 6, 7, 9, 14, 15, 25, 26, 19, 29].

In this article, we propose a novel strategy for automatically parallelizing Hessian computations using OpenMP which is implemented in ADIFOR [5, 18], a software tool for automatic differentiation of Fortran 77 programs. The feasibility of this approach is demonstrated by an application to the atmospheric reference

model MSIS–86 [24] predicting temperature and concentration profiles of species in the Earth's atmosphere above 120 km.

The structure of this note is as follows. In Sect. 2, the technology of automatic differentiation is briefly sketched. The new strategy to automatically parallelize the computation of Hessians is introduced in Sect. 3. This strategy is applied to the atmospheric reference model MSIS–86 in Sect. 4 where the parallel performance of the approach is reported.

2 Automatic Differentiation

Automatic Differentiation (AD) is a technology for automatically augmenting computer programs with statements for the computation of derivatives. The basic idea behind AD is that any computer program P performs a—potentially very long—sequence of elementary mathematical operations like binary addition or multiplication, or intrinsic functions, of which the derivatives are known. For each elementary mathematical operation occurring in P, the AD technology generates a corresponding derivative computation. Combining the elementary derivative operations according to the chain rule yields a new program P^{AD} that is capable of not only computing the original function implemented by P, but also derivatives of selected outputs, called *dependent variables*, with respect to certain input parameters, referred to as *independent variables*.

In the so-called *forward mode* of automatic differentiation the derivatives are computed along with the original function. For example, for a statement $c = f(a, b)$, where f denotes an elementary binary operation, the derivative of c can be computed by

$$\nabla c = \frac{\partial c}{\partial a} \nabla a + \frac{\partial c}{\partial b} \nabla b \; . \tag{1}$$

It is assumed that the derivatives of the elementary operation f are known, and the gradients ∇a and ∇b are computed along with the values a and b. The size of the gradients corresponds to the number of directional derivatives propagated through the code and may in general be greater than one, say n. Therefore the computation in (1) may actually involve a loop iterating over the n entries of the gradients. Thus, the AD-generated program P^{AD} needs $O(n)$ times more operations than the original program P.

There is a number of software tools available, implementing the AD technology for various languages such as Fortran, C, C++, or MATLAB. For a detailed list of tools visit the web portal of automatic differentiation, `www.autodiff.org`. A thorough introduction into the theory of AD is given in [30, 22]; applications of AD in different numerous areas are contained in [23, 4, 16, 10].

AD can also be employed to compute higher-order derivatives. For the statement $c = f(a, b)$, the Hessian of c can be computed by

$$\begin{aligned} \nabla^2 c = \frac{\partial c}{\partial a} \nabla^2 a + \frac{\partial c}{\partial b} \nabla^2 b + \frac{\partial^2 c}{\partial a^2} (\nabla a \cdot \nabla a^T) + \frac{\partial^2 c}{\partial b^2} (\nabla b \cdot \nabla b^T) \\ + \frac{\partial^2 c}{\partial a \partial b} (\nabla a \cdot \nabla b^T + \nabla b \cdot \nabla a^T) \; . \end{aligned} \tag{2}$$

This way, the computation of the Hessian $\nabla^2 c$ can be performed by $O(n^2)$ times the number of operations of the original program P, where n is the number of independent variables. In practice, however, one would save half of the operations and storage by exploiting the symmetry of the Hessians. For example, in [1], the Hessians are stored in the LAPACK packed symmetric scheme.

3 Automatically Parallelizing Hessian Computations

The execution time and memory requirement of the *differentiated* program computing first- and second-order derivatives along with the original function values increases by an order of n^2, relative to the original program. Especially for large n the execution time of the differentiated program dramatically increases, and, if the program is executed multiple times, e.g., within an optimization framework requiring Hessian evaluations at different points, this overhead in CPU time is even multiplied. To overcome this situation we suggest the parallel execution of the differentiated code.

The idea is based on the fact that, for large n, almost all the computational work is derivative computation, and the type of the operations for the derivative computations are always similar, e.g., vector linear combinations for first-order derivatives, which can be easily parallelized using appropriate OpenMP directives. In [11, 12] two strategies for parallelizing the computation of first-order derivatives with OpenMP have been proposed. Both strategies can be implemented in software tools for AD, making such an AD tool capable of generating parallel differentiated code.

In this work, we specificly extend the strategy presented in [12] to the parallel computation of Hessians, and report on a recent implementation using the ADIFOR 3.0 [18] automatic differentiation software and the language-independent Hessian module presented in [1] which is interfacing with ADIFOR and ADIC [8].

Recognizing that the loops iterating over the Hessians have always the same length, namely $n(n+1)/2$, if symmetry is exploited, and that each entry in the Hessian can be computed independently, we suggest the following strategy for automatically parallelizing AD-generated code with OpenMP:

1. The whole differentiated code is executed in parallel, i.e., the call to the differentiated subroutine is performed within a parallel region. In case a driver routine for the differentiated subroutine is generated, as provided by ADIFOR 3.0, the corresponding OpenMP directives can be automatically inserted in this driver without changing the calling sequence of the driver routine. This way, the user can call the driver just like in the serial case.
2. All program variables holding second-order derivatives are *shared*. For the second-order derivatives occurring in the lexical extent of the parallel region that has been created in the previous step, this could be achieved by using the OpenMP *shared* clause. All remaining variables holding second-order derivatives, i.e., Hessian variables occurring outside the lexical extent of the parallel region, need to be static in order to make them *shared*. In Fortran,

this is achieved by explicitly adding the `save` attribute to these Hessian variables.

In the present implementation, all second-order derivatives are stored in one-dimensional arrays of length $n(n+1)/2$, representing Hessians in LAPACK packed symmetric storage format. When a Hessian array is updated, like, e.g., $\nabla^2 c$ in (2), the work is shared by the available threads. Since the total size of the Hessian arrays is known in advance, the portion of work delegated to each thread can be determined in advance and kept fixed for all the loops involving Hessian computations. When entering the parallel region, we divide the work on the Hessian arrays such that p threads are assigned disjoint portions of size approximately $n(n+1)/(2p)$. More precisely, for each thread, we compute a pair of indices (`LB`,`UB`) specifying the lower and upper bound of the chunk of the Hessian assigned to this thread. These indices are *private* to each thread. In the current implementation, we also store a pair of indices (`my_i`,`my_j`) indicating the row and column of the Hessian entry that corresponds to the element number `LB` in the one-dimensional array representation in LAPACK packed symmetric storage scheme. These indices are used to pick the correct values from the gradient vectors when computing the outer products element-wise for the elements `LB` to `UB` in the one-dimensional array representation of the Hessian.

3. The computations related to the original function are performed redundantly by each thread. Hence, all variables from the original code must be private. If the variables are in the scope of the parallel region generated in the first step, additional *private* or *firstprivate* clauses for these variables need to be inserted. Intermediate variables occurring outside the lexical scope are *private* by default, except for static variables. These need to be made private by using the *threadprivate* directive. In the present implementation, inactive static variables, those static variables that do not require derivative information, are not recognized and need a *threadprivate* directive explicitly added by the user.
4. Computations of the first-order derivatives are also performed redundantly. The rules for the variables containing gradients are the same as for the variables related to the original function, described in the previous step. The reason for redundant computation of the first-order derivatives is that, for nonlinear operations, the computation of the Hessian requires the full gradients of the arguments for each entry of the Hessian. Hence, additional barriers would be needed before every nonlinear operation when parallelizing the first-order derivative computation. However, we suggest redundant evaluation of gradients in order to save those barriers. In fact, the method presented in this note has the clear advantage that it does not create any barriers within the differentiated code.

