Interpretative Adjoints for Numerical Simulation Codes using MPI*

Michel Schanen, Uwe Naumann, Laurent Hascoët, Jean Utke [a] [b] [c]

[a]LuFG Informatik 12: Software and Tools for Computational Engineering, RWTH Aachen University, Germany, Email: {naumann,schanen}@stce.rwth-aachen.de

[b]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA, Email: utke@mcs.anl.gov

[c]Projet TROPICS, INRIA Sophia-Antipolis, Valbonne, France, Email: laurent.hascoet@sophia.inria.fr

An essential performance and correctness factor in numerical simulation and optimization is access to exact derivative information. Adjoint derivative models are particularly useful if a function's number of inputs far exceeds the number of outputs. The propagation of adjoints requires the data flow to be reversed, implying the reversal of all communication in programs that use message-passing. This paper presents recent advances made in developing the adjoint MPI library AMPI. The described proof of concept aims to serve as the basis for coupling other overloading AD tools with AMPI. We illustrate its use in the context of a specific overloading tool for algorithmic differentiation (AD) for C++ programs. A simplified but representative application problem is discussed as a case study.

1. Motivation

Automatic (or algorithmic) differentiation (AD) [7] is a technique for transforming implementations of multivariate vector functions $\mathbf{y} = F(\mathbf{x})$, where $F : I\!R^n \to I\!R^m$, into programs that compute directional derivatives $\dot{\mathbf{y}} = F' \cdot \dot{\mathbf{x}}$ or adjoints $\bar{\mathbf{x}} = (F')^T \cdot \bar{\mathbf{y}}$ with machine accuracy. $F' = F'(\mathbf{x})$ denotes the Jacobian of F at point $\mathbf{x}$. Several AD tools have been developed over the years (see www.autodiff.org), and numerous successful applications reported [1–3,5,4]. Adjoint codes generated by *reverse mode AD* are of particular interest in the context of large-scale nonlinear optimization ($m = 1$, $n >> 1$) as they allow for gradients of F to be computed at a typically small constant multiple of the cost of evaluating F itself. The downside of adjoint code is that its efficient implementation is a highly challenging exercise for most real-world numerical simulation programs.

The given implementation of F is assumed to decompose into a *single assignment code* (SAC) at every point of interest as follows:

$$\begin{aligned} &\text{for } j = n+1, \ldots, n+p+m \\ &\quad v_j = \varphi_j(v_i)_{i \prec j} \quad , \end{aligned} \tag{1}$$

*This work was supported by the Fond National de la Recherche of Luxembourg under grant PHD-09-145.

where $i \prec j$ denotes a direct dependence of v_j on v_i. The result of each elemental function φ_j is assigned to a unique auxiliary variable v_j. The n *independent inputs* $x_i = v_i$, for $i = 1, \ldots, n$, are mapped onto m *dependent outputs* $y_j = v_{n+p+j}$, for $j = 1, \ldots, m$, and involve the computation of the values of p *intermediate variables* v_k, for $k = n+1, \ldots, n+p$.

For given adjoints of the dependent and independent variables, reverse mode AD propagates adjoints backward through the SAC as follows.

$$\begin{array}{ll} \text{for } j = n+1, \ldots, n+p+m & \text{for } i \prec j \text{ and } j = n+p+m, \ldots, n+1 \\ \quad v_j = \varphi_j(v_i)_{i \prec j} & \quad \bar{v}_i = \bar{v}_i + \bar{v}_j \cdot \dfrac{\partial \varphi_j}{\partial v_i}(v_k)_{k \prec j} \quad . \end{array} \tag{2}$$

The variables $\bar{v}_j$ are assumed to be initialized to $\bar{y}_j$ for $j = n+p+1, \ldots, n+p+m$ and to zero for $j = 1, \ldots, n+p$. A forward evaluation of the SAC is performed to compute all intermediate variables whose values are required for the adjoint propagation in reverse order. The elemental functions in the SAC are processed in reverse order in the second part of Equation (2). See Figure 1 for a simple example. The two entries of the gradient are computed by setting $\bar{v}_4 = 1$. The correctness of this approach follows immediately from the associativity of the chain rule of differential calculus. The problem of performing this data flow reversal within limited memory as efficiently as possible is known to be NP-complete [11].

$$\begin{array}{l} v_3 = v_1 \cdot v_2 \\ v_4 = \cos(v_3) \\ \hline \bar{v}_3 = -\sin(v_3) \cdot \bar{v}_4 \\ \bar{v}_2 = v_1 \cdot \bar{v}_3 \\ \bar{v}_1 = v_2 \cdot \bar{v}_3 \end{array}$$

Figure 1: Adjoint code

AD tools can be separated into two categories depending on the method of implementation. Source code transformation parses the given code and produces a semantically transformed derivative code typically in the same programming language. Alternatively, operator and function overloading can be used to store an internal representation of the SAC, followed by the interpretation of this *tape* to propagate the required adjoints.

The reverse propagation of adjoints in the second part of Equation (2) implies the reversal of any communication in a message-passing setup. If the given implementation of F uses MPI, then sends must become receives, receives become sends, and so forth. First foundations for this approach were laid in [13]. The focus of the present paper is on coupling a further extended version of the adjoint MPI library with an existing overloading tool for AD (Section 2). The approach is verified with a simplified version of a real-world case study in Section 3. Conclusions are drawn (Section 4), followed by a discussion of ongoing and potential future activities.

2. Adjoints by Tape Interpretation

Our AD library dco (derivative code by overloading) uses overloading in C++ based on a user-defined data type to generate a tape in the form of an array tape of s entries.

```
1 class tape_entry {
2   int oc;   // operation code
3   double v; // function value
4   double a; // adjoint value
5   int arg1; // first argument
6   int arg2; // second argument
7 };
```

Function	Tape	Adjoint
$v_1 = \frac{1}{2}$	1:[ASG,$\frac{1}{2}$,-,-,-]	
$v_2 = \pi$	2:[ASG,π,-,-,-]	
$v_3 = v_1 \cdot v_2$	3:[MUL,$\frac{\pi}{2}$,-,1,2]	
	4:[ASG,$\frac{\pi}{2}$,-,3,-]	
$v_4 = \cos(v_3)$	5:[COS,0,-,4,-]	
	6:[ASG,0,-,5,-]	
set_dep(v_4)	7:[DEP,0,-,6,-]	
	7:[DEP,0,1,6,-]	$\bar{v}_4 = 1$
	6:[ASG,0,1,5,-]	
	5:[COS,0,1,4,-]	$\bar{v}_3 = -\sin(v_3) \cdot \bar{v}_4 = -1$
	4:[ASG,$\frac{\pi}{2}$,-1,3,-]	
	3:[MUL,$\frac{\pi}{2}$,-1,1,2]	$\bar{v}_2 = v_1 \cdot \bar{v}_3 = -\frac{1}{2},\ \bar{v}_1 = v_2 \cdot \bar{v}_3 = -\pi$
	2:[ASG,π,$-\frac{1}{2}$,-,-]	
	1:[ASG,$\frac{1}{2}$,$-\pi$,-,-]	

Figure 2. Tape generation and interpretation

All arithmetic operators and the relevant intrinsic functions of C++ are overloaded for variables of the user-defined type. The extended semantics of the elemental functions results in the storage (also *recording*) of its operation code, the computed value, and the indexes of its (up to two) arguments. The interpreter propagates the values of the adjoints backwards through the tape.

```
1  void interpret_tape () {
2    for (int i=s;i>=0;i--)
3      switch (tape[i].oc) {
4          ...
5          case MUL : {
6            tape[tape[i].arg1].a+=
7              tape[tape[i].arg2].v*tape[i].a;
8            tape[tape[i].arg2].a+=
9              tape[tape[i].arg1].v*tape[i].a;
10           break;
11         }
12         case SIN : {
13           tape[tape[i].arg1].a+=
14             cos(tape[tape[i].arg1].v)*tape[i].a;
15           break;
16         }
17     }
18 }
```

The taping and interpretation mechanism is illustrated in Figure 2. Each operation is recorded according to the definition of class tape_entry. The last operation of the forward section is a call of set_dep to declare v_4 as dependent. Hence, the interpreter is run with $\bar{v}_4 = 1$. An extension of dco to MPI requires the taping of MPI calls as well as their correct reversal. A significantly enhanced implementation of the adjoint MPI (AMPI) library proposed in [13] has been developed. The current state of the AMPI library is summarized in Table 1. Refer to [13] for details on awaitall. We simplified the listed MPI calls by omitting parameters that are not relevant for this discussion. From a user perspective MPI structure and specification should be preserved as much as possible.

