1 Introduction

In scientific computing, we often need to compