Time-Parallel Computation of Pseudo-Adjoints for a Leapfrog Scheme*

Christian H. Bischof and Po-Ting Wu
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439-4844
e-mail: {bischof,pwu}@mcs.anl.gov

Argonne Preprint ANL/MCS-P639-0197

Abstract

The leapfrog scheme is a commonly used second-order difference scheme for solving differential equations. If $Z(t)$ denotes the state of the system at time t, the leapfrog scheme computes the state at the next time step as $Z(t+1) = H(Z(t), Z(t-1), W)$, where H is the nonlinear timestepping operator and W are parameters that are not time dependent. In this article, we show how the associativity of the chain rule of differential calculus can be used to compute a so-called adjoint $x^T \cdot (dZ(t)/d[Z(0), W])$ efficiently in a parallel fashion. To this end, we (1) employ the reverse mode of automatic differentiation at the outermost level, (2) use a sparsity-exploiting incarnation of the forward mode of automatic differentiation to compute derivatives of H at every time step, and (3) exploit chain rule associativity to compute derivatives at individual time steps in parallel. We report on experimental results with a 2-D shallow-water equation model problem on an IBM SP parallel computer and a network of Sun SPARCstations.

1 Introduction

The leapfrog method is a commonly used second-order method for solving differential equations (see, for example, [22, pp. 53 ff.] , [8]). Let $Z(t)$ and $Z(t-1)$ denote the

*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.

current and previous state of a time dependent system, respectively. Except for possibly the initial steps, a leapfrog scheme computes the state $Z(t+1)$ at the next time step as

$$Z(t+1) = H(Z(t), Z(t-1), W), \tag{1}$$

where $Z \in \mathbb{R}^n$, $W \in \mathbb{R}^p$ (typically $n \gg p$) are system parameters that are not time dependent and H is the nonlinear operator advancing the system state. That is, the code has the structure shown in Figure 1 (T denotes the last time step).

```
Initialize Z(0) and W.
Compute Z(1).
for t = 1 to T  do
    Z(t+1) = H(Z(t), Z(t-1), W);
    Z(t-1) = Z(t); Z(t) = Z(t+1);
end do
```

Figure 1: Schematic of a Leapfrog Scheme

Let $r : \mathbb{R}^n \mapsto \mathbb{R}$ be a scalar-valued function that is applied to $Z(T)$. In many applications, such as data assimilation, r is a relatively simple merit function, which assesses the distance of the current state with respect to some desirable state. Typically, one is interested in

$$\frac{d\, r}{d\, [Z(0), W]}. \tag{2}$$

The quantity (2) is a "long gradient" of size $n+p$, and, usually, so-called adjoint approaches are employed for its efficient computation. Adjoint approaches can ideally compute the gradient at a floating-point cost that is a few times that of the function, independent of $n+p$, and can be derived either mathematically from the definition of the underlying operator (the continuous adjoint) or by employing the so-called reverse mode of automatic differentiation on the computer code (the discrete adjoint). For details on development and use of adjoint approaches see, for example, the articles in [1, 20, 21].

The ADOL-C [15], GRESS [19], Odyssee [23], and TAMC [12] tools implement the reverse mode of automatic differentiation in an automated fashion. However, as explained, for example, in [6], the implementation of the reverse mode requires one to remember or recompute all intermediate values that may nonlinearly impact the final result. That is, if a variable is assigned 100 different values in the course of the execution of a code, we may need to remember all 100 values that it assumed during its lifetime. A snapshotting scheme proposed by Griewank [13] suggests a way to overcome the potentially enormous storage requirements that would be induced by straightforward tracing. This approach is being adopted in reverse mode tools [16, 3], but the fact that potentially many values need to be stored and/or recomputed still implies that the reverse mode

may require considerable and unpredictable storage requirements to achieve the desired low floating-point complexity.

In contrast, the so-called forward mode of automatic differentiation which is employed, for example, at the outermost level of the ADIFOR [6] and ADIC [9] tools, has predictable storage requirements. When s directional derivatives are computed, memory and runtime increase by a factor that is at most $O(s)$, independent of how linear or nonlinear the code is or how often storage is overwritten. Moreover, this upper bound may be grossly pessimistic. In particular, by exploiting sparsity inherent in many large-scale optimization problems, gradients of such problems can be efficiently computed with forward-mode based tools [5, 7]. More in-depth information on automatic differentiation can be found in [4, 14], and an overview of currently available AD tools is provided at URL `http://www.mcs.anl.gov/Projects/autodiff/AD_Tools`.

In this article, we show how the reverse and forward modes of automatic differentiation can be combined to compute the adjoint quantity (2). In the literature, the term "adjoint" is often used somewhat loosely to denote both the gradient to be computed and the either continuous or discrete adjoint approach that will be employed in its computation. To distinguish our work, where we compute the same gradient as in a true adjoint approach but with a different methodology, we call it a "pseudo-adjoint" approach. The main algorithmic ingredients of our approach are

- the exploitation of the sparsity of the Jacobian that is associated with the operator H in typical stencil-based computations,
- the concurrent computation of derivatives of H at different time steps, and
- a reverse-mode harness that accumulates the derivatives from different time steps in an efficient fashion.

This article is structured as follows. In the next section, we review the capabilities of current forward-mode based AD tools with respect to exploiting sparsity in computing Jacobians, and we show how the Jacobians associated with individual time steps can be computed in parallel. In Section 3, we show how the desired adjoint quantity (2) can be computed in a recursive fashion through (in essence) a series of sparse matrix-vector multiplications. In Section 4, we present experimental results with both a serial and a parallel implementation of this scheme on an IBM/SP parallel computer and a network of Sun SPARCstations. Throughout, we use a 2-D shallow-water equations model problem [25, 24] for illustration. Lastly, we summarize our results.

2 Computing Sparse Timestep Jacobians in Parallel

In typical stencil-based PDE solution schemes, one gridpoint is updated based on the values of a fixed number of surrounding gridpoints. Thus, the Jacobians $\frac{\partial H}{\partial Z(t)}$ and $\frac{\partial H}{\partial Z(t-1)}$ in

$$\frac{d\,Z(t+1)}{d\,[Z(0),W]} = \frac{\partial\,H}{\partial\,Z(t)} \cdot \frac{d\,Z(t)}{d\,[Z(0),W]} + \frac{\partial\,H}{\partial\,Z(t-1)} \cdot \frac{d\,Z(t-1)}{d\,[Z(0),W]} + \frac{\partial\,H}{\partial\,W} \cdot \frac{d\,W}{d\,[Z(0),W]}, \quad (3)$$

will be sparse $n \times n$ matrices. In contrast, $\frac{\partial\,H}{\partial\,W}$, is likely to be a dense $n \times p$ matrix. Equation (3) was obtained by differentiating Equation (1) with respect to some set of parameters X, and all partial derivatives are evaluated at $(Z(t), Z(t-1), W)$. To illustrate, we consider a 2-D shallow-water equations model problem [24, 25].

This code is simple but employs a leapfrog difference scheme and a stencil-based propagation operator. In this particular example, any row of $\left(\frac{\partial\,H}{\partial\,Z(t)}, \frac{\partial\,H}{\partial\,Z(t-1)}, \frac{\partial\,H}{\partial\,W}\right)$ has at most 13 nonzeros and the nonzero structure of this matrix for $t > 1$ is shown in Figure 2 for $n = 363$ and $s = 4$. The total number of nonzeros is 3,101.

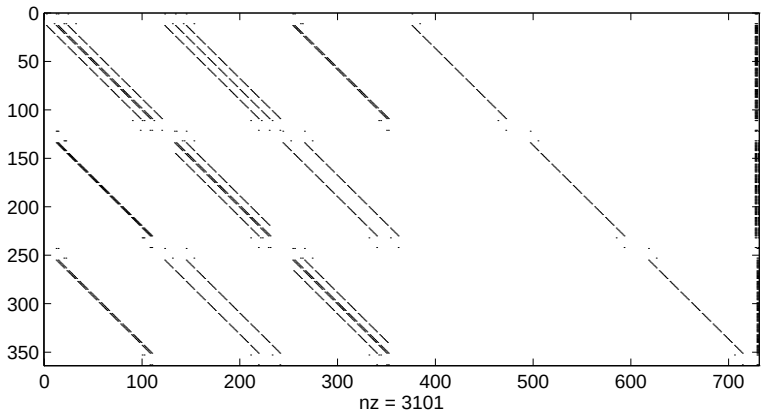

Figure 2: The Sparse Jacobian Associated with the Shallow Water Equations Model

In [11], it was shown how the sparsity of $\frac{\partial\,H}{\partial\,Z(t)}$ and $\frac{\partial\,H}{\partial\,Z(t-1)}$ could be exploited to

efficiently compute the derivatives of $Z(T)$ with respect to some of the initial values X, say, with automatic differentiation. In this approach, the derivative-augmented version g_H of H,

$$[Z(t+1), g\_Z(t+1)] = g\_H(Z(t), g\_Z(t), Z(t-1), g\_Z(t-1), W, g\_W). \tag{4}$$

which computes both $Z(t+1)$ as in (1) and

$$g\_Z(t+1) = \frac{\partial H}{\partial Z(t)} g\_Z(t) + \frac{\partial H}{\partial Z(t-1)} g\_Z(t-1) + \frac{\partial H}{\partial W} g\_W \tag{5}$$

(all partial derivatives are evaluated at $(Z(t), Z(t-1), W)$), was used to compute $\frac{\partial H}{\partial Z(t)}$, $\frac{\partial H}{\partial Z(t-1)}$, and $\frac{\partial H}{\partial W}$ explicitly at each time step, and the overall derivatives $\frac{d Z(t+1)}{d X}$ were computed explicitly through matrix-matrix multiplications, as suggested by Equation(3).

This approach can use either the SparsLinC library, which exploits sparsity without any prior knowledge of the sparsity structure, or a so-called compressed Jacobian approach, which calls SparsLinC once to determine the sparsity structure and then computes a linear combination of columns of the original Jacobian based on a graph-coloring scheme. In either case, $\left(\frac{\partial H}{\partial Z(t)}, \frac{\partial H}{\partial Z(t-1)}, \frac{\partial H}{\partial W}\right)$ can be computed at a cost that is independent of n and dependent only on the cost associated with the PDE stencil.

