Aachen Institute for Advanced Study in Computational Engineering Science

Preprint: AICES-2008-4

30/October/2008

Semi-automatic Parallelization of Direct and Inverse Problems for Geothermal Simulation

M. Bücker, A. Rasch, V. Rath and A. Wolf

RWTH AACHEN UNIVERSITY

AICES

Financial support from the Deutsche Forschungsgemeinschaft (German Research Association) through grant GSC 111 is gratefully acknowledged.

©M. Bücker, A. Rasch, V. Rath and A. Wolf 2008. All rights reserved

List of AICES technical reports: http://www.aices.rwth-aachen.de/preprints

Semi-automatic Parallelization of Direct and Inverse Problems for Geothermal Simulation*

H. Martin Bücker, Arno Rasch
Institute for Scientific Computing
RWTH Aachen University
D–52056 Aachen, Germany
{buecker,rasch}@sc.rwth-aachen.de

Volker Rath, Andreas Wolf†
Institute for Applied Geophysics
RWTH Aachen University
D–52056 Aachen, Germany
{rath,wolf}@geophysik.rwth-aachen.de

ABSTRACT

We describe a strategy for parallelizing a geothermal simulation package using the shared-memory programming model OpenMP. During the code development OpenMP is employed for the direct problem in such a way that, in a subsequent step, the OpenMP-parallelized code can be transformed via automatic differentiation into an OpenMP-parallelized code capable of computing derivatives for the inverse problem. Performance results on a Sun Fire X4600 using up to 16 threads are reported demonstrating that, for the derivative computation, an approach using nested parallelism is more scalable than a single level of parallelism.

Categories and Subject Descriptors

G.1.4 [**Numerical Analysis**]: Quadrature and Numerical Differentiation—*automatic differentiation*; D.1.2 [**Programming Techniques**]: Automatic Programming—*program transformation*; G.1.0 [**Numerical Analysis**]: General—*numerical algorithms, parallel algorithms*; J.2 [**Computer Applications**]: Physical Sciences and Engineering—*Earth and atmospheric sciences*

General Terms

Performance, Languages

Keywords

Automatic differentiation, forward mode, ADIFOR, automatic parallelization, shared-memory programming, nested parallelism, OpenMP

*This research is partially supported by the German Federal Ministry of Education and Research (BMBF) under the contract 03SF0326A "MeProRisk: Novel methods for exploration, development, and exploitation of geothermal reservoirs - a toolbox for prognosis and risk assessment."

†corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.
Copyright 2009 ACM 978-1-60558-166-8/09/03 ...$5.00.

1. INTRODUCTION

Over the past decades numerical simulations have been gaining popularity to predict geophysical phenomena of increasing level of complexity. The availability of advanced computer architectures as well as sophisticated computational techniques have made possible the study of computer models for important geophysical processes such as, for instance, the deformation within the Earth's lithosphere[17] or the propagation of seismic waves through fully three-dimensional models of the whole Earth[16].

To better asses the quality of numerical simulations, it is often desirable to quantify the sensitivities of the results subject to changes in the model parameters. Derivatives play also a key role in inverse problems in which actual observations are used to infer properties of the underlying computational model [22]. Prompted by their high computational effort and their widespread use in the geosciences, inverse problems are—most likely—among the top candidates to consume the ever increasing computational power offered by future multicore architectures. The recent paradigm shift from serial to parallel computing thus enforces the computational scientist to bring together computation of derivatives and parallel processing.

The aim of this note is to study the evaluation of derivatives by automatic differentiation in the context of shared-memory parallel programming using a concrete geophysical application. More precisely, we consider a geothermal simulation code sketched in Sect. 2 and transform this OpenMP-parallelized code via automatic differentiation mentioned in Sect. 3. The main contribution reported in Sect. 4 is to introduce a novel strategy to arrange for an OpenMP-parallelization of a Fortran program in such a way that a subsequent transformation via automatic differentiation can reuse this given parallelization strategy, optionally adding an additional level of parallelism trying to improve the performance. In Sect. 5, results are reported demonstrating that nested parallelism is advantageous in terms of performance when using a large number of threads.

