High Order Automatic Differentiation with MPI

[Extended Abstract]

Mu Wang
Purdue University
305 N. University Street
West Lafayette, Indiana, USA
wang970@purdue.edu

Alex Pothen
Purdue University
305 N. University Street
West Lafayette, Indiana, USA
apothen@purdue.edu

ABSTRACT

Automatic Differentiation (AD) is an algorithmic technique for augmenting computer programs to compute derivatives accurately and efficiently. AD has two major modes: forward and reverse. Previous work on parallel reverse mode AD algorithms mainly focus on first order. In those work, the communication is considered as a remote value assignment. So the communication is also reversed when evaluating derivatives. Here, we describe a new parallel Reverse mode algorithm to compute higher order derivative for an MPI program. The innovation is that in a parallel reverse mode AD algorithm, a Send/Recv pair should not be considered only as a remote value assignment, but as dummy independent/dependent variables. The innovation permits communication to be treated analogous to computation. Hence communication does not need to be reversed, but can be reperformed in a separate phase. So complications created by reversing the communications is avoided.

Categories and Subject Descriptors

G.1.0 [**General**]: Parallel algorithms; G.1.4 [**Quadrature and Numerical Differentiation**]: Automatic differentiation

General Terms

Theory

Keywords

Automatic Differentiation, Parallel Computing, MPI

1. BACKGROUND

Automatic Differentiation or *Algorithmic Differentiation* (AD) is a technology for analytically (thus accurately within machine precision) computing the derivatives of a function encoded as a computer program. AD relies on the premise that the execution of any computer-coded function, regardless of how complex it is, can be decomposed into a finite sequence of elementary functions/operators or intrinsic functions. For each type of the elementary function, the derivatives can be evaluated directly via essentially a "table look-up". The derivatives of each elementary function are retrieved via essentially a "table look-up". And the derivatives of the objective function can be evaluated by applying chain rule of calculus on the derivatives of the elementary functions.

In general, the objective function can be either scalar or vector. In this paper, we only focus on the scalar case. For the vector case, we can simply make duplicated data structures for each dependent variables.

Let $\mathbf{y} = \mathbf{F}(\mathbf{x})$, $\mathbf{x} \in \mathcal{R}^{\mathbf{n}\times\mathbf{1}}$ and $\mathbf{y} \in \mathcal{R}$ denote a scalar objective function of real values. Following the notations of Griewank and Walther [1], both here and elsewhere in the paper, let $v_{1-n}, \cdots, v_0$ denote the *independent variables* ($\mathbf{x}$). The execution of the objective function can then be decomposed as:

$$\textbf{for } i = 1, 2, \cdots, l$$
$$v_i = \varphi_i(v_j)_{\{v_j : v_j \prec v_i, 1-n \le j < i\}}.$$

where each $v_i = \varphi_i(v_j)_{\{v_j : v_j \prec v_i\}}$ represents an elementary function, and $v_j \prec v_i$ denotes that variable v_i directly *depends* on variable v_j. That is, in each step, v_i is the result of the elementary function φ_i, which takes v_j where $v_j \prec v_i$ as operands or parameters (for intrinsics). And the set $\{v_j : v_j \prec v_i\}$ contains all precedents of v_i. We call each $v_i = \varphi_i(v_j)$ a *Single Assignment Code* (SAC). The last variable in the sequence of elementary functions, v_l, holds the return value of the objective function, the only *dependent variable*. Evaluating the objective function $\mathbf{F}$ is, thus, equivalent to evaluating the SAC sequence.

1.1 First Order AD: Forward mode and reverse mode

To evaluate the first order derivative, i.e, the Jacobain of the objective function, we can simple apply the first order chain rule on the SAC sequence. Since the first order chain rule is associative, we can evaluate the derivatives in the same order as the SAC sequence is evaluated, or in an opposite order [1, 4]. The former yields the *forward mode* of AD and the latter yields the *reverse mode* of AD. In this work, we only focus on reverse mode.

Fig 1 gives the flowchart of the incremental reverse mode of AD. No matter how many independent variables there are, the reverse mode always computes all the components in the

Algorithm: The Incremental Reverse Mode

Input: an initial adjoint $\bar{v}_l = 1$

Output: $\bar{v}_{1-n} = \frac{\partial v_l}{v_{1-n}}, \cdots, \bar{v}_0 = \frac{\partial v_l}{\partial v_0}$

Set: $\bar{v}_{1-n} = \cdots = \bar{v}_0 = \bar{v}_1 = \cdots = \bar{v}_{l-1} = 0$

for $i = 1, \cdots, l$ **do**

$w = \bar{v}_i$; $\bar{v}_i = 0$

for each $v_j \prec v_i$ **do**

$\bar{v}_j += \frac{\partial \varphi_i}{\partial v_j} w$

Figure 1: The Incremental Reverse Mode

gradient vector directly in one round. The values $\bar{v}_i$ are also called *adjoints*.