In the sequel, we refer to $\left(\frac{\partial H}{\partial Z(t)}, \frac{\partial H}{\partial Z(t-1)}, \frac{\partial H}{\partial W}\right)$ evaluated at $(Z(t), Z(t-1), W)$ as the "timestep Jacobian" associated with timestep t. We note that, for a particular time step t, we can compute its associated timestep Jacobian once $Z(t)$ (and hence $Z(t-1)$) has been computed. That is, to compute the Jacobian associated with a particular time step, we require only that the simulation has progressed up to this time step; we do not require the derivatives of previous time steps. Thus, with P processors, these individual time step Jacobians can be generated in the parallel fashion shown in Figure 3. Here, we precompute and store all time steps and the associated states $Z(t)$ and then compute the derivative matrices in a round-robin fashion. Alternatively, processes can be dynamically assigned the time steps whose Jacobian they are responsible for. The mypid() function returns the unique ID number of a particular process between 0 and $P-1$. Suitable invocations of g_H to exploit the sparsity of the timestep Jacobian are detailed for this particular example in [11] and, in general, in [2, 7]. The scheme outlined in Figure 3 allows us to generate all timestep Jacobians in a parallel fashion. The additional floating-point and memory complexity is a fixed multiple of that of the complexity for H, depending on the particular stencil, but not the size of the grid. We also note that a production version of this approach would incorporate the snapshotting approach suggested by Griewank [13] to decrease the memory requirements associated with storing timestep Jacobians. We omitted this aspect in our implementation in order to concentrate on the novel ideas.

Initialize $Z(0)$ and W.
Compute and save $Z(1)$.
for $t = 1$ **to** T **do**
  $Z(t+1) = H(Z(t), Z(t-1), W)$;
  Save $Z(t+1)$;
  $Z(t-1) = Z(t); Z(t) = Z(t+1)$;
end do
for $t = 1$ **to** T **do**
  if (mod(t,P) == myid()) **then**
    if ($t == 1$) **then**
      Compute and store $\left(\frac{d\,Z(1)}{d\,Z(0)}\ ,\ \frac{d\,Z(1)}{d\,W} \right)$.
    else
      Invoke $g\_H(Z(t), g\_Z(t), Z(t-1), g\_Z(t-1), W, g\_W)$ with suitable initializations for $g\_Z(t), g\_Z(t-1)$, and g_W to compute
      $Z(t+1)$ as well as $g\_Z(t+1) = \left(\frac{\partial\,H}{\partial\,Z(t)}, \frac{\partial\,H}{\partial\,Z(t-1)}, \frac{\partial\,H}{\partial\,W} \right)$.
      Store $[t, \frac{\partial\,H}{\partial\,Z(t)}, \frac{\partial\,H}{\partial\,Z(t-1)}, \frac{\partial\,H}{\partial\,W}]$
    end if
  end if
end do

Figure 3: Schematic for Computing Timestep Jacobians in Parallel

3 A High-Level Adjoint Recursion

Returning to our original problem of computing the adjoint (2), we observe that differentiation of (1) implies

$$\begin{aligned}\frac{d\,r}{d\,[Z(0),W]} &= \frac{d\,r}{d\,Z(T)}\cdot\frac{d\,Z(T)}{d\,Z(0)} \\ &= x^T_{1\times n}\cdot\frac{d\,Z(T-1)}{d\,[Z(0),W]}_{n\times(n+p)} + y^T_{1\times n}\cdot\frac{d\,Z(T-2)}{d\,[Z(0),W]}_{n\times(n+p)} + w^T_{1\times(n+p)},\end{aligned}$$

where, invoking (3), we have

$$\begin{aligned}x^T_{1\times n} &= \frac{d\,r}{d\,Z(T)}_{1\times n}\cdot\frac{\partial\,H(Z(T-1),Z(T-2),W)}{\partial\,Z(T-1)}_{n\times n}, \\ y^T_{1\times n} &= \frac{d\,r}{d\,Z(T)}_{1\times n}\cdot\frac{\partial\,H(Z(T-1),Z(T-2,W)}{\partial\,Z(T-2)}_{n\times n}, \text{ and} \\ w^T_{1\times(n+p)} &= \frac{\partial\,H(Z(T-1),Z(T-2),W)}{\partial\,W}_{n\times p}\left(\ 0_{p\times n}\ ,\ I_{p\times p}\ \right).\end{aligned}$$

Here I is the identity, 0 is a zero matrix, and subscripts indicate the size of matrices. For clarity we also indicated the arguments for which the partial derivatives were computed. Note that x and y are n-vectors, while w is a vector of size $n+p$. The computation of x^T and y^T involves two multiplications of sparse $n\times n$ matrices with a dense n-vector, the computation of w the left-multiplication of a dense $n\times p$ matrix by a row-vector of length n. If we recursively apply this approach to the computation of $x^T\cdot\frac{d\,Z(T-1)}{d\,[Z(0),W]}$ and $y^T\cdot\frac{d\,Z(T-2)}{d\,[Z(0),W]}$, we will form the desired adjoint vector through a chain of sparse matrix-vector multiplications involving the timestep Jacobians that were computed in Figure 3. The resulting algorithm is shown in Figure 4. Note that for $t == 1$, the computation of y and z involves the left-multiplication of an $n\times(n+p)$ matrix with an n-vector. To compute $\frac{d\,r}{d\,[Z(0),W]}$, we then execute the algorithm in Figure 3, followed by an invocation of

$$\text{leapfrog\_adjoint}(\frac{d\,r}{d\,Z(T)}, T-1).$$

If $n \gg p$, the main work in the scheme outlined in Figure 4 at every time step consists of two left-multiplications of a sparse $n\times n$ matrix with an n-vector and the generation of two recursive processes. Thus, a total of 2^{T-1} $n\times n$ sparse matrix-vector multiplies will be computed.