As an example, we demonstrate the computation of first- and second-order derivatives of a binary multiplication using the parallelization strategy described above. For a binary multiplication $c = a \cdot b$, the terms of (2) involving $\partial^2 c/\partial a^2$

and $\partial^2 c/\partial b^2$ vanish. So, the gradient ∇c and Hessian $\nabla^2 c$ can be computed by

$$\begin{aligned} \nabla c &= b \cdot \nabla a + a \cdot \nabla b \qquad \text{and} \\ \nabla^2 c &= b \cdot \nabla^2 a + a \cdot \nabla^2 b + \nabla a \cdot \nabla b^T + \nabla b \cdot \nabla a^T \ . \end{aligned}$$

The corresponding Fortran code with OpenMP directives is given in Figure 1.

```
      SUBROUTINE ad_fh_fmulad(pmax,qmax,p1,p2,c,g_c,h_c,a,g_a,h_a,b,g_b,
     +    h_b)

      INTEGER i, j, k, p1, p2, pmax, qmax
      DOUBLE PRECISION c, a, b, g_c(pmax), g_a(pmax), g_b(pmax),
     +    h_c(qmax), h_a(qmax), h_b(qmax)

      INTEGER LB, UB, my_i, my_j
      COMMON /adomp/ LB, UB, my_i, my_j
c$omp threadprivate (/adomp/)
      i = my_i
      j = my_j
c-- Hessian computation
      DO k = LB, UB
        h_c(k) = b*h_a(k) + a*h_b(k) + g_a(i)*g_b(j) + g_a(j)*g_b(i)
        IF (i.eq.j) then
          i = i+1
          j = 1
        ELSE
          j = j+1
        ENDIF
      ENDDO
c-- gradient computation
      DO i = 1, p1
        g_c(i) = b*g_a(i) + a*g_b(i)
      ENDDO
      END
```

Fig. 1. Fortran code for a parallel update of the Hessian `h_c` for a binary multiplication with scalar arguments `a` and `b`. The Hessians of `a` and `b` are denoted by `h_a` and `h_b`, respectively. The corresponding gradients are denoted by `g_a` and `g_b`. The gradient of `c` is updated in a redundant fashion.

The example presented in this figure performs a parallel update of the Hessian (returned in the one-dimensional array `h_c`) for a binary multiplication with scalar arguments `a` and `b`. The Hessians of `a` and `b` are denoted by `h_a` and `h_b`, respectively. The corresponding gradients are denoted by `g_a` and `g_b`. The gradient vector of `c` is also updated, but in a redundant fashion. Practically, the code for the derivative computations is encapsulated in subroutines which are provided in a library. As described before, each thread has a private pair of integers, (`LB`,`UB`), specifying lower and upper bounds of the chunk it is assigned to. The parallel computation of the outer products is performed using temporary integer variables (`i`,`j`) selecting the correct elements from the gradient vectors. Another private pair of integers, (`my_i`,`my_j`), is needed to initialize the temporary variables (`i`,`j`). The values for `LB`, `UB`, `my_i`, and `my_j` are computed once, and passed to the library routines via a *threadprivate* common block. Alternatively, these values could be passed via the argument list.

4 Parallel Second-order Derivatives for MSIS–86

Atmospheric reference models predicting temperature and concentration profiles of species in the upper atmosphere were first developed in the early sixties based on theoretical considerations and satellite drag data. A prominent example of an atmospheric reference model is a suite of models known as the Mass Spectrometer Incoherent Scatter (MSIS) models [24] developed at NASA's Goddard Space Flight Center.

In a preparatory step, the MSIS–86 code was slightly modified to strictly conform to the Fortran 77 standard. All variables are initialized before their first use. In addition, all single precision variables and constants are promoted to double-precision in order to avoid under- or overflow in the derivative computations which tend to have a larger dynamic range of values than computations occurring in the original function.

