A Matrix-Matrix Multiplication Approach to the Automatic Differentiation and Parallelization of Straight-Line Codes

H. M. Bücker [a], K. R. Buschelman [b], and P. D. Hovland [b]

[a] *Institute for Scientific Computing, Aachen University of Technology, 52056 Aachen, Germany.*

[b] *Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Ave, Argonne, IL 60439, USA.*

Abstract

A Straight-line code, which consists of assignment, addition, and multiplication statements, is an abstraction of a serial computer program to compute a function with n inputs. Given a serial straight-line code with N statements, we derive an algorithm that automatically evaluates not only the function but also its first-order derivatives with respect to the n inputs on a parallel computer. The basic idea of the algorithm is to marry automatic computation of derivatives with automatic parallelization of serial programs. The algorithm requires $O(M_N \log dN)$ scalar operations, where $O(M_N)$ is the time complexity of a parallel multiplication of two dense $N \times N$ matrices and d represents a measure of the complexity of the straight-line code. Although d can be exponential in N in the worst case, it tends to be only polynomial in N for many important problems.

Key words: Automatic differentiation, forward mode, automatic parallelization, arithmetic circuit.

1 Introduction

Given a serial computer program to compute a function, one can apply techniques of automatic differentiation to evaluate the function simultaneously with its first-order derivatives [6]. One can also parallelize a serial computer program automatically [8,9]. In this note, we show how these two concepts can be married. The resulting algorithm takes as input a serial code for a function and automatically evaluates the function and its first-order derivatives on a parallel computer. Parallel calculation of higher-order derivatives based on Taylor series expansion can be found in [7].

The algorithm described in this note is of theoretical interest and enhances our understanding of parallel automatic differentiation. It is not intended to represent practical issues involved in a particular implementation of such software. More practical issues in this field are discussed in [1–3,5].

The structure of this note is as follows. In Sec. 2, straight-line codes are specified as an abstraction of more complicated programs. For the parallel evaluation of the underlying function, a common representation of a straight-line code known as an arithmetic circuit is introduced. In Sec. 3, we show how the circuit can be adapted to a so-called augmented arithmetic circuit in order to include derivative information. In Sec. 4, transformations on the augmented arithmetic circuit are described that will be used for its parallel evaluation, which is described in Sec. 5. The resulting algorithm is illustrated by an example in Sec. 6.

2 Conversion of Straight-Line Code into an Arithmetic Circuit

A straight-line code is a finite sequence of elementary operations without loops, conditionals, branching, or subroutines. Straight-line codes may be considered an abstraction of more complicated programs. More precisely, they represent a trace of a particular run of a program provided specific values for the input variables are given. A straight line code has no branches or jumps of any type; that is, every loop is unrolled, every conditional statement is replaced by the appropriate branch, and every subroutine is inlined. This note is concerned with straight-line codes in which every statement is of one of the following three forms:

(i) $x_k \leftarrow c$
(ii) $x_k \leftarrow x_i + x_j$
(iii) $x_k \leftarrow x_i \cdot x_j$,

where x_i and x_j are previously defined variables and $c \in \mathbb{R}$ is a constant. For the sake of simplicity, we assume that, in a straight-line code, each variable has only one assignment. This assumption may lead to tremendous growth of the number of variables but preserves uniqueness of the left-hand sides.

Straight-line codes are commonly represented by graphs, known as arithmetic circuits. Formally, an *arithmetic circuit* is an edge- and node-weighted directed acyclic graph $G = (X, E, \rho, \sigma)$ with a set of nodes X and a set of edges E that are weighted by the functions ρ and σ, respectively. The set of nodes X is the disjoint union of three different kinds of nodes, namely, $X = L \,\dot\cup\, A \,\dot\cup\, M$, where L denotes the set of *L*eaves, A the set of *A*ddition nodes, and M the

$$
\begin{aligned}
x_1 &\leftarrow 6 \\
x_2 &\leftarrow 2 \\
x_3 &\leftarrow x_1 + x_2 \\
x_4 &\leftarrow x_2 + x_3 \\
x_5 &\leftarrow x_3 \cdot x_4 \\
x_6 &\leftarrow x_3 + x_5 \\
x_7 &\leftarrow x_4 \cdot x_5
\end{aligned}
$$

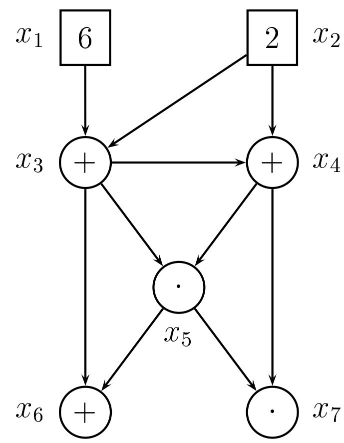

Fig. 1. Taking $(x_1, x_2) = (6, 2)$ as input, the straight-line code and its associated arithmetic circuit compute x_6 and x_7 given by (1) and (2), respectively.

set of Multiplication nodes. The nodes of the arithmetic circuit satisfy

$$
\begin{aligned}
\text{indegree}(l) &= 0, \qquad && \forall\, l \in L, \\
\text{indegree}(a) &> 0, \qquad && \forall\, a \in A, \\
\text{indegree}(m) &= 2, \qquad && \forall\, m \in M.
\end{aligned}
$$

Let N denote the number of statements of the straight-line code. With every statement of the straight-line code, a node is associated. Therefore, every left-hand side variable can be thought of as a node, and setting $X = \{x_1, x_2, x_3, \ldots, x_N\}$ is appropriate. All edges of the arithmetic circuit are directed away from leaves. There is an edge from node x_i to node x_j whenever x_i is an input to x_j. Both nodes and edges are weighted.