We can reduce the number of matrix-vector multiplies by turning the leapfrog scheme into an Euler scheme. To this end, we adjoin successive time steps and define an extended

function leapfrog_adjoint(x,t)
  if $(\ t > 1)$ **then**
    $y^T = x^T \cdot \dfrac{\partial H}{\partial Z(t)};$
    $z^T = x^T \cdot \dfrac{\partial H}{\partial Z(t-1)};$
    } /* left-multiplication of sparse */ /* $n \times n$ matrix by an n vector */
    $w^T = x^T \cdot \dfrac{\partial H}{\partial W} \cdot (\ O_{p\times n}\ ,\ I_{p\times p}\ );$
    $y = \text{leapfrog\_adjoint}(y, t-1);$
    $z = \text{leapfrog\_adjoint}(z, t-1);$
    } /* binary recursion */
    $result = w + y + z;$
  else
    $result = x^T \cdot \left(\dfrac{d\,Z(1)}{d\,Z(0)}\ ,\ \dfrac{d\,Z(1)}{d\,W}\ \right);$
  end if
  return$(result)$;
end function

Figure 4: Recursive Evaluation of Adjoint for a Leapfrog Scheme

state vector $\overline{Z(t)} = (Z(t), Z(t-1))$. We compute the new next state $\overline{Z(t+1)}$ as

$$\begin{aligned} \overline{Z(t+1)} &= (Z(t+1), Z(t)) \\ &= (H(Z(t), Z(t-1), W), Z(t)) \\ &= \overline{H}(Z(t), Z(t-1), W) \\ &= \overline{H}(\overline{Z(t)}, W), \end{aligned} \tag{6}$$

with the "new" update operator $\overline{H}$

$$\overline{H}_{2n\times(2n+p)} = \begin{pmatrix} H_{n\times(2n+p)} \\ I_{n\times n} \quad 0_{n\times(n+p)} \end{pmatrix}. \tag{7}$$

Differentiating (6) with respect to $[Z(0), W]$, we obtain

$$\frac{d\,\overline{Z(t+1)}}{d\,[Z(0),W]} = \frac{\partial\,\overline{H}}{\partial\,Z(t)} \cdot \frac{d\,\overline{Z(t)}}{d\,[Z(0),W]} + \frac{\partial\,\overline{H}}{\partial\,W} \cdot \frac{d\,W}{d\,[Z(0),W]},$$

which, taking into account (7), is equivalent to

$$\begin{pmatrix} \dfrac{d\,Z(t+1)}{d\,[Z(0),W]} \\ \dfrac{d\,Z(t)}{d\,[Z(0),W]} \end{pmatrix} = \begin{pmatrix} \dfrac{\partial\,H}{\partial\,Z(t)} & \dfrac{\partial\,H}{\partial\,Z(t-1)} \\ I_{n\times n} & 0_{n\times n} \end{pmatrix} \cdot \begin{pmatrix} \dfrac{d\,Z(t)}{d\,[Z(0),W]} \\ \dfrac{d\,Z(t-1)}{d\,[Z(0),W]} \end{pmatrix} + \begin{pmatrix} \dfrac{\partial\,H}{\partial\,W} \\ 0_{n\times p} \end{pmatrix} \cdot \begin{pmatrix} \dfrac{d\,W}{d\,[Z(0),W]} \end{pmatrix}. \tag{8}$$

```
function euler_adjoint(x̄, t)
    if ( t > 1) then
        ȳ^T = x̄^T · ( ∂H/∂Z(t)      ∂H/∂Z(t−1) )
                    ( I_{n×n}       0_{n×n}    ) ;
        w̄^T = x̄^T · ( ∂H/∂W ) · ( 0_{p×n} , I_{p×p} ) ;
                    ( 0_{n×p} )
        ȳ = euler_adjoint(ȳ, t − 1); /* linear recursion */
        result‾ = w̄ + ȳ;
    else
        result‾^T = x̄^T · ( dZ(1)/dZ(0)   dZ(1)/dW ) ;
                          ( I_{n×n}       0_{n×p}  )
    end if
    return(result‾);
end function
```

Figure 5: Recursive Evaluation of Adjoint for an Euler Scheme

The recursive evaluation of an adjoint associated with $\overline{Z}$ is shown in Figure 5. While all the extended state vectors are of length $2n$, the exploitation of the matrix structure apparent in (9) implies that the amount of linear algebra work per time step remains, to first order, unchanged. However, since we now have a linear recursion, the overall number of matrix-vector multiplications is $2(T-1)$. To compute $\dfrac{d\,r}{d\,[Z(0),W]}$, we would then execute the algorithm in Figure 3, followed by an invocation of

$$\text{euler\_adjoint}\left(\begin{pmatrix} \dfrac{d\,r}{d\,Z(T)} & , & 0_{1\times n} \end{pmatrix}, T-1 \right).$$

To summarize, the work required for the generation of the timestep Jacobians, as illustrated in Figure 3, is a fixed multiple (independent of n) of the work to evaluate H. This holds with respect to both runtime and memory, since we employ a sparse variant of the forward mode with predictable memory requirements. The accumulation scheme in Figure 5, in contrast, is an incarnation of the reverse mode of automatic differentiation for an Euler scheme at a level where we consider $\overline{H}$ to be an elementary operator. The second-order leapfrog difference scheme was converted to a first-order Euler scheme by adjoining successive time steps. Efficiency was maintained by exploiting the special structure of the Jacobians of the resulting Euler scheme.

4 Experimental Results

We implemented the pseudo-adjoint approach on an IBM SP computer and a network of Sun SPARCstations. For the problems shown in Table 1, we computed the sensitivity of the value of $Z(T)$ with respect to interior gridpoint number 101, that is, $r = e_{101}$, where e_i is the ith canonical unit vector, for a period of $T = 60$ time steps.

Table 1: Shallow Water Equations Models

Grid Size	n	p
11×11	$3 * 11 * 11 = 363$	4
16×16	$3 * 16 * 16 = 768$	4
21×21	$3 * 21 * 21 = 1323$	4

We computed the sparse Jacobian shown in Figure 2 using SparsLinC or, alternatively, using SparsLinC in the first time step to determine the sparsity structure and then using the compressed Jacobian approach in subsequent time steps. The latter approach is feasible here because the sparsity pattern does not change for $t > 2$. In contrast, continuous use of SparsLinC could accommodate varying sparsity patterns.

4.1 Serial Implementation

Table 2 shows (1) the overall runtime of the black-box ADIFOR approach, (2) the time spent by the pseudo-adjoint approach in the computation of the sparse Jacobian, using either the SparsLinC or the compressed Jacobian approach, and (3) the time spent in matrix-vector multiplications. In the serial implementation, the initial function computation loop shown in Figure 3 is not present. The code does a fair amount of copying, and these memory accesses account for the considerable difference between

the Jacobian computation and matrix-vector multiplication times. Table 3 contains a summary of the memory requirements of these runs (the numbers for the IBM and Sun platforms are similar).

Table 2: Serial Runtime for Pseudo-Adjoint Approach (in seconds)

	IBM SP Node			SPARCstation 5
	11×11	16×16	21×21	11×11
Black-Box ADIFOR	4.24	36.68	71.98	26.63
Pseudo-adjoint Using SparsLinC				
Total	6.90	27.52	69.46	13.94
Jacobians	4.90	17.77	42.98	12.26
Pseudo-adjoint Using Compressed Jacobian				
Total	4.53	18.50	51.31	9.09
Jacobians	1.93	8.70	21.51	6.55
Matrix-Vector Mult	0.53	1.62	2.21	0.15

Table 3: Serial Memory Requirements for Pseudo-Adjoint Approach (in Mbytes)

	11×11	16×16	21×21
Black-Box ADIFOR	4.70	18.82	53.31
SparsLinC	4.67	12.23	27.15
Compressed Jacobian	4.81	12.46	27.50

We see that compared with the black-box ADIFOR approach, which propagates $n+p$ derivatives for each of the n grid points and hence has an $O(n^2)$ complexity with respect to both memory and runtime, the pseudo-adjoint approach shows a much weaker rate of growth since both the computation of the individual time-step Jacobians and that of the sparse matrix-vector multiplies require $O(n)$ memory and floating-point operations. For example, while the pseudo-adjoint approach is slower than black-box ADIFOR for the 11×11 problem on the IBM, it beats ADIFOR on the 21×21 problem. We fare even better on the SPARCstation, where vector operations are, in comparison with scalar code, not executed as fast as on the IBM. For the larger problems, we also observe a memory savings of a factor of almost two.

The code for the shallow water equations model problem is based on a five-point stencil and updates the east and west wind components as well as the geopotential at a cost of 59 flops per gridpoint. To simulate performance for a computationally more intensive problem, where the computation of the sparse Jacobian corresponding to one time step requires more work, we executed the code for one time step from 1 (the original

problem) to 16 times. Thus, we increased the work required for black-box ADIFOR as well as the timestep Jacobian by the repetition factor, while the cost of the matrix-matrix multiply, the number of memory moves, and the memory requirements of the code remain unchanged. The resulting computational behavior is shown in Figure 6.

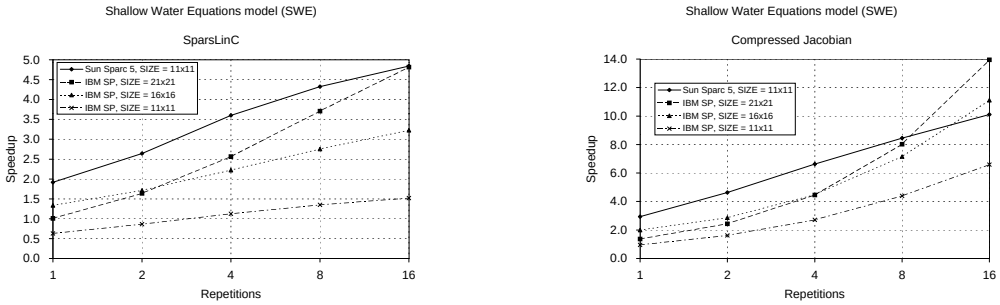

Figure 6: Serial Speedup of Pseudo-adjoint Scheme

Figure 6 shows that we can in fact obtain considerable speedup compared with a black-box application of ADIFOR, in particular as the work per nonzero entry in the timestep Jacobian increases. Computing Jacobians via SparsLinC is slower than via the compressed Jacobian approach, so we observe less improvement for the "all-SparsLinC" approach. In all cases, the percentage of time spent on the sparse matrix-vector multiplications is less than 2% of the total time.