In the following experiments, we consider the function

$$\varrho = f(\delta, \lambda, \mathbf{p})$$

computing the total mass density ϱ for a given geodetic latitude δ, longitude λ, and 300 scalar parameters $\mathbf{p}$ representing measurements from several rockets, satellites and incoherent scatter radars. Note that the altitude and local apparent solar time are kept constant. The function f is evaluated on an equidistant 10×10 grid varying δ and λ. Selecting ϱ and $\mathbf{p}$ as dependent and independent variable, respectively, we automatically generate OpenMP-parallelized derivative code for computing $\partial\varrho/\partial\mathbf{p}$ and $\partial^2\varrho/\partial\mathbf{p}^2$, in addition to ϱ. The sparsity pattern of the Hessian $\partial^2\varrho/\partial\mathbf{p}^2$ for fixed parameters δ and λ is displayed in Figure 2.

Performance experiments have been conducted on a Sun Fire E6900 system, equipped with 24 UltraSparc IV dual core processors running at 1.2 GHz clock speed, and 96 GByte of main memory. The speedup for 1 to 32 threads is given in Figure 3 where the speedup is related to the parallel version with a single thread.

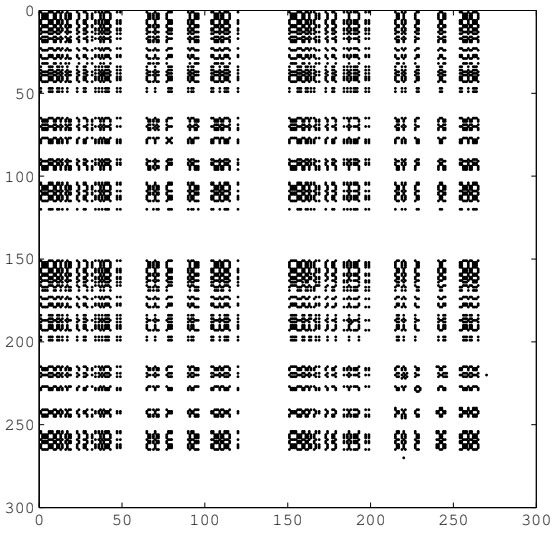

Fig. 2. The sparsity pattern of the Hessian $\partial^2 \varrho / \partial \mathbf{p}^2$

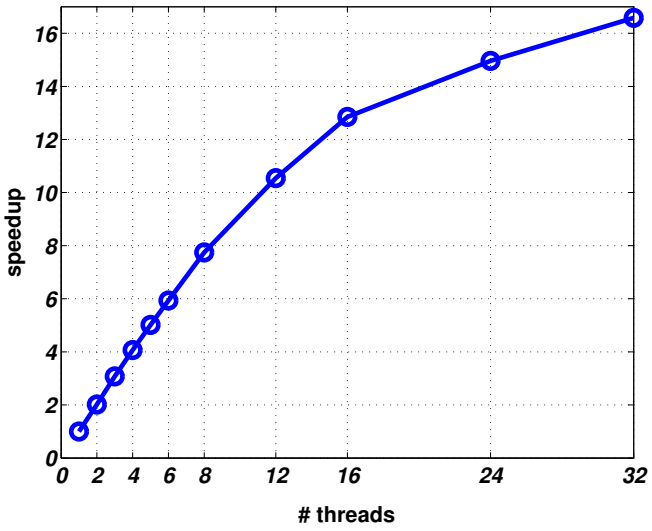

Fig. 3. Speedup of the parallel Hessian computation for the MSIS–86 model.

For a small number of threads, the speedup is almost linear. For larger numbers of threads, only a moderate increase of the speedup can be observed. This is likely caused by the dramatic increase of memory traffic generated by large numbers of threads. In the Sun Fire E6900 system, there are 6 CPU boards containing 4 processors each. Since the majority of the data is located on one board, the limiting resource is the memory bandwidth between board and backplane, which is 4.8 GB/s on the Sun Fire E6900. The approximate memory bandwidth consumed by the application and the Mflops rate per thread is summarized in Table 1. In particular, for large number of threads, the memory bandwidth consumed by the application is quite close to the theoretical maximum of 4.8 GB/s. Nevertheless, the result is remarkable since speedup is achieved in a fully automatic way that requires no interaction with the user.