2. GEOTHERMAL SIMULATION CODE

The Institute for Applied Geophysics at RWTH Aachen University is currently developing a new geothermal simulation package. Like its predecessor SHEMAT [8] (Simulator for HEat and MAss Transport), the new software solves the coupled transient equations for groundwater flow, heat transport, and the transport of reactive solutes in porous media at high temperatures in three space dimensions. In

this study, we consider a simplified sample case with steady-state flow and heat transport in which the chemically reactive transport is not taken into account. More precisely, the groundwater flow equation is reformulated as

$$\nabla \cdot \left(\frac{\rho_f \, g}{\mu_f} k \cdot \nabla h \right) + Q = 0, \tag{1}$$

using the hydraulic head h as the primary variable and denoting the source term by Q. Here, the symbol k denotes the hydraulic permeability, which is assumed to be isotropic for the sake of simplicity. The symbols ρ_f and μ_f represent density and dynamic viscosity of the pore fluid, respectively, and g denotes the gravity constant. Both ρ_f and μ_f depend on temperature T and pore water pressure P. The latter has to be calculated from the distribution of head and the depth z according to the definition given by de Marsily [10]

$$P(z,h) = P(0) + \int_0^z \rho_f(\tilde{z})\, g(h-\tilde{z}) d\tilde{z}\,, \tag{2}$$

where $P(0) \approx 10^5$ Pa is the pressure at the surface.

The temperature is controlled by the steady state heat transport equation

$$\nabla \cdot (\lambda_e \nabla T) - \rho_f \, c_f \, \mathbf{v} \cdot \nabla T + A = 0, \tag{3}$$

which comprises a conductive term, a source term A similar to the head equation, and an additional advective transport term containing the filtration velocity $\mathbf{v}$, where c_f is the fluid specific heat capacity, and

$$\lambda_e = \lambda_r^{1-\phi} \cdot \lambda_f^{\phi} \tag{4}$$

is the effective thermal conductivity of the porous medium with porosity ϕ. The symbols λ_r and λ_f denote the thermal conductivity of the rock matrix and the fluid, respectively. The filtration velocity can be calculated from hydraulic head by Darcy's law [10]

$$\mathbf{v} = \frac{\rho_f g}{\mu_f}\, k \cdot \nabla h\,. \tag{5}$$

Thermal conductivities of the rock matrix and fluid as well as the other fluid properties, ρ_f, μ_f and c_f, and consequently the filtration velocity in (5), depend on temperature and pressure. Together with (2) these functions constitute a nonlinear coupling between (1) and (3) which are discretized by standard finite-difference techniques as detailed in [8]. The resulting nonlinear system of equations is solved by a simple alternating fixed-point iteration as described, e.g., in [15].

3. AUTOMATIC DIFFERENTIATION

The term "Automatic Differentiation" (AD) [11, 19] refers to a set of techniques that transforms a given computer program to evaluate a function f at some point $\mathbf{x}$ into a new program to evaluate $\partial f/\partial \mathbf{x}$ or higher-order derivatives at the same point $\mathbf{x}$. The automatically-generated program is called the *differentiated* program associated with the original program. The AD technology exploits the fact that any program consists of a sequence of elementary functions like addition or multiplication as well as intrinsic functions. The chain rule is then used to accumulate the known derivatives of these functions in a mechanical way. In contrast to numerical differentiation by divided differences, AD relies on derivatives without truncation error to evaluate $\partial f/\partial \mathbf{x}$.

There are different AD techniques to generate differentiated programs that, in exact arithmetic, compute the same numerical results but differ dramatically in time and storage. The two fundamental AD techniques are called forward and reverse mode. For the sake of simplicity, we focus on the forward mode in this note. The advantages of AD in different application areas are reported in the proceedings of the international conferences [1, 4, 5, 9, 12]. A list of software tools implementing the AD technology is compiled at the community portal `www.autodiff.org`. At the time of writing, no AD tool supports the complete OpenMP standard such that, currently, a certain amount of manual massaging is necessary to transform OpenMP-parallelized code.