	MPI Routine	Forward AMPI	Backward AMPI
1	send(V)	send(V)	recv(V)
2	recv(V)	recv(V)	send(V)
3	isend(V,r)	isend(V,r)	wait(V,r)
4	irecv(V,r)	irecv(V,r)	wait(V,r)
5	wait(r)	wait(r)	isend(r) || irecv(r)
6	waitall(r[])	waitall(r[])	[isend(r) || irecv(r)]
7	awaitall(r[])		[wait(r)]
8	bcast(V)	bcast(V)	root: recv(V), not root: send(V)
9	reduce(V)	reduce(V)	bcast(V)

Table 1
AMPI Library: Implemented and tested routines

AMPI provides for every MPI routine a version to be called in the forward section of the adjoint code and its matching implementation for the reverse section.

Let us extend our simple example such that $v_3 = v_1 \cdot v_2$ is computed by one process and $v_4 = \cos(v_3)$ by a second one. Assume blocking communication for exchanging the value of v_3. New opcodes are provided within dco to represent AMPI calls by tape entries.

Figure 3 shows the tape generation and interpretation phases for the two processes. Each process has its own tape. The interpretation is similar to the serial case; the only difference is in the `SEND` and `RECV` entries. Passing v_3 from process 1 to process 2 yields the communication of $\bar{v}_3$ from process 2 to process 1 during the interpretation of the tape. The corresponding AMPI routines are called by the interpreter.

A fundamental difference exists between MPI routines and the elemental operations: MPI uses requests to link variables with their communication. Consequently, an AMPI_request contains memory for the function value, adjoint value, operation code, destination, communicator, and request. As an example we consider a nonblocking communication using isend, irecv, and wait. For the sake of brevity, we restrict the AMPI_request data to the operation code and the communicated value (double precision) in addition to the original MPI_Request.

```
1 typedef struct AMPI_Request {
2     MPI_Request r;
3     int oc;
4     double v;
5     ...
6 } AMPI_Request;
```

A tape-specific dco extension of AMPI_request is used to link wait operations to their respective nonblocking communication.

```
1 typedef struct AMPI_dco_Request {
2     AMPI_Request r;
3     int a;
4     ...
5 } AMPI_dco_Request;
```

This mechanism is explained best with the help of an example. Consider Figure 4. We use a flattened notation to access the relevant entries within AMPI_dco_Request ($r.oc$, $r.v$, $r.a$). The first four tape entries are generated by process 1 as in Figure 3. The isend operation is recorded next, together with the value to be communicated ($\frac{\pi}{2}$), the point of

Process 1		Process 2	
Code	Tape	Code	Tape
$v_1 = \frac{1}{2}$	1:[ASG,$\frac{1}{2}$,-,-,-]		
$v_2 = \pi$	2:[ASG,π,-,-,-]		
$v_3 = v_1 \cdot v_2$	3:[MUL,$\frac{\pi}{2}$,-,1,2]		
	4:[ASG,$\frac{\pi}{2}$,-,3,-]		
send(v_3)	5:[SEND,$\frac{\pi}{2}$,-,4,-]	recv(v_3)	1:[RECV,$\frac{\pi}{2}$,-,-,-]
		$v_4 = \cos(v_3)$	2:[COS,0,-,1,-]
			3:[ASG,0,-,2,-]
		set_dep(v_4)	4:[DEP,0,-,3,-]
		$\bar{v}_4 = 1$	4:[DEP,0,1,3,-]
			3:[ASG,0,1,2,-]
		$\bar{v}_3 = -\sin(v_3) \cdot \bar{v}_4$ $= -1$	2:[COS,0,1,1,-]
recv($\bar{v}_3$)	5:[SEND,$\frac{\pi}{2}$,-1,4,-]	send($\bar{v}_3$)	1:[RECV,$\frac{\pi}{2}$,-1,-,-]
	4:[ASG,$\frac{\pi}{2}$,-1,3,-]		
$\bar{v}_2 = v_1 \cdot \bar{v}_3 = -\frac{1}{2}$	3:[MUL,$\frac{\pi}{2}$,-1,1,2]		
$\bar{v}_1 = v_2 \cdot \bar{v}_3 = -\pi$			
	2:[ASG,π,$-\frac{1}{2}$,-,-]		
	1:[ASG,$\frac{1}{2}$,$-\pi$,-,-]		