# threads	memory bandwidth [GB/s]	Mflops/thread
1	0.5944	129.7795
2	0.9678	131.3873
3	1.3082	134.1513
4	1.5915	133.5491
5	1.8365	132.3858
6	2.0469	131.0441
8	2.4253	129.3738
12	2.8741	119.2794
16	3.2792	110.7988
24	3.6324	88.6993
32	3.8884	75.9986

Table 1. Approximate memory bandwidth consumed by the application running with various numbers of threads, and achieved Mflops rate per thread.

5 Concluding remarks

Given a serial source code, a set of techniques referred to as automatic differentiation can be used to generate code for computing gradients and Hessians. Depending on the size of the Hessians, this code may consume considerably more execution time and memory, compared to the original program. Therefore, we suggest to use OpenMP to automatically parallelize the computation of the Hessians, which is by far the most expensive task. The key idea of our new approach is the fact that the computation of each element of a Hessian can be performed independently by relatively simple loops which are easy to parallelize. The total size of the Hessian is typically large leading to large data structures

and loops involving many iterations. Hence, the shared-memory model is particularly well-suited for this parallelization approach. Since the evaluation of the function computed by the original code is tightly interleaved with the first- and second-order derivative computation, we perform a redundant evaluation of the original function and its gradients, enabling parallel processing without synchronization. The new implementation of the parallelization strategy using ADIFOR 3.0 [18] and the Hessian module [1] is able to generate ready-to-use parallel code for computing first- and second-order derivatives.

Finally, we report on an application of the proposed strategy to the MSIS–86 atmospheric model leading to an augmented model capable of evaluating second-order derivatives in parallel. Performance results have shown the feasibility of the new approach. We stress that this approach is fully automatic, which, integrated in automatic differentiation tools, allows the user to accurately compute large Hessians in an efficient way.

Acknowledgments

The authors are grateful to Mike Fagan of Rice University for sharing his insight into the ADIFOR 3.0 system. This work was stimulated by Marc Kalkuhl and Wolfgang Wiechert of the Department of Simulation, University of Siegen, Germany, by their interest in high precision satellite orbit simulations. We also thank Samuel Sarholz, Alexander Spiegel, and Christian Terboven for valuable discussions on the performance analysis. This research is partially supported by the Deutsche Forschungsgemeinschaft (DFG) within SFB 401 "Modulation of flow and fluid–structure interaction at airplane wings," RWTH Aachen University, Germany.

References

1. J. Abate, C. Bischof, A. Carle, and L. Roh. Algorithms and design for a second-order automatic differentiation module. In *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computing (ISSAC'97)*, pages 149–155, New York, NY, USA, 1997. ACM Press.
2. A. C. Atkinson and A. N. Donev. *Optimum Experimental Designs*, volume 8 of *Oxford Statistical Science Series*. Oxford University Press, Oxford, UK, 1992.
3. J. Benary. Parallelism in the reverse mode. In Berz et al. [4], pages 137–148.
4. M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools*. SIAM, Philadelphia, 1996.
5. 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.
6. C. Bischof, A. Griewank, and D. Juedes. Exploiting parallelism in automatic differentiation. In E. Houstis and Y. Muraoka, editors, *Proceedings of the 1991 International Conference on Supercomputing*, pages 146–153, New York, 1991. ACM Press.