4.2 Parallel Implementation

The parallel implementation of the pseudo-adjoint scheme is shown in Figure 7. We have three kind of processes. The *timestep Jacobian processes* implement the algorithm depicted in Figure 3, computing the timestep Jacobians at selected time steps and sending them to the *matrix-vector multiplication process*, which receives them and stores them in order. When all individual timestep Jacobians have been received, the matrix-vector multiplication process performs the high-level reverse mode accumulation described in Figure 5. The optional *task manager process* is employed in a heterogeneous system where the round-robin approach of Figure 3 would lead to severe load imbalance. This process keeps track of which timestep Jacobians have been computed and dynamically assigns them as Jacobian processes become available. The public-domain MPICH [17] implementation of the MPI message-passing interface was employed as parallel transportation layer.

Figures 8 and 9 shows the performance of the 11×11 problem using the SparsLinC

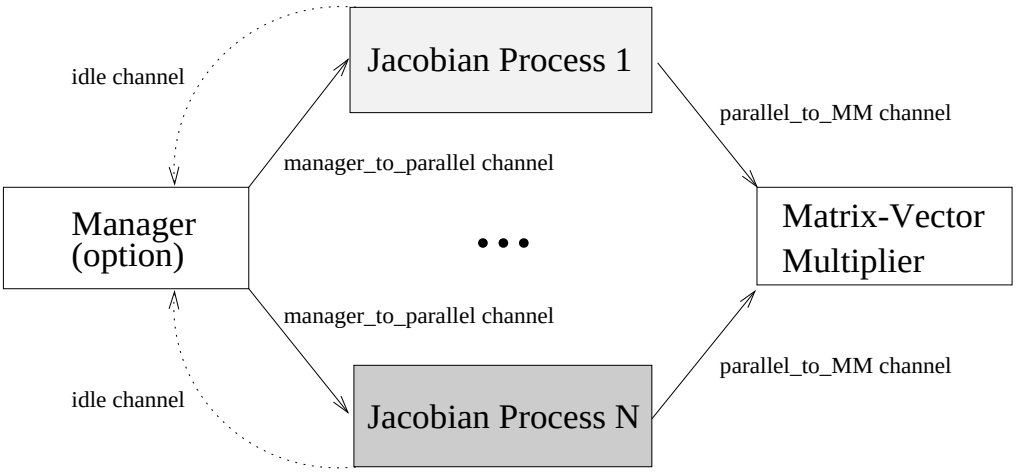

Figure 7: System Design of Time-Parallel Computation

approach with a repetition factor of 16. We visualized the computation with the Upshot [18] tool. In this example, we employ four processors. The matrix-vector multiplication and task manager tasks are unified in one process (number 0) that is located on the same node as one of the derivative slaves (number 1-4). This approach does not appreciably slow the derivative slave, since task management requires virtually no work, and the matrix-vector process does not begin computing before all derivative slaves have finished. In the top bar, "Compute_Der" indicates that a process is computing a timestep Jacobian, "Compute_Fun" indicates that it is just running the simulation (this happens only at the beginning), "Compute_Mat" indicates the series of matrix-vector multiplies corresponding to Figure 1 (which is executed by process 0 only at the end and is hardly visible because of to its short duration), "Receive" indicates that a process is waiting to receive a message, and "Send" indicates the time spent in sending a message. Figures 8 and 9 show that despite the conceptually equal-sized jobs, severe load imbalance can occur, however, a dynamic assignment of time steps for Jacobian computation leads to a balanced computation.

In contrast, Figure 10 shows the performance of the compressed Jacobian approach with a repetition factor of 16 on five processors of the IBM SP. Here we actually allocate a separate node for the matrix-vector multiplication process, and we do not employ a task manager. Note the two long "Compute_Der" blocks at the beginning of process 1, while there is only one such block for the other Jacobian slaves. This is due to the fact that the sparsity pattern changes between iteration 1 and subsequent iterations, and the process that computes the Jacobian for time step 1 needs to invoke SparsLinC twice.

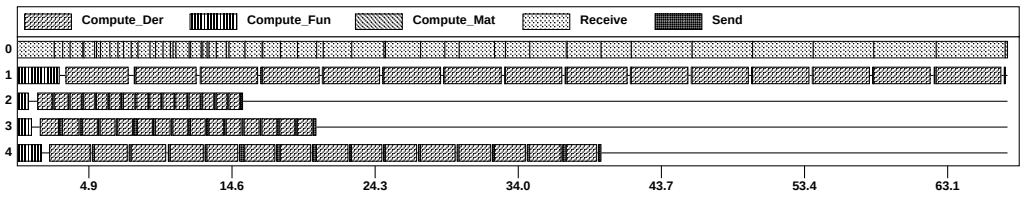

Figure 8: Performance of SparsLinC Approach without Task Manager on Sun Network

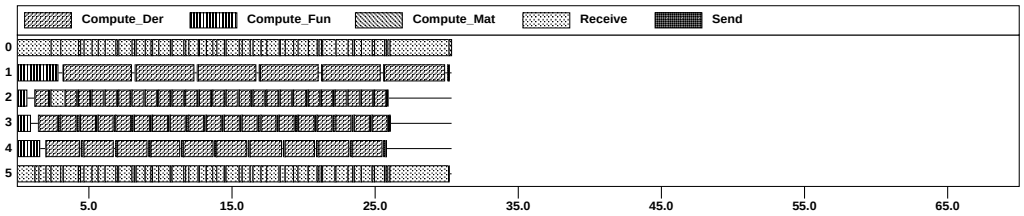

Figure 9: Performance of SparsLinC Approach with Task Manager on Sun Network

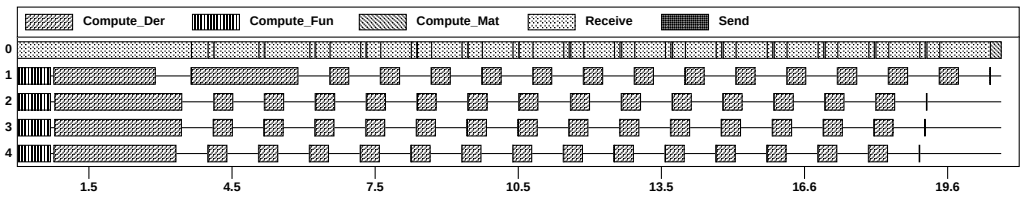

Figure 10: Performance of Compressed Jacobian Scheme on IBM SP

We see that in all these figures, the initial function computation and the matrix-vector accumulation, which are the serial parts of the computation, take very little time overall. Figure 11 shows the speedup we obtain on various numbers of processors on the IBM SP and SPARCstation networks using a repetition factor of 1 and 16. The more work that is done within one time step (the work that is performed in parallel), the better we do. Hence, speedup increases with problem size and with the number of repetitions. It trails off as there are not enough time steps to keep the derivative slaves busy, in particular in the compressed Jacobian scheme, where all processes need to invoke the relatively slow SparsLinC process only once at the beginning.

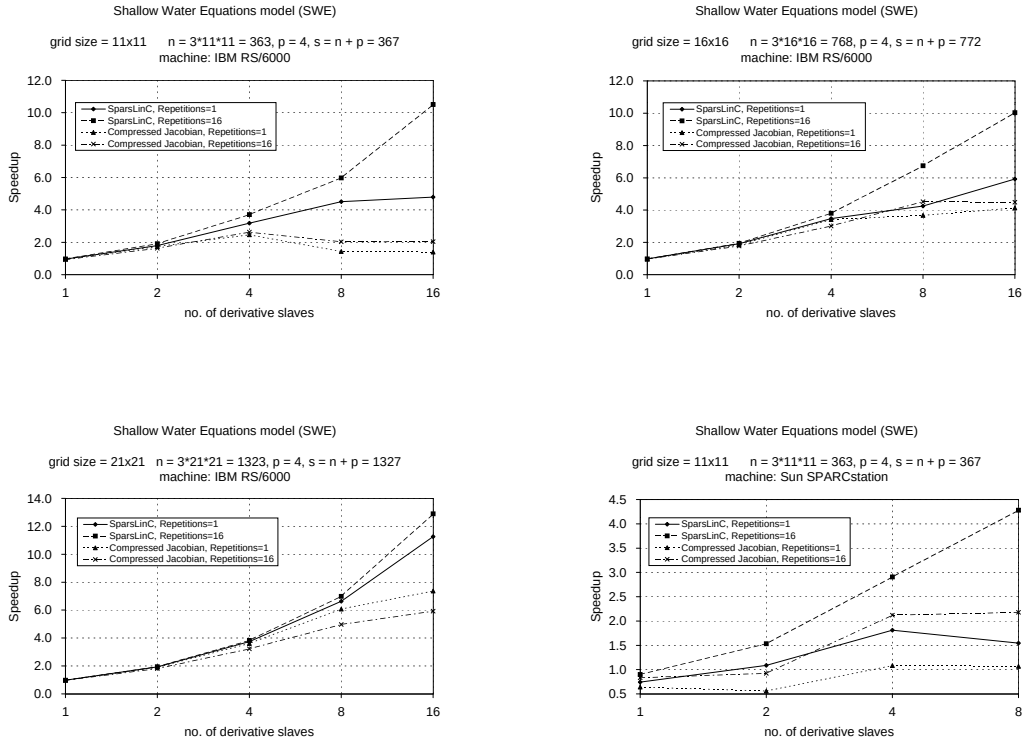

Figure 11: Parallel Speedup of Reverse Mode of Grid-Size: 11×11 for IBM SP, 16×16 for IBM SP, 21×21 for IBM SP, and 11x11 for Sun Network. We denote by `Sparse` the scheme of ADIFOR with full SparsLinC and `Compressed Jacobian` the hybrid scheme of ADIFOR with SparsLinC for the first time step and compressed Jacobian for the further time steps.

Note that, with respect to black-box ADIFOR, the speedup reported in Figure 11 is in addition to the one reported in Figure 6. Thus, for example, compared with black-box

ADIFOR, using eight processors with the compressed Jacobian approach for the 21×21 grid, runtime was reduced for the original problem from 72 seconds to 8.4, a speedup of 8.5, and for the problem with 16 repetitions runtime was reduced from 931 to 13.4 seconds, a speedup of almost 70. For the 11×11 problem on a four-node Sun network, this approach reduced runtime from 27 to 8.4 seconds (a speedup of 3.2) on the original problem, and for the problem with 16 repetitions runtime was reduced from 327 to 15.3 seconds, a speedup of 21. In all cases, the derivative values agreed to machine precision.

5 Conclusions

In this article, we showed how to compute a so-called adjoint for a leapfrog difference scheme with a combination of the forward and reverse modes of automatic differentiation. We developed a generic harness that employs the reverse mode of automatic differentiation at a level that considers the timestepping operator as an atomic operation, and we evaluated the Jacobian associated with a particular timestep with a forward-mode based tool, exploiting the sparsity of the underlying operator. Because of the associativity of the chain rule, Jacobians corresponding to different time steps can be evaluated in parallel. The resulting scheme exhibits the predictable floating-point and memory requirements of the forward mode, independent of the complexity and nonlinearity of the underlying operator, while still retaining the floating-point efficiency of the reverse mode.