4. AUTOMATIC DIFFERENTIATION OF OPENMP-PARALLELIZED CODE

4.1 Parallelization of the Original Code

Due to the high computational requirements for realistic geophysical simulations, the new simulation code is parallelized using OpenMP [6, 7]. As described in [23] the parallelization aims at the two most time- and memory-intensive parts of the program, namely the solution of large sparse linear systems of equations and the assembly of the corresponding coefficient matrices, which together consume about 90% of the overall computing time. A common task when parallelizing sequential source code with OpenMP compiler directives is the specification of data-sharing attributes. In general, the programmer needs to classify the program variables into *shared* and *private* variables using clauses. A variable is *shared* if it resides in the shared memory, i.e., all OpenMP threads have access to the shared variables. On the other hand, if a variable is *private* then each thread has a local copy of it. The OpenMP standard [18] defines a default behavior for program variables that have not been explicitly classified via data-sharing attribute clauses. For example, global variables are shared by default, while local variables declared inside subroutines are private. When parallelizing the new geothermal simulation code, we make frequent use of this default behavior. To this end, variables which need to be private to each thread are removed from the global scope and passed between subroutines via argument lists. The reason for using OpenMP's default data-sharing behavior as opposed to explicit manual specification of data-sharing attributes of variables is to avoid an additional step of manually specifying data-sharing attributes after the AD transformation. As an example, consider the code fragments given in Fig. 1 where a global array variable `w` of size `m` is *shared* in a subroutine `foo` without any data-sharing attribute clause. The choice whether a variable is global and therefore *shared* is discussed in Sect. 4.3.

4.2 Differentiating the parallelized Code

To obtain sensitivity information for the simulation code, the automatic differentiation tool ADIFOR [2] is employed. Rather than applying ADIFOR in a black-box fashion, certain knowledge of the program is taken into account. For example, instead of blindly applying the AD transformation to the linear solver, the mathematical operation of solving a linear system is differentiated by a hierarchical approach [14, 3], and the corresponding derivative code is implemented manually, gluing together existing pieces of code. For further details we refer to [20]. Since ADIFOR generally preserves comments in the source code, the OpenMP directives

```
c global variable w(m)
      call foo(w)
```

```
      subroutine foo(w)
c$omp parallel
c     compute w(1:m)
      ...
c$omp end parallel
```

Figure 1: Example code where OpenMP's default data-sharing behavior is used for a global variable `w`.

```
c global variables w(m), g_w(m)
      do j = 1, n
         call g_foo(w,g_w)
      enddo
```

```
      subroutine g_foo(w,g_w)
c$omp parallel
c     compute w(1:m), g_w(1:m)
      ...
c$omp end parallel
```

Figure 2: Example of AD-generated code with parallelization for `w` and `g_w` stemming from corresponding original code given in Fig. 1.

in the original program are also present in the differentiated code. In general, the classification of the original program variables into *shared* and *private* can be also applied to the corresponding derivative objects in the differentiated program. Most of this classification in the original program is intentionally done implicitly, i.e., utilizing OpenMP's default data-sharing behavior. This default behavior is then adopted in the differentiated code, reducing manual intervention to a minimum. This way, the OpenMP parallelization of the original program works also for the derivative computation. See Fig. 2 for an example where the subroutine `g_foo` is obtained by AD. The variable `g_w` contains a single directional derivative corresponding to `w`. Thus, to compute `n` directional derivatives, the subroutine `g_foo` is called `n` times. The initializations and processing of the results are omitted in this loop.

We compute the Jacobian matrix of temperature T and head h with respect to four scalar parameters, namely the porosity ϕ, the hydraulic permeability k, the rock thermal conductivity λ_r, and the source term A, in three different sedimentary layers. The resulting derivatives of T and h with respect to these twelve scalar model parameters are computed one after another. The derivative computation is started only after the steady-state solution of the forward problem is available, thus reducing the overall execution time. The convergence criterion is no longer applied to the original function but to the corresponding derivatives. In our sample problem this leads to significant variations in computation time for each of the twelve directional derivatives. In Fig. 5, the serial time to compute the derivatives is shown normalized to the minimum computation time of all directional derivatives, $\partial[T,h]/\partial A$ in the first layer.

```
c global variable w(m,n) with n = 1
      call foo(w,1)
```

```
      subroutine foo(w,j)
c$omp parallel
c     compute w(1:m,j)
      ...
c$omp end parallel
```

Figure 3: Example code using an artificial additional dimension, enabling thread-safety in Fig. 4.

```
c global variables w(m,n), g_w(m,n)
c$omp parallel do
      do j = 1, n
         call g_foo(w,g_w,j)
      enddo
c$omp end parallel do
```

```
      subroutine g_foo(w,g_w,j)
c$omp parallel
c     compute w(1:m,j), g_w(1:m,j)
      ...
c$omp end parallel
```

Figure 4: Example of AD-generated code with parallelization in `g_foo` stemming from original code given in Fig. 3 and an additional level of parallelism obtained from executing multiple instances of `g_foo`.