The following example, depicted in Fig. 1, is borrowed from [8]. Given a straight-line code, then one can construct the corresponding arithmetic circuit by associating a node in the circuit with every statement in the code. Notice that the node types are different and correspond to the three kinds of statements and that edges are used to propagate the appropriate input to an operation. The code and the arithmetic circuit given in the figure compute a function f satisfying $(x_6, x_7) = f(x_1, x_2)$, where

$$
\begin{aligned}
x_6 &= x_1^2 + 3x_1x_2 + 2x_2^2 + x_1 + x_2 \qquad (1) \\
x_7 &= x_1^3 + 5x_1^2x_2 + 8x_1x_2^2 + 4x_2^3. \qquad (2)
\end{aligned}
$$

In [8, Sec. 2.6], it is shown how the function f can be evaluated in a polylogarithmic number of parallel steps using a computer whose network architecture is a so-called three-dimensional mesh of trees. The algorithm given there is quite general in the sense of being able to handle additions and multiplications not only over $\mathbb{R}$ but over any commutative semiring where the terms “addition” and “multiplication” are interpreted appropriately. It consists of repeated applications of basic matrix and vector operations. The purpose of this note is to show how fast evaluation of the function *and* the first-order derivatives of that function can be accomplished. The algorithm given here is a modification of the algorithm given in [8] for a particular choice of the semiring. However, both the arithmetic circuit and the operations involved in evaluating the circuit need modifications, which we describe in the following sections.

3 Augmenting an Arithmetic Circuit with Derivative Information

Suppose that a straight-line code for the computation of a function f takes n independent variables as input and produces some dependent variables as output. Furthermore, assume that not only the function f evaluated at specific input values is sought but also its Jacobian evaluated at the same input. To this end, the arithmetic circuit sketched in Fig. 1 is augmented with derivative information. While the structure of the circuit remains unchanged, the node and edge weights are modified to propagate the derivative information. The resulting graph is called an *augmented arithmetic circuit.*

Without loss of generality, a straight-line code can be arranged such that all assignments of the form $x_i \leftarrow c_i$ are given at the beginning. Furthermore, let the first n constant assignments represent the input values for the function. For example, if $f(x) = x + 1$, the straight-line code evaluating f at $x = 7$ is given by

$$
\begin{aligned}
x_1 &\leftarrow 7 \\
x_2 &\leftarrow 1 \\
x_3 &\leftarrow x_1 + x_2,
\end{aligned}
$$

where $n = 1$ and, more important, the order of the first two assignments is determined by the above assumption. The constants involved in the function evaluation are associated with the leaves of the circuit, whereas the addition and multiplication operations introduce nodes of corresponding type. Moreover, calculation of derivatives with respect to n inputs gives rise to the propagation of gradient vectors with dimension n, as is reflected in the concept of doublets. The functions $\rho : X \to \mathbb{D}$ and $\sigma : E \to \mathbb{D}$ are used to denote node weights and edge weights, respectively, where the set $\mathbb{D} := \mathbb{R} \times \mathbb{R}^n$ is the set of *doublets.*

The use of doublets arises from the need to store intermediate values during the simultaneous computation of f and its Jacobian. A doublet is a pair, denoted by square brackets, with a *function part* and an associated *gradient part.* If $u = [u^f, \mathbf{u}^\nabla]$ is a doublet, then $u^f \in \mathbb{R}$ is used to refer to some intermediate scalar value involved in the evaluation of the function f, whereas $\mathbf{u}^\nabla \in \mathbb{R}^n$ refers to some intermediate gradient value involved in the derivative computation.

The symbols $\oplus$ and $\otimes$ denote addition and multiplication on doublets. More precisely, the addition of two doublets v and w is defined by $u = v \oplus w$, where

$$u^f = v^f + w^f \tag{3}$$
$$\mathbf{u}^\nabla = \mathbf{w}^\nabla + \mathbf{v}^\nabla; \tag{4}$$

that is, the separate addition of function and gradient part. The product of two doublets v and w is defined by $u = v \otimes w$, where

$$u^f = v^f \cdot w^f \tag{5}$$
$$\mathbf{u}^\nabla = v^f \mathbf{w}^\nabla + w^f \mathbf{v}^\nabla. \tag{6}$$

Here, the gradient part is defined in a product rule-like manner.