Figure 3. dco and AMPI

its definition (tape entry 4), and the AMPI_dco_Request r, whose value is set to $r.v = \frac{\pi}{2}$ and $r.a = 5$ in order for the upcoming wait to be able to link with the current isend. W.l.o.g., we omit any additional computation between the isend/irecv - wait pairs. The tape entry for the wait operation contains the value $r.v$ and the tape index $r.a = 5$ of the corresponding isend retrieved from the associated AMPI_dco_Request r.

Process 2 receives the value $\frac{\pi}{2}$ and sets the AMPI_dco_Request r correspondingly; that is, $r.v = \frac{\pi}{2}$ and $r.a = 1$. This information is used to generate the tape entry for the associated wait operation; that is, the value is set to $r.v = \frac{\pi}{2}$, and the index of its sole argument becomes $r.a = 1$. The cosine of $\frac{\pi}{2}$ obtained from tape entry 2 is found to be zero and is recorded correspondingly. This step completes the tape generation.

The relevant part of the interpretation starts with process 2. The $r.v$ field is now used to convey the adjoint value. In order to compute the desired gradient, the adjoint field of the tape entry corresponding to the dependent variable v_4 is set to one. Interpretation of the COS entry yields $\bar{v}_3 = -\sin(v_3) \cdot \bar{v}_4 = -1 \cdot 1 = -1$, which is stored in the adjoint field of the WAIT entry. Interpretation of the latter amounts to sending the adjoint value to process 1. The associated irecv becomes a wait for the completion of this communication. Process 1 receives the adjoint $\bar{v}_3 = r.v = -1$. The corresponding wait is followed by the remaining interpretation of the MUL entry to get $\bar{v}_1$ and $\bar{v}_2$.

3. Case Study: Heat Equation

We consider a simple data assimilation problem involving the one-dimensional heat equation. A bar of given length is heated on one side for some time. The simulated

Process 1		Process 2	
Code	Tape	Code	Tape
$v_1 = \frac{1}{2}$	1:[ASG,$\frac{1}{2}$,-,-,-]		
$v_2 = \pi$	2:[ASG,π,-,-,-]		
$v_3 = v_1 \cdot v_2$	3:[MUL,$\frac{\pi}{2}$,-,1,2]		
	4:[ASG,$\frac{\pi}{2}$,-,3,0]		
$r.v = v_3$, $r.a = 5$	5:[ISEND,$\frac{\pi}{2}$,-,4,-,r]	irecv$(r.v, r)$	1:[IRECV,-,-,-,-,r]
$r.oc =$ isend		$r.oc =$ irecv	
isend$(r.v, r)$		$r.a = 1$	
wait(r)	6:[WAIT,$r.v$,-,$r.a$,-,r]	wait(r)	2:[WAIT,$r.v$,-,$r.a$,-,r]
	$\equiv$ [WAIT,$\frac{\pi}{2}$,-,5,-,r]	$v_3 = r.v = \frac{\pi}{2}$	$\equiv$ [WAIT,$\frac{\pi}{2}$,-,1,-,r]
		$v_4 = \cos(v_3)$	3:[COS,0,-,2,-]
			4:[ASG,0,-,3,-]
		set_dep(v_4)	5:[DEP,0,-,4,-]
		$\bar{v}_4 = 1$	5:[DEP,0,-,4,-]
			4:[ASG,0,1,3,-]
		$\bar{v}_3 = -1 =$	3:[COS,0,1,2,-]
		$-\sin(v_3) \cdot \bar{v}_4$	
		$r.v = \bar{v}_3$	2:[WAIT,0,-1,1,-,r]
irecv$(r.v, r)$	6:[WAIT,$\frac{\pi}{2}$,0,5,-,r]	isend$(r.v, r)$	
wait(r)	5:[ISEND,$\frac{\pi}{2}$,$r.v$,4,-,r]	wait(r)	1:[IRECV,-,-,-,-,r]
$\bar{v}_3 = r.v$	$\equiv$ [ISEND,$\frac{\pi}{2}$,-1,4,-,r]		
	4:[ASG,$\frac{\pi}{2}$,-1,3,-]		
$\bar{v}_2 = -\frac{1}{2}$	3:[MUL,$\frac{\pi}{2}$,-1,1,2]		
$\bar{v}_1 = -\pi$			
	2:[ASG,π,$-\frac{1}{2}$,-,-]		
	1:[ASG,$\frac{1}{2}$,$-\pi$,-,-]		