4.3 Parallelizing the Derivative Computation

One option to employ parallelism in the derivative computation is to reuse the parallelization of the original program as described in Sect. 4.2. An obvious alternative is to execute the computations of the twelve directional derivatives in parallel while ignoring the parallelization stemming from the original program. This can be achieved by an OpenMP directive specifying a parallel loop, where in each iteration one of the above mentioned twelve directional derivatives is computed. This requires the derivative computation to be thread-safe. However, as described before, we rely on the default data-sharing attribute rules which regard all global variables as shared. For example, large arrays containing temperature, head, or the corresponding derivative information reside in the shared memory, prohibiting independent, parallel computation of multiple directional derivatives. To remedy this problem, the following strategy is applied. First, an additional dimension is appended to every global variable that is written at least once during the computation. Secondly, all references to these global variables in the differentiated program are modified such that in each iteration of the parallel loop different array portions (slices) are accessed. These preparatory steps are illustrated in Fig. 3, where another dimension, `n`, is added to the variable `w`. Applying AD to the subroutine `foo` yields the subroutine `g_foo` depicted in Fig. 4 which computes `w` and its directional derivative `g_w`. The subroutine `g_foo` is invoked within a loop over the `n` directional derivatives, where in each iteration, `j`, only the slice `g_w(1:m,j)` of the variable

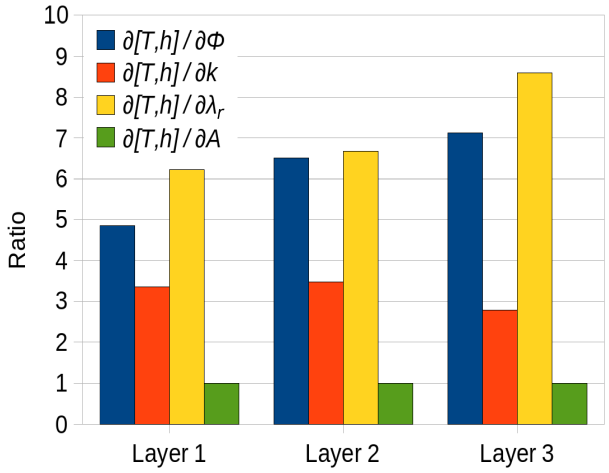

Figure 5: Ratio of the serial execution times for the twelve directional derivatives. The ratios are taken with respect to the minimum execution time.

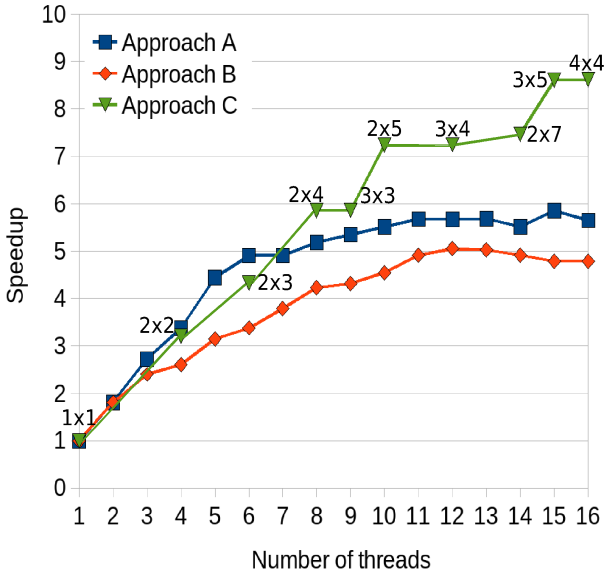

Figure 6: Speedup versus number of threads. Approach A: reusing parallelization of the original code for the derivative computation (blue squares). Approach B: parallel evaluation of the directional derivatives, where the original parallelization is ignored (red diamonds). Approach C: combining approaches A and B yielding nested parallelism (green triangles). The annotations `n` × `m` indicate the distribution of the threads on the outermost and innermost level of parallelism.

`g_w` is accessed, allowing thread-safe parallel execution.

Since the parallelization over the directional derivatives is orthogonal to the parallelization that comes from the original code, these two parallelization approaches can be combined, potentially increasing the granularity of the parallelization. This leads to a program with two nested levels of parallelism, where the outermost level is introduced by the above mentioned loop over the derivative computations while the innermost level corresponds to the OpenMP parallelization stemming from the original code. Suppose that `n` threads are working in the outermost parallel region, distributing the loop over the directional derivatives. When entering the innermost parallel region, each thread creates a team of, say `m`, threads working according to the original parallelization. In total, this results in $\mathtt{n} \cdot \mathtt{m}$ threads.

At this point the rationale behind our approach becomes clear. The OpenMP standard [18] does not allow a global variable to be *private* in one parallel region and *shared* in another. Since we want to exploit OpenMP's default data-sharing behavior in the innermost parallel region, the global variables are *shared* in both levels of parallelism. This is easily accomplished by the additional dimension.

5. PARALLEL PERFORMANCE RESULTS

In this section we compare the performance of the following three parallelization approaches:

A) Reuse of the original parallelization in the differentiated code

B) Parallel evaluation of the directional derivatives while ignoring the original parallelization

C) Combination of A and B yielding nested parallelism

For the performance evaluation we compute the Jacobian matrix on a $30 \times 30 \times 30$ grid. As described in [23], the BiCGStab algorithm [13] with ILU(0) preconditioning [21] is employed. All computations are carried out on a Sun Fire X4600 system which is equipped with eight dual-core Opteron 885 processors operating at 2.6 GHz clock speed. The application is compiled with the Intel compiler version 10.1 for Linux, using the flags "-O3 -axT -openmp -fpp2."

In Fig. 6 the speedup for the three parallelization approaches utilizing up to 16 OpenMP threads is given. Approach A yields almost perfect speedup for up to 6 threads. For a larger number of threads no significant improvement can be observed. The reason for this behavior is that the granularity of parallelism for the given problem size is limited to 6 which is caused by the specific blocking techniques used in the implementation of the preconditioner.

Approach B performs a simultaneous evaluation of the directional derivatives. Hence, the granularity is limited by the number of directional derivatives, which is 12 in the present case study. In practice, the observed speedup is much lower because the execution times for the different directions vary significantly, as shown in Fig. 5, leading to an unbalanced work distribution. Dividing the serial execution time for computing the whole Jacobian matrix by the maximal serial execution time for evaluating one directional derivative yields an upper limit of the potential speedup. In the present case, this limit is given by approximately 6.4. Employing OpenMP's dynamic scheduling mechanism for load balancing, the speedup constantly increases for 1 to 12 threads, achieving a maximum of 5.1.

Approach C which results from a combination of the approaches A and B yields a better speedup than either of the two other approaches, if 8 or more threads are employed. The available threads can be distributed on the two levels of parallelism in various ways. In Fig. 6 each data point is annotated by a pair `n` × `m`, where `n` denotes the number

of threads on the outermost level, and `m` is the number of threads created by each of the `n` threads when entering the innermost level of parallelization. Since the two parallelization strategies A and B are orthogonal, one can expect that the speedup for strategy C using $\mathtt{n} \times \mathtt{m}$ threads is approximately the product of the speedup of approaches A and B with `m` and `n` threads, respectively. In the present study, this can be verified for the configuration $\mathtt{n} \times \mathtt{m} = 4 \times 4$. The measured speedup of approaches A and B in this case are 3.37 and 2.61, so the expected speedup for approach C with the 4×4 configuration is 8.80. The actual speedup is 8.62.

6. CONCLUSIONS

Parallel processing and computation of derivatives are two crucial ingredients to the solution of inverse problems arising in various fields of computational science. The development of a geothermal simulation package is taken as an example to introduce a novel strategy to parallelize a serial program using the OpenMP shared-memory parallel programming paradigm. The specification of parallelism is carefully carried out in such a way that the parallelized code can be transformed by means of automatic differentiation, adopting in the resulting differentiated program the given parallelization strategy specified in the original program. The strategy allows for adding a second level of parallelism in the differentiated program via OpenMP's nested parallelism in which a new team of threads is spawned from within each thread of an existing team of threads, thus increasing the granularity. The performance results indicate that nested parallelism is more scalable than using a single level of parallelism.

7. ACKNOWLEDGMENTS

We thank the Center for Computing and Communication at RWTH Aachen University for giving us access to the Sun Fire X4600 system. The Aachen Institute for Advanced Study in Computational Engineering Science (AICES) provided a stimulating research environment for our work.

8. REFERENCES

[1] M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors. *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 Comput. Sci. Eng.*, 3(3):18–32, 1996.

[3] C. H. Bischof, H. M. Bücker, and P. D. Hovland. On combining computational differentiation and toolkits for parallel scientific computing. In A. Bode et al., editors, *Proc. 6th Int. Euro-Par Conference*, volume 1900 of *LNCS*, pages 86–94, Berlin, 2000. Springer.

[4] C. H. Bischof, H. M. Bücker, P. D. Hovland, U. Naumann, and J. Utke, editors. *Advances in Automatic Differentiation*, volume 64 of *LNCSE*. Springer, Berlin, 2008.

[5] H. M. Bücker, G. F. Corliss, P. D. Hovland, U. Naumann, and B. Norris, editors. *Automatic Differentiation: Applications, Theory, and Implementations*, volume 50 of *LNCSE*. Springer, 2005.

[6] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, 2001.

[7] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2008.

[8] C. Clauser, editor. *Numerical Simulation of Reactive Flow in Hot Aquifers. SHEMAT and Processing SHEMAT*. Springer, New York, 2003.

[9] G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors. *Automatic Differentiation of Algorithms: From Simulation to Optimization*. Springer, New York, 2002.

[10] G. de Marsily. *Quantitative Hydrogeology: Groundwater Hydrology for Engineers*. Academic Press, 1986.

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

[12] A. Griewank and G. Corliss. *Automatic Differentiation of Algorithms*. SIAM, Philadelphia, 1991.

[13] H. A. van der Vorst. BI-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13(2):631–644, 1992.

[14] P. Hovland, B. Norris, L. Roh, and B. Smith. Developing a derivative-enhanced object-oriented toolkit for scientific computations. In M. E. Henderson et al., editors, *Object Oriented Methods for Interoperable Scientific and Engineering Computing*, pages 129–137, Philadelphia, 1999. SIAM.

[15] P. S. Huyakorn and G. F. Pinder. *Computational Methods in Subsurface Flow*. Academic Press, 1983.

[16] D. Komatitsch, S. Tsuboi, and J. Tromp. The spectral-element method in seismology. In A. Levander and G. Nolet, editors, *Seismic Earth: Array Analysis of Broadband Seismograms*, volume 157 of *Geophysical Monograph Series*, pages 205–228. American Geophysical Union, Washington DC, USA, 2005.

[17] L. Moresi, F. Dufour, and H.-B. Mühlhaus. Lagrangian integration point finite element method for large deformation modeling of viscoelastic geomaterials. *J. Comput. Phys.*, 184(2):476–497, 2003.

[18] OpenMP Architecture Review Board. OpenMP application program interface. Version 3.0, May 2008. http://www.openmp.org.

[19] L. B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *LNCS*. Springer, 1981.

[20] V. Rath, A. Wolf, and H. M. Bücker. Joint three-dimensional inversion of coupled groundwater flow and heat transfer based on automatic differentiation: Sensitivity calculation, verification, and synthetic examples. *Geophys. J. Int.*, 167(1):453–466, 2006.

[21] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia, second edition, 2003.

[22] A. Tarantola. *Inverse Problem Theory and Methods for model parameter estimation*. SIAM, Philadelpha, 2004.

[23] A. Wolf, V. Rath, and H. M. Bücker. Parallelisation of a geothermal simulation package: A case study on four multicore architectures. In C. Bischof et al., editors, *Parallel Computing: Architectures, Algorithms and Applications*, volume 15 of *Advances in Parallel Computing*, pages 451–458, Amsterdam, 2008. IOS.