The development of better automatic differentiation schemes is a current research area. Hybrid schemes that combine the forward and hybrid modes have been shown to hold considerable promise. For example, [10] shows that it can be advantageous to employ reverse-mode pieces inside a forward-mode-oriented driver. The paper [13] goes even further and considers "freestyle" automatic differentiation schemes on a computational graph representation of the program. The work we presented further shows the promise of approaches that exploit chain rule associativity and employ principles of automatic differentiation and automatic differentiation tools in a fashion that takes advantage of user insight at a high level. For example, our algorithms assume a leapfrog (or Euler) explicit difference scheme and a sparse timestep propagator H. The differentiation of H itself is handled by automatic differentiation. In our view, work on easily retargetable method-specific harnesses, such as the one we developed for a leapfrog scheme, and the use of automatic differentiation tools, which are rapidly improving, will go a long way toward "fast and easy" derivatives.

Acknowledgments

We thank David Zhi Wang of the Center for Analysis and Prediction of Storms at the University of Oklahoma for providing us with the code for the shallow water equation model.

References

[1] Proceedings of the 5th AIAA/NASA/USAF/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Panama City, Florida, American Association of Aeronautics and Aerospace Engineers, 1994.

[2] Brett Averick, Jorge Moré, Christian Bischof, Alan Carle, and Andreas Griewank. Computing large sparse Jacobian matrices using automatic differentiation. *SIAM Journal on Scientific Computing*, 15(2):285–294, 1994.

[3] Jochen Benary. Parallelism in the reverse mode. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 137–148, Philadelphia, 1996. SIAM.

[4] Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank. *Computational Differentiation: Techniques, Applications, and Tools.* SIAM, Philadelphia, 1996.

[5] Christian Bischof, Ali Bouaricha, Peyvand Khademi, and Jorge Moré. Computing gradients in large-scale optimization using automatic differentiation. Preprint MCS-P488-0195, Mathematics and Computer Science Division, Argonne National Laboratory, 1995. To appear in ORSA Journal of Computing.

[6] Christian Bischof, Alan Carle, Peyvand Khademi, and Andrew Mauer. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering*, 3(3):18–32, 1996.

[7] Christian Bischof, Peyvand Khademi, Ali Bouaricha, and Alan Carle. Efficient computation of gradients and Jacobians by transparent exploitation of sparsity in automatic differentiation. *Optimization Methods and Software*, 7(1):1–39, July 1996.

[8] Christian Bischof, Gordon Pusch, and Ralf Knoesel. Sensitivity analysis of the MM5 weather model using automatic differentiation. *Computers in Physics*, 10(6):605–612, 1996.

[9] Christian Bischof, Lucas Roh, and Andrew Mauer. ADIC — An extensible automatic differentiation tool for ANSI-C. Preprint ANL/MCS-P626-1196, Mathematics and Computer Science Division, Argonne National Laboratory, 1996.

[10] Christian H. Bischof and Mohammad R. Haghighat. On hierarchical differentiation. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 83–94, Philadelphia, 1996. SIAM.

[11] Christian H. Bischof and Po-Ting Wu. Exploiting intermediate sparsity in computing derivatives of a leapfrog scheme. Preprint ANL/MCS-P572-0396, Mathematics and Computer Science Division, Argonne National Laboratory, 1996.

[12] Ralf Giering. Tangent linear and adjoint model compiler, users manual. Unpublished Information, 1996.

[13] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1(1):35–54, 1992.

[14] Andreas Griewank and George Corliss. *Automatic Differentiation of Algorithms.* SIAM, Philadelphia, 1991.

[15] Andreas Griewank, David Juedes, and Jean Utke. ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software*, 22(2):131–167, 1996.

[16] José Grimm, Löc Pottier, and Nicole Rostaing-Schmidt. Optimal time and minimum space time product for reversing a certain class of programs. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation, Techniques, Applications, and Tools*, pages 95–106, Philadelphia, 1996. SIAM.

[17] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI – Portable Parallel Programming with the Message Passing Interface.* MIT Press, Cambridge, 1994.

[18] V. Herrarte and E. Lusk. Studying parallel program behavior with Upshot. Technical Report ANL-91/15, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[19] Jim E. Horwedel. GRESS: A preprocessor for sensitivity studies on Fortran programs. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 243–250. SIAM, Philadelphia, 1991.

[20] Ed. Oktay Baysal, editor. *CFD for Design Optimization, FED Volume 232, Proceedings of the 1995 ASME Int. Mech. Engr. Congress & Exposition.* 1995.

[21] Special Issue on Adjoint Applications in Dynamic Meterology. Tellus, 45a(5), 1993.

[22] Patrick Roache. *Computational Fluid Dynamics.* Hermosa Publishers, Albuquerque, NM, 2nd edition, 1996.

[23] Nicole Rostaing, Stephane Dalmas, and Andre Galligo. Automatic differentiation in Odyssee. *Tellus*, 45a(5):558–568, October 1993.

[24] Z. Wang, I. M. Navon, X. Zou, and F. LeDimet. A truncated Newton optimization algorithm in meteorology applications with analytic Hessian/vector products. *Computational Optimization and Applications*, 4:241–262, 1995.

[25] Zhi Wang. Variational data assimilation with 2-D shallow water equations and 3-D FSU global spectral models. Technical Report FSU-SCRI-93T-149, Florida State University, Department of Mathematics, December 1993.