Note that both addition and multiplication on doublets are commutative and associative. Furthermore, the operation $\otimes$ distributes over $\oplus$ from left and from right; in other words, for all $u, v, w \in \mathbb{D}$ the relations

$$u \otimes (v \oplus w) = (u \otimes v) \oplus (u \otimes w) \qquad \text{and} \qquad (v \oplus w) \otimes u = (v \otimes u) \oplus (w \otimes u)$$

hold. Hence, the triplet $(\mathbb{D}, \oplus, \otimes)$ is a commutative semiring [10], and the algorithm given in [8] is applicable. The doublet $[1, \mathbf{0}_n]$ is the multiplicative identity element in $\mathbb{D}$, where $\mathbf{0}_n$ denotes the n-dimensional zero vector. The doublet $[0, \mathbf{0}_n]$ is the additive identity element as well as the multiplicative absorbent in $\mathbb{D}$. For the sake of brevity, the doublet $[0, \mathbf{0}_n]$ is hereafter referred to as the *zero doublet*.

Let $\mathbf{e}_i \in \mathbb{R}^n$ denote the ith Cartesian unit vector, and recall that c_i denotes the constants assigned at the beginning of the straight-line code. Then, the initial node and edge weights of the augmented arithmetic circuit are given by

$$\rho(x_i) = [c_i, \mathbf{e}_i], \qquad \forall\, x_i \in L \quad \text{for } i \le n, \tag{7}$$
$$\rho(x_i) = [c_i, \mathbf{0}_n], \qquad \forall\, x_i \in L \quad \text{for } i > n, \tag{8}$$
$$\rho(x_i) = [0, \mathbf{0}_n], \qquad \forall\, x_i \in A \,\dot{\cup}\, M, \tag{9}$$
$$\sigma(e) = [1, \mathbf{0}_n], \qquad \forall\, e \in E. \tag{10}$$

If a leaf corresponds to the ith input to the function, that is, a variable of the function whose derivative is to be evaluated, the gradient part of its doublet is initialized to the ith unit vector; otherwise, the gradient part is initialized to the zero vector. Addition and multiplication nodes are set to the zero doublet. Edges are weighted with the multiplicative identity element in $\mathbb{D}$. The initialized augmented arithmetic circuit related to the example given in Fig. 1

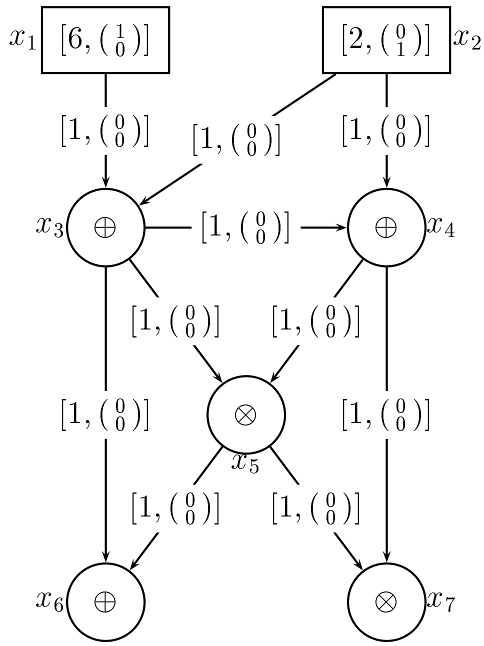

Fig. 2. Given the arithmetic circuit from Fig. 1, its augmented arithmetic circuit is initialized according to (7)–(10)

is depicted in Fig. 2, where, for the sake of clarity, addition and multiplication nodes are labeled reflecting their type rather than with their initial node weights, the zero doublet.

The algorithm to be presented in this note can be adequately described by means of linear algebra expressions involving matrices and vectors whose entries are modified throughout the course of the algorithm. Recall that N denotes the number of nodes of the augmented arithmetic circuit. Then, an $N \times N$ matrix of edge weights, W, is introduced. The (i, j) entry of this matrix is defined to be $\sigma(e_{i,j})$, the weight of the edge between node x_i and node x_j. If there is no edge between these nodes, the corresponding matrix entry is set to the zero doublet. Note that, from the above construction of the circuit, the matrix W is upper triangular with zero doublets along the diagonal. Its nonzero entries are initially given by (10). Furthermore, we introduce an N-dimensional vector of node weights, $\mathbf{v}$, whose ith component is given by $\rho(x_i)$, the weight of node x_i. This vector is initialized according to (7)–(9).

The complexity of the algorithm presented in this note will be described in terms of a parameter of the straight-line code and its augmented arithmetic circuit. It is useful to introduce this parameter here while having the circuit of Fig. 2 in mind. The *degree of a node* is defined inductively. The degree of a leaf is 1. The degree of an addition node is the maximum degree of its inputs. The degree of a multiplication node is the sum of the degrees of its inputs. For instance, the degree of node x_6 is 2 and the degree of node x_7 is 3;

notice the degree of the multivariate polynomials (1) and (2), respectively. The *degree of an (augmented) arithmetic circuit* is then the maximum degree of any node. For instance, the degree of the circuit depicted in Fig. 2 is 3. If a circuit with N nodes has long chains of multiplication nodes, its degree can be exponential in N; however, the degree may be polynomial in N for a large class of problems. Note that the degree of a circuit is the degree of the multivariate polynomial that this circuit computes.