Figure 4. Tape generation and interpretation with `AMPI_Request`

temperature distribution is compared with available measurements at a number of discrete points. The initial temperature distribution within the bar is to be estimated such that the discrepancy between simulated and measured values is minimized. This case study has been designed to illustrate the use of the AMPI library with dco. We do not report on the runtime statistics of the parallel version, nor do we discuss any numerical properties of this problem in detail.

We aim to solve the optimization problem $\min_{I\!R} f$, where

$$f = \sum_{i=0}^{n_x} \left(T(1, x_i) - \tilde{T}_i \right)^2$$

such that

$$T_t = c \cdot T_{xx} \quad \text{for } 0 \leq x, t \leq 1$$

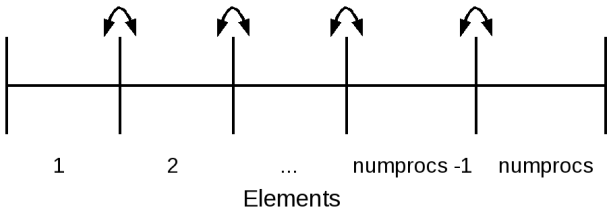


Figure 5. Synchronization

and where

$$
\begin{aligned}
x_i &= \frac{i}{n_x} \quad \text{for } i = 0, \ldots, n_x \\
T &= T(t,x) \ : \ I\!R^2 \to I\!R \\
\tilde{T} &\in I\!R^{n_x} \quad \text{(observations)} \\
T^0 &= T(0,x) = f(x) \quad \text{for } 0 < x < 1 \quad \text{(initial condition)} \\
T(t,0) &= \alpha \quad \text{for } 0 \le t \le 1 \quad \text{(left boundary condition)} \\
T(t,1) &= \beta \quad \text{for } 0 \le t \le 1 \quad \text{(right boundary condition).}
\end{aligned}
$$

Discretization in space is done by centered finite differences with step size δx. Explicit Euler is used for time integration with time step

$$\delta t \le \frac{(\delta x)^2}{2 \cdot c}$$

to ensure stability [8] and to get

$$\frac{T_j^{k+1} - T_j^k}{\delta t} = c \cdot \frac{T_{j+1}^k - 2 \cdot T_j^k + T_{j-1}^k}{(\delta x)^2}$$

and hence

$$
\begin{aligned}
T_j^{k+1} &= T_j^k + c \cdot \frac{\delta t}{(\delta x)^2} \cdot (T_{j+1}^k - 2 \cdot T_j^k + T_{j-1}^k) \\
&= T_j^k + c \cdot \frac{n_x^2}{n_t} \cdot (T_{j+1}^k - 2 \cdot T_j^k + T_{j-1}^k),
\end{aligned}
$$

where $n_x = (\delta x)^{-1}$ and $n_t = (\delta t)^{-1}$.

The cost function is implemented in C++ as shown in Listing 1. For parallelization the bar is decomposed into numprocs elements (lines 3-6). Each process computes one time step on its element (lines 8-10), followed by a synchronization with the neighboring elements (lines 11-33). This simple setup is illustrated in Figure 5. The individual contributions mpicost to the overall costs are finally reduced to cost (lines 34-38).