7. C. H. Bischof. Issues in parallel automatic differentiation. In A. Griewank and G. Corliss, editors, *Automatic Differentiation of Algorithms*, pages 100–113, Philadelphia, PA, 1991. SIAM.
8. C. H. Bischof, L. Roh, and A. Mauer. ADIC — An extensible automatic differentiation tool for ANSI-C. *Software: Practice and Experience*, 27(12):1427–1456, 1997.
9. H. M. Bücker, K. R. Buschelman, and P. D. Hovland. A Matrix-Matrix Multiplication Approach to the Automatic Differentiation and Parallelization of Straight-Line Codes. In U. Brinkschulte, K.-E. Großpietsch, C. Hochberger, and E. W. Mayr, editors, *Workshop Proceedings of the International Conference on Architecture of Computing Systems ARCS 2002, Germany, April 8–12, 2002*, pages 203–210, Berlin, 2002. VDE Verlag.
10. H. M. Bücker, G. F. Corliss, P. D. Hovland, U. Naumann, and B. Norris, editors. *Automatic Differentiation: Applications, Theory, and Implementations*, volume 50 of *Lecture Notes in Computational Science and Engineering*. Springer, New York, 2005.
11. H. M. Bücker, B. Lang, D. an Mey, and C. H. Bischof. Bringing Together Automatic Differentiation and OpenMP. In *Proceedings of the 15th ACM International Conference on Supercomputing, Sorrento, Italy, June 17–21, 2001*, pages 246–251, New York, 2001. ACM Press.
12. H. M. Bücker, B. Lang, A. Rasch, C. H. Bischof, and D. an Mey. Explicit Loop Scheduling in OpenMP for Parallel Automatic Differentiation. In J. N. Almhana and V. C. Bhavsar, editors, *Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications, Moncton, NB, Canada, June 16–19, 2002*, pages 121–126, Los Alamitos, CA, USA, 2002. IEEE Computer Society.
13. C. H. Bischof and P. D. Hovland. Automatic Differentiation: Parallel Computation. In C. A. Floudas and P. M. Pardalos, editors, *Encyclopedia of Optimization*, volume I, pages 102–108. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2001.
14. A. Carle. Automatic Differentiation. In J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennnedy, L. Torczon, and A. White, editors, *Sourcebook of Parallel Computing*, pages 701–719. Morgan Kaufmann, San Francisco, CA, 2003.
15. A. Carle and M. Fagan. Automatically Differentiating MPI-1 Datatypes: The Complete Story. In Corliss et al. [16], pages 215–222.
16. G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors. *Automatic Differentiation of Algorithms: From Simulation to Optimization*. Springer, New York, 2002.
17. J. E. Dennis, Jr. and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, Englewood Cliffs, 1983.
18. M. Fagan and A. Carle. Adifor 3.0 overview. Technical Report CAAM–TR00–03, Rice University, Department of Computational and Applied Mathematics, 2000.
19. H. Fischer. Automatic differentiation: Parallel computation of function, gradient and Hessian matrix. *Parallel Computing*, 13:101–110, 1990.
20. R. Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, New York, 2nd edition, 1987.
21. P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization*. Academic Press, New York, 1981.
22. A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, Philadelphia, 2000.

23. A. Griewank and G. Corliss. *Automatic Differentiation of Algorithms*. SIAM, Philadelphia, 1991.
24. A. E. Hedin. MSIS–86 thermospheric model. *Journal of Geophysical Research*, 92(A5):4649–4662, 1987.
25. P. Hovland. *Automatic Differentiation of Parallel Programs*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, USA, 1997.
26. P. D. Hovland and C. H. Bischof. Automatic differentiation of message-passing parallel programs. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 98–104, Los Alamitos, CA, 1998. IEEE Computer Society Press.
27. C. T. Kelley. *Iterative Methods for Optimization*. SIAM, Philadelphia, 1999.
28. J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, New York, 1999.
29. P. Heimbach and C. Hill and R. Giering. An efficient exact adjoint of the parallel MIT general circulation model, generated via automatic differentiation. *Future Generation Computer Systems*, 21(8):1356–1371, 2005.
30. L. B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1981.