4 Transformations on the Augmented Arithmetic Circuit

Upon instantiation of an augmented arithmetic circuit, the weights of the leaves are doublets, the first n of which can be thought of as inputs to the circuit. We shall evaluate the circuit by carrying forward these doublets using repeated application of three elementary procedures: MULT, SKIP, and ADD. After each iteration of applying these three procedures, the resulting graph is still an augmented arithmetic circuit with the same number of nodes. However, the weights of both nodes and edges may be modified. The type of a node may switch from a multiplication node to an addition node and from an addition node to a leaf. Similarly, an edge weight may change from a nonzero doublet to a zero doublet, hereafter referred to as the deletion of an edge, and one may change from a zero doublet to a nonzero doublet, creating an edge. Eventually, all edges will be deleted, and all nodes will become leaves with weights containing the desired function and derivative information.

The three procedures to be described operate simultaneously on all nodes of the circuit and will be illustrated by figures. In these figures, rectangles denote leaves; white circles are used for addition and multiplication nodes; and gray-shaded circles stand for nodes of any type, that is, leaves, addition, or multiplication nodes. The most straightforward of the three procedures is illustrated in Fig. 3. The procedure ADD evaluates those addition nodes x_k in parallel whose inputs are all leaves. Hence, the type of a node x_k is changed from an addition node to a leaf. The weight of the new leaf x_k is the sum of all input node weights, v_i, scaled by the input edge weights, w_i; thus

$$\rho(x_k) = \bigoplus_{i=1}^{s} (w_i \otimes v_i). \tag{11}$$

After the new weight of node x_k is assigned, all incoming edges are set to the zero doublet, that is, all incoming edges are deleted.

The application of ADD can be formulated in terms of the vector of node weights, $\mathbf{v}$, and the matrix of edge weights, W. Recall from the definition of $\mathbf{v}$ that the node weights, v_i, in Fig. 3 are the entries of $\mathbf{v}$ at position ℓ_i

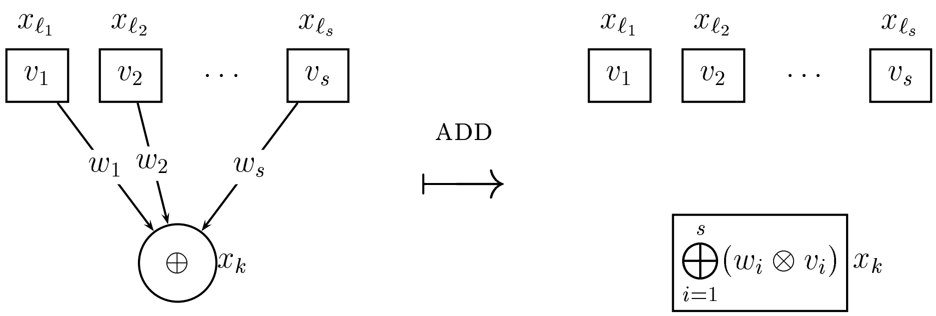

Fig. 3. Application of ADD on an addition node x_k whose inputs are all leaves evaluates x_k, changes its type to a leaf, and deletes all incoming edges.

for $i = 1, 2, \ldots, s$. Similarly, from the definition of W, the edge weights, w_i, are the entries of column k of W at position ℓ_i for $i = 1, 2, \ldots, s$. An interpretation of the transformation of the circuit based on simultaneously applying (11) to all nodes x_k whose inputs are all leaves is therefore as follows: the vector of node weights is updated by the matrix-vector multiplication $\mathbf{v} \leftarrow W^T\mathbf{v}$ where additions and multiplications are executed on doublets; in addition, the matrix of edge weights is modified only at certain entries. Consequently, applying ADD is no harder than computing a matrix-vector multiplication, $W^T\mathbf{v}$, in parallel.

The procedure MULT is used to simultaneously handle multiplication nodes and is depicted in Fig. 4. Only those multiplication nodes that have at least one leaf as input are transformed by MULT; that is, any multiplication node that has no input from a leaf is simply ignored in this transformation. Assume that one input of a multiplication node x_k is a leaf, say x_i, and one is an arbitrary node, say x_j. Then, the edge from x_i to x_k is removed, the edge from x_j to x_k is weighted by the weight v_i of the leaf x_i, and the type of x_k is changed from a multiplication node to an addition node. We note that if both inputs are leaves, a rule is required to determine which node is removed. We have chosen to remove the edge between x_i and x_k, where $i < j$, but other rules are possible.

The effect of MULT in terms of $\mathbf{v}$ and W is as follows. Since node weights are not modified, the vector $\mathbf{v}$ remains unchanged. The deletion of edges and the assignments from node weights to edge weights correspond to an update of W for specific entries.

The circuit can be evaluated completely by applying ADD and MULT in an alternating fashion. However, long addition chains in the circuit would be evaluated rather inefficiently because ADD can be applied only to nodes whose inputs are all leaves. Hence, partial sums occurring in the evaluation of long addition chains are carried forward linearly with the length of the chains. It would be desirable to collapse long addition chains into chains of about half the length to enable logarithmic complexity in evaluating long addition chains. Therefore, a way of making available the input of an addition node to

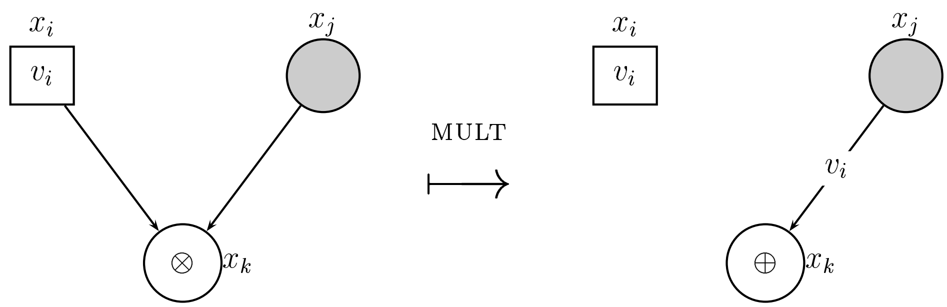

Fig. 4. Application of MULT on a multiplication node x_k with at least one leaf as input deletes the incoming edge from the leaf, transfers the leaf's weight to the edge from the arbitrary node to x_k, and changes x_k's type to an addition node.

its predecessor in the circuit is sought. Roughly speaking, that addition node is skipped over. The procedure SKIP serves that purpose by transforming those addition nodes that themselves point to addition nodes, as is shown in Fig. 5. This procedure does not evaluate any node but rearranges edges so that the next application of ADD may evaluate significantly more addition nodes. For each pair of addition nodes x_j and x_k, the procedure SKIP removes the edge from x_j to x_k and introduces new edges from each of x_j's input nodes x_{ℓ_i} to node x_k for $i = 1, 2, \ldots, s$. A new edge from x_{ℓ_i} to x_k is weighted by $w_i \otimes w$, where w is the weight of the edge being removed.

Similar to MULT, the procedure SKIP does not change the vector of node weights. To see the effect of SKIP on the matrix of edge weights, W, whose entries will be denoted by w_{ik} in the following discussion, we define the matrix of edge weights linking addition nodes exclusively. Let $W^{\oplus} = (w^{\oplus}_{ik})$ be this

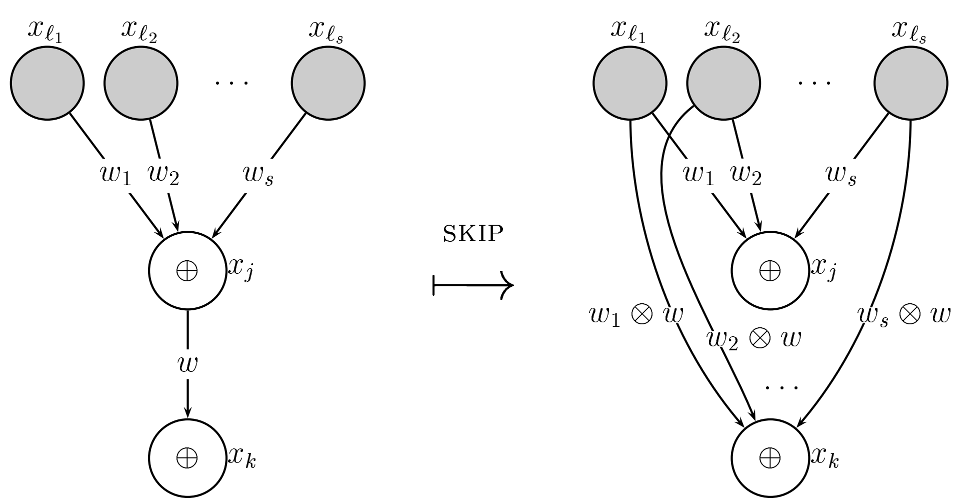

Fig. 5. Application of SKIP to a pair of addition nodes x_j and x_k deletes the edge from x_j to x_k and introduces a new edge from each of x_j's inputs to x_k that is weighted by the product of the corresponding input edge weight and the weight of the edge being removed.

matrix with entry

$$w_{ik}^{\oplus} = \begin{cases} w_{ik} & \text{if nodes } x_i \text{ and } x_k \text{ are addition nodes} \\ [0, \mathbf{0}_n] & \text{otherwise.} \end{cases}$$