It's important to notice that in the reverse mode, the SAC sequence is traversed in an opposite order as it's evaluated during the evaluation of the objective function. It means that the reverse mode requires that the information of the SAC sequence is stored externally. So that to read each SAC in the reverse order is possible. The stored information of the SAC sequence is also called a trace of the objective function. When the objective function is very complex, the tracing may become the bottleneck for reverse mode. Many techniques have been used to reduce the tracing overhead, for example, check pointing.

2. HIGH ORDER REVERSE ALGORITHM FOR MPI

In this section, we will briefly describe how we derive our high order reverse mode algorithm for `MPI`.

2.1 Background

To give an AD algorithm the capability of evaluating the derivatives of an MPI program, one need to figure out what should be done in the AD algorithm to deal with the communication. For forward mode, it's not hard. Because a `Send/Recv` pair can be viewed as a value assignment, we only need to send the derivative informations corresponding to the variable as well. And upon receiving the derivative information, the algorithm proceeds exactly the same as if it is a local assignment.

For reverse mode, the story is different. Because how to properly treat communication is the key problem. Previously, most work [2, 3, 5] focus on first order reverse mode. And in those algorithms, the communication is also reversed when evaluating the derivatives. The idea is that a `Send/Recv` pair can be viewed as a remote value assignment, like `a = b`. Then according to Figure 1, the derivative evaluation becomes $\bar{b} += \bar{a}$. So in the MPI version of that algorithm, a `Send/Recv` pair for variable value during function evaluation becomes a `Recv/Send` pair for adjoint value during first order reverse mode. For first order derivatives, the algorithm works correctly.

However, the reverse communication approach does not work for evaluating derivatives other than first order. One reason there is no efficient high order reverse mode algorithm ever proposed. Another reason is, in reverse mode, a `Send/Recv` pair should not be simply viewed as a "remote" value assignment. In the next section, we'll give a closer look at what should a `Send/Recv` pair in MPI corresponds to in terms of AD and give the high order reverse mode algorithm for `MPI`.

2.2 The MPI Version Algorithm

In AD, besides the SAC sequence, there are two primitives : declaration of an independent variable and declaration of a dependent variable. The declaration of an independent variable indicates that the value of this variable is the initial value of the variable is pre-determined. And the declaration of an dependent variable indicates that the derivatives of the declared variable need to be computed.

On the other hand, a `Send` in MPI means the value of the variable will be used by another (remote) process. This implies that if we want to enable AD, the derivatives of that variable will also be used by the other process. In that sense, a `Send` corresponds to a declaration of a dependent variable in terms of AD. Meanwhile, a `Recv` in MPI means the value of the variable is define by another (remote) process. Therefore, it corresponds to a declaration of an independent variable in terms of AD.

So, if we replace every `Send` by an *dummy dependent variable* and every `Recv` by an *dummy independent variable*, we can run the high order reverse mode locally on each process to evaluate the derivatives for every local dependent variable (including dummy ones) with respect to local independent variables (including dummy ones also). Then we can redo the communication in the order they took place during function evaluation. On `Send`, we send the derivatives computed for the dummy dependent variable. And on `Recv`, we use the received derivatives and apply Equation **??** to expand the dummy independent variable. That is, re-perform one pair of `Send/Recv` and apply Equation **??**. This operation eliminates a pair of dummy dependent variable and independent variable created by the `Send/Recv` pair. Once we've done that for all communications, the left dependent variables and independent variables are all explicitly declared ones in the objective function. And the derivatives of each dependent variable is available.

3. REFERENCES

[1] A. Griewank and A. Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. Siam, 2008.

[2] P. Heimbach. 3.13 application of derivative code in climate modeling. *Adjoint Methods in Computational Science, Engineering, and Finance*, page 14, 2015.

[3] P. D. Hovland. *Automatic differentiation of parallel programs*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[4] U. Naumann. *The art of differentiating computer programs: an introduction to algorithmic differentiation*, volume 24. Siam, 2012.

[5] J. Utke, L. Hascoët, P. Heimbach, C. Hill, P. Hovland, and U. Naumann. Toward adjoinable mpi. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.