Two adjustments must be made to the original dco code. All variables of type active must have their types changed to AMPI_dco_double, and the names of all MPI_$*$ routines must become AMPI_$*$.

```
1  AMPI_double cost(int& nx, int& nt, AMPI_double& delta_t, AMPI_double& c, AMPI_double*
       temp, AMPI_double* temp_obs) {
2    AMPI_double buf[4]; AMPI_double mpi_cost = 0; AMPI_Request request[4]; AMPI_Status
         status[4];
3    AMPI_double cost=0;
4    int mpi_j=(id * (nx/numprocs))+1;
5    int mpi_nx=((id+1) * (nx/numprocs))+1;
6    if(id == numprocs -1)
7      mpi_nx--;
8    for(int i=0 ; i <= nt ; i++){
9      for(int j=mpi_j ; j<mpi_nx) ; j++) {
10       temp[j] = temp[j]*c*nx*delta_t*(temp[j+1]-2*temp[j]+temp[j-1]);
11     }
12     // recieve from right & send to right
13     if(id != numprocs - 1){
14       buf[2]=temp[mpi_nx-1];
15       AMPI_Isend(&buf[2],1,MPI_DOUBLE,id+1,0,AMPI_COMM_WORLD,&request[2]);
16       AMPI_Irecv(&buf[1],1,MPI_DOUBLE,id+1,0,AMPI_COMM_WORLD,&request[1]);
17     }
18     // recieve from left & send to left
19     if(id != 0){
20       buf[0]=temp[mpi_j];
21       AMPI_Isend(&buf[0],1,MPI_DOUBLE,id-1,0,AMPI_COMM_WORLD,&request[0]);
22       AMPI_Irecv(&buf[3],1,MPI_DOUBLE,id-1,0,AMPI_COMM_WORLD,&request[3]);
23     }
24     if(id != numprocs-1) {
25       AMPI_Wait(&request[1],&status[1]);
26       AMPI_Wait(&request[2],&status[2]);
27       temp[mpi_nx] = buf[1];
28     }
29     if(id != 0) {
30       AMPI_Wait(&request[0],&status[0]);
31       AMPI_Wait(&request[3],&status[3]);
32       temp[mpi_j-1] = buf[3];
33     }
34   }
35   for(int j=mpi_j ; j<mpi_nx ; j++) {
36     mpi_cost = mpi_cost +(temp[j] - temp_obs[j])* (temp[j]-temp_obs[j]);
37   }
38   AMPI_Reduce(&mpi_cost, &cost, 1, AMPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
39   return cost;
40 }
```

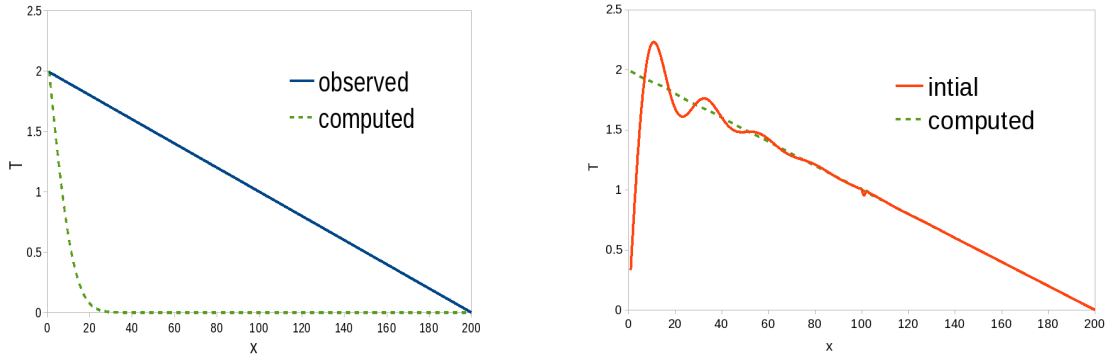
Listing 1: heat

We consider the case $c = 10^3, n_x = 200, n_t = 1000, \alpha = 2, \beta = 0$, and $f(x_i) = 2 - \frac{i}{100}$ for $i = 0, \ldots, n_x$. The results are plotted in Figure 6. The substantial discrepancy between the originally simulated and the observed temperature distributions is reduced by applying a simple steepest descent method. Starting from an initial cost of 227 for $T^0 = 0$, the algorithm performs 401 iterations (taking ∼70 seconds on our PC in the serial case and ∼35 seconds when using 4 processes) to decrease the norm of the gradient to a value less than 10^{-3}.

4. Conclusion and Outlook

Although the AMPI library is still under development, we have been able to verify its current version as robust and user-friendly in the context of both source transformation [13] and overloading tools for automatic differentiation in adjoint mode. Ongoing efforts focus on rigorous testing by application to a number of large-scale numerical simulation codes from the Earth and atmospheric sciences.

Second-order methods for nonlinear optimization require second derivatives in the form



(a) Discrepancy between original simulation of heat distribution starting from $T^0 = 0$ and observed values

(b) Simulation of heat distribution with optimized initial values

Figure 6. Results

of Hessians or projections thereof. For example, a single iteration is performed to push the value of the gradient of f below 10^{-9}, as

$$f(\mathbf{x}^0) = \sum_{i=0}^{n_x} \left(T^{n_t}(x_i^0) - \tilde{T}_i \right)^2$$

is quadratic and

$$T_j^0 = 0$$
$$T_j^{k+1} = T_j^k + c \cdot \frac{n_x^2}{n_t} \cdot (T_{j+1}^k - 2 \cdot T_j^k + T_{j-1}^k)$$

for $j = 1, \ldots, n_x - 1$ and $k = 0, \ldots, n_t - 1$. Consequently, its gradient ∇f is linear, yielding a constant Hessian $\nabla^2 f$. With $\mathbf{x}^0 = 0$ we get the solution $\mathbf{x} = \mathbf{x}^{j+1} = \mathbf{x}^j$ for $j = 1, \ldots$ from the linear system

$$\nabla^2 f(\mathbf{x}^0) \cdot \mathbf{x} = -\nabla f(\mathbf{x}^0)$$

(e.g., by Gauss), since according to Newton's algorithm (see, e.g., [10])

$$\mathbf{x} = \mathbf{x}^0 - \left(\nabla^2 f(\mathbf{x}^0) \right)^{-1} \cdot \nabla f(\mathbf{x}^0) \quad .$$

Unfortunately, the inverse heat propagation problem is ill-posed, leading to a largely meaningless solution unless regularization [12] is applied. Consequently we solve the regularized problem

$$(\nabla^2 f(\mathbf{x}^0) + \alpha \cdot I_{n_x}) \cdot \mathbf{x} = -\nabla f(\mathbf{x}^0)$$

with an appropriately chosen regularization parameter $\alpha > 0$, yielding an acceptable loss in accuracy. While the accumulation of the (dense but constant) Hessian dominates the computation, the overall serial runtime of approximately 45 seconds significantly undercuts that of the corresponding steepest descent algorithm. The latter takes several hours to reduce the residual below 10^{-9}.

The efficient computation of second derivatives of MPI codes requires further extension of the AMPI library in order to be able to communicate second-order adjoint information. Refer to [9] for further details on the computation of directional derivatives for message-passing programs. Feasibility studies are under way. The coupling of the AMPI library with the popular overloading AD tool for C++ ADOL-C [6] is planned.

REFERENCES

1. M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools*, Proceedings Series, Philadelphia, 1996. SIAM.
2. C. Bischof, M. Bücker, P. Hovland, U. Naumann, and J. Utke, editors. *Advances in Automatic Differentiation*, number 64 in LNCSE, Berlin, 2008. Springer.
3. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors. *Automatic Differentiation: Applications, Theory, and Tools*, number 50 in Lecture Notes in Computational Science and Engineering, Berlin, 2005. Springer.
4. G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors. *Automatic Differentiation of Algorithms – From Simulation to Optimization*, New York, 2002. Springer.
5. G. Corliss and A. Griewank, editors. *Automatic Differentiation: Theory, Implementation, and Application*, Proceedings Series, Philadelphia, 1991. SIAM.
6. A. Griewank, D. Juedes, and J. Utke. Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software*, 22(2):131–167, 1996.
7. A. Griewank and A. Walter. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation (2nd Edition)*. SIAM, Philadelphia, 2008.
8. M. Heath. *Scientific Computing. An Introductory Survey*. McGraw-Hill, New York, 1998.
9. P. Hovland and C. Bischof. Automatic Differentiation for Message-Passing Parallel Programs. In *IPPS '98: Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*, Washington, DC, USA, 1998. IEEE Computer Society.
10. C. T. Kelley. *Solving Nonlinear Equations with Newton's Methods*. SIAM, Philadelphia, 2003.
11. U. Naumann. DAG reversal is NP-complete. *J. Discr. Alg.*, 2008. To appear. Appeared online on Elsevier's ScienceDirect as doi:10.1016/j.jda.2008.09.008.
12. A. Tikhonov. On the stability of inverse problems. *Dokl. Akad. Nauk SSSR*, 39(5):195–198, 1943.
13. J. Utke, L. Hascoët, P. Heimbach, C. Hill, P. Hovland, and U. Naumann. Toward Adjoinable MPI. In *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium*, Washington, DC, USA, 2009. IEEE Computer Society.