An update of the form $W \leftarrow W - W^{\oplus}$ then represents the simultaneous removal of all edges between any pair of addition nodes. Consider now an arbitrary entry w_{ik} of the matrix of edge weights representing the edge from node x_i to node x_k and study w_{ik}'s transformation by the procedure SKIP. If x_k is not an addition node the entry w_{ik} remains unchanged. If x_k is an addition node and there is another addition node x_j that points to x_k with edge weight $w_{jk}^{\oplus}$ and there is an edge from x_i to x_j, then the new edge between x_i and x_k introduced by SKIP is weighted by $w_{ij} \otimes w_{jk}^{\oplus}$. Summing over all possible addition nodes x_j leads to a term $WW^{\oplus}$ so that, in summary, the update of the matrix of edge weights for the application of SKIP is represented by

$$W \leftarrow W - W^{\oplus} + WW^{\oplus},$$

where subtraction, addition, and multiplication of matrices are to be understood in the usual sense but operating on doublets rather than on scalars. Therefore, applying SKIP is no harder than computing a parallel matrix-matrix multiplication.

5 Parallel Evaluation of the Augmented Arithmetic Circuit

The three transformations introduced in the preceding section are now used in a parallel algorithm to evaluate the function associated to the given straight-line code along with its Jacobian evaluated at the same input. We have already mentioned that the augmented arithmetic circuit can be evaluated by applying ADD and MULT in an alternate fashion. The purpose of SKIP is to rearrange the edges of the circuit, enabling the evaluation of as many addition nodes as possible; therefore, SKIP can be regarded as a preprocessing step for ADD. The complete algorithm follows from properly arranging the three transformations:

Algorithm 1

repeat
  apply procedure MULT
  apply procedure SKIP
  apply procedure ADD
until circuit is evaluated

Parallelism is obtained by using parallel implementations of the linear algebra operations corresponding to the three procedures. The time complexity of the algorithm depends on the number of iterations necessary to evaluate all nodes of the augmented arithmetic circuit and is given by the following result.

Theorem 1 *Given a (serial) straight-line code with N statements and associated augmented arithmetic circuit of degree d, the (parallel) algorithm to evaluate and carry forward a gradient of each intermediate variable x_i with respect to n input variables simultaneously with the evaluation of x_i itself requires $O(nM_N \log dN)$ scalar operations, where $O(M_N)$ is the time complexity of a parallel multiplication of two dense $N \times N$ matrices on a given parallel architecture.*

PROOF. The above algorithm is a modification of the algorithm given in [8, Sec. 2.6] with a particular choice of the semiring. Specifically, the number of iterations within Alg. 1 needed to evaluate the circuit is the same, namely, $O(\log dN)$. The overall complexity of a single iteration within Alg. 1 is given by the matrix-matrix multiplication $WW^{\oplus}$ whose time complexity is $O(M_N)$ provided the entries of W are taken from $\mathbb{R}$ (i.e., scalar entries). The result then follows from observing that, by (3)–(6), additions and multiplications on doublets require at most $3n + 1$ scalar operations on $\mathbb{R}$. $\square$

An additional level of parallelism can be exploited to remove the dependence on n in the above count. In particular, one can decompose a doublet consisting of a scalar function part and an n-dimensional gradient part into n "reduced" doublets each having a scalar function part and a scalar gradient part. Hence, one can replicate the function information across n different collections of processors by storing reduced doublets on each collection of processors. By simultaneously performing n matrix-matrix multiplications based on these reduced doublets, one achieves $O(M_N \log dN)$ at the cost of increasing the number of processors by a factor of n which is assumed in the remainder of this section.

In the worst case, the degree d can be exponential in N leading to a running time of $O(M_N N \log N)$. For many problems, d is polynomial in N, leading to a running time of $O(M_N \log N)$.

The time complexity of matrix-matrix multiplication $O(M_N)$ is strongly dependent on the computer architecture used. Dekel, Nassimi, and Sahni have shown that, with a PRAM model, this operation can be performed with a running time of $O(\log N)$ requiring $O(N^3)$ processors [4]. This yields a lower bound on the time complexity for computing derivatives: $O(N \log^2 N)$ in the worst case, and simply $O(\log^2 N)$ in many important cases. In the case where the number of processors, p, is significantly less than N, matrix-matrix multiplication has a running time of $O(N^3/p)$, yielding corresponding complexities.

6 Example

The parallel algorithm is illustrated by an example showing how function and simultaneous derivative evaluation is carried forward through the augmented arithmetic circuit. The example is based on the straight-line code shown in Fig. 1. The corresponding augmented arithmetic circuit with leaves and edges initialized by doublets is depicted in Fig. 2. The modifications of node and edge weights during the course of Alg. 1 is given in Fig. 6. For simplicity, edges weighted by multiplicative identity element on doublets, $[1, \binom{0}{0}]$, are represented in this figure without label, and those edges weighted by the zero doublet are not drawn.

The algorithm begins with the procedure MULT, but since none of the multiplication nodes has at least one leaf as input, this operation does nothing. The first application of the procedure SKIP finds two pairs of addition nodes. Considering the first pair x_3 and x_4, notice that the edge between these two nodes is deleted and two new edges pointing to x_4 are introduced, one from x_1 and another from x_2. Both new edges are weighted by the product of multiplicative identity elements $[1, \binom{0}{0}] \in \mathbb{D}$. The second pair of addition nodes is x_3 and x_6. The edge connecting these nodes is removed, and two new edges from x_1 and x_2 are created that both point to x_6.

The first application of ADD evaluates the nodes x_3 and x_4 whose types are changed to leaves. Additionally, all their incoming edges are deleted. Notice that the addition node x_6 is not involved in the first ADD because one of its inputs, x_5, is not a leaf.

The following MULT has an effect on x_5 and x_7 because these nodes both have at least one leaf as input. Both nodes are changed to addition nodes, and the edges are transformed as follows. One input of the node x_7 is a leaf and one is not. As a result, the incoming edge from the leaf x_4 is removed, and the incoming edge from the nonleaf x_5 is assigned a weight equal to that of x_4. For the node x_5, both inputs are leaves. Following our heuristic, we remove the incoming edge from x_3 and assign the weight of x_3 to the remaining incoming edge from x_4.

Through the course of applying SKIP for the second time, two pairs of nodes, x_5 and x_6 as well as x_5 and x_7, are encountered. The outgoing edges from x_5 are deleted, and two new outgoing edges from x_4 are introduced, one to x_6 and another to x_7. The weights of the new edges are given by multiplying the weight of the incoming edge, $[8, \binom{1}{1}]$, with those of the corresponding edges being removed.

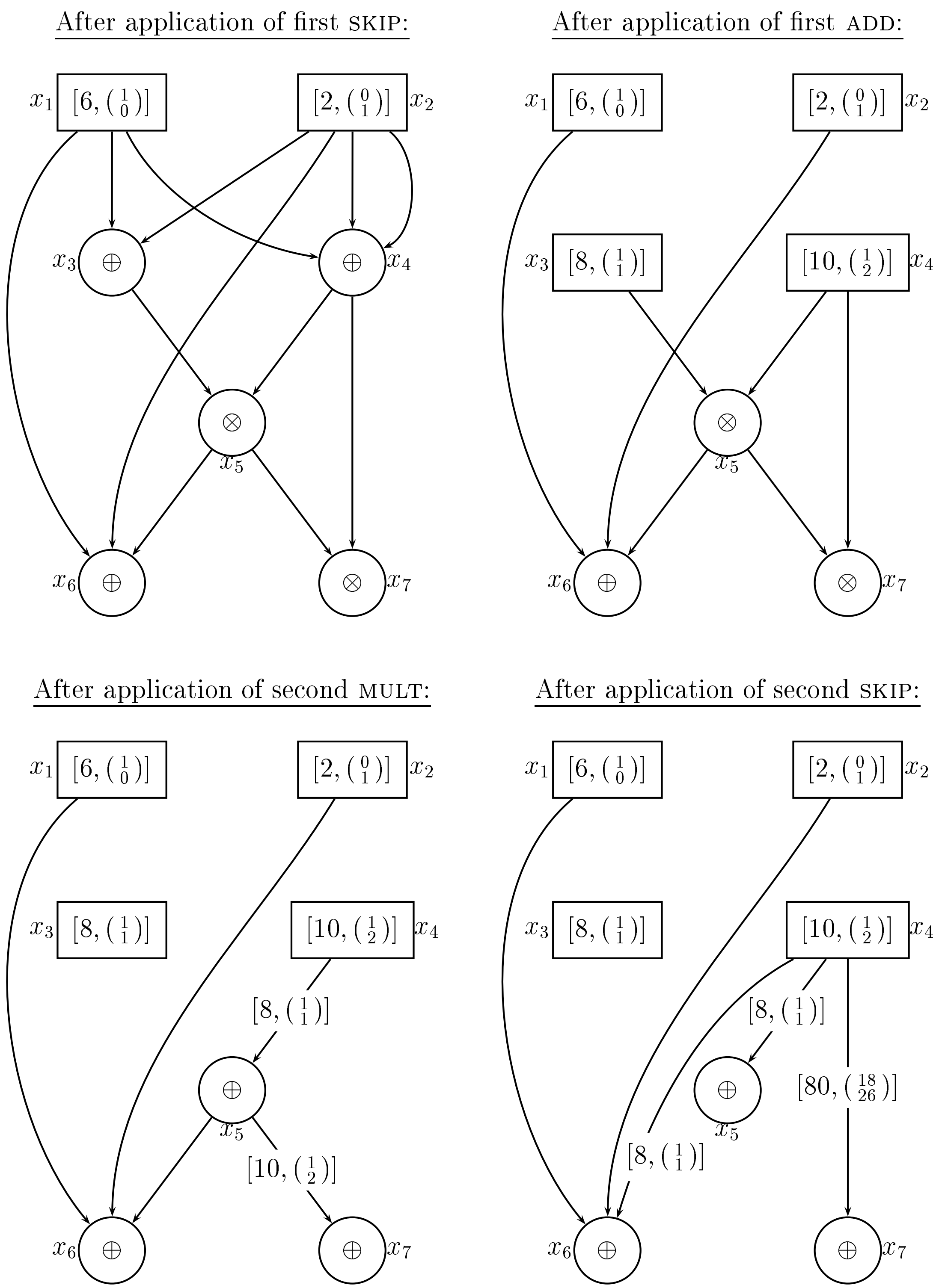

Fig. 6. Application of Alg. 1 to the augmented arithmetic circuit whose initialization is shown in Fig. 2. The result is the function given by (1) and (2) and its Jacobian, both evaluated at $(x_1, x_2) = (6, 2)$.

The following application of ADD, which is not shown in the figure, completes the computation by evaluating node x_6 as the doublet

$$[6, (\begin{smallmatrix}1\\0\end{smallmatrix})] \oplus [2, (\begin{smallmatrix}0\\1\end{smallmatrix})] \oplus \Big([8, (\begin{smallmatrix}1\\1\end{smallmatrix})] \otimes [10, (\begin{smallmatrix}1\\2\end{smallmatrix})]\Big) = [88, (\begin{smallmatrix}19\\27\end{smallmatrix})];$$

calculating node x_7, which is the doublet

$$[80, (\begin{smallmatrix}18\\26\end{smallmatrix})] \otimes [10, (\begin{smallmatrix}1\\2\end{smallmatrix})] = [800, (\begin{smallmatrix}260\\420\end{smallmatrix})];$$

and computing node x_5, which is unnecessary because it corresponds to an intermediate value and not to an output of the function.

These results agree with those obtained from analytically differentiating (1) and (2) and evaluating them at the same input, namely,

$$\begin{array}{lll} x_6\big|_{(6,2)} = 88, & \left.\dfrac{\partial x_6}{\partial x_1}\right|_{(6,2)} = 19, & \left.\dfrac{\partial x_6}{\partial x_2}\right|_{(6,2)} = 27, \\ x_7\big|_{(6,2)} = 800, & \left.\dfrac{\partial x_7}{\partial x_1}\right|_{(6,2)} = 260, & \left.\dfrac{\partial x_7}{\partial x_2}\right|_{(6,2)} = 420. \end{array}$$

7 Summary

Given a function in the form of a serial straight-line code, one can compute derivatives of this function in a parallel and automatic fashion. The key is to marry automatic differentiation and automatic parallelization. The algorithm for this task is derived from a representation of the straight-line code in terms of an arithmetic circuit. The arithmetic circuit is augmented with derivative information. The function and its derivative are evaluated by transforms on the arithmetic circuit. These transformations, in turn, are described by basic linear algebra operations whose parallel execution leads to the time complexity $O(M_N \log dN)$, where $O(M_N)$ is the time complexity of a parallel multiplication of two dense $N \times N$ matrices and d is the degree of the arithmetic circuit.

Acknowledgements

We thank Gail Pieper for her suggestions, which improved the readability of this manuscript. This work was performed while the first author was visiting the Mathematics and Computer Science Division, Argonne National Laboratory, USA. The work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

References

[1] J. Benary, *Parallelism in the reverse mode*, in Computational Differentiation: Techniques, Applications, and Tools, M. Berz, C. Bischof, G. Corliss, and A. Griewank, eds., Philadelphia, 1996, SIAM, pp. 137–148.

[2] C. Bischof, A. Griewank, and D. Juedes, *Exploiting parallelism in automatic differentiation*, in Proceedings of the 1991 International Conference on Supercomputing, E. Houstis and Y. Muraoka, eds., Baltimore, Md., 1991, ACM Press, pp. 146–153.

[3] C. H. Bischof, *Issues in parallel automatic differentiation*, in Automatic Differentiation of Algorithms, A. Griewank and G. Corliss, eds., Philadelphia, PA, 1991, SIAM, pp. 100–113.

[4] E. Dekel, D. Nassimi, and S. Sahni, *Parallel matrix and graph algorithms*, SIAM Journal on Computing, 10 (1981), pp. 657–675.

[5] H. Fischer, *Automatic differentiation: Parallel computation of function, gradient and Hessian matrix*, Parallel Computing, 13 (1990), pp. 101–110.

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

[7] E. Kaltofen and M. F. Singer, *Size Efficient Parallel Algebraic Circuits for Partial Derivatives*, in IV International Conference on Computer Algebra in Physical Research, D. V. Shirkov, V. A. Rostovtsev, and V. P. Gerdt, eds., Singapore, 1991, World Scientific, pp. 133–145.

[8] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees and Hypercubes*, Morgan Kaufmann, San Mateo, 1992.

[9] G. L. Miller, V. Ramachandran, and E. Kaltofen, *Efficient parallel evaluation of straight-line code and arithmetic circuits*, SIAM Journal on Computing, 17 (1988), pp. 687–695.

[10] A. Rosenfeld, *An Introduction to Algebraic Structures*, Holden-Day, San Francsico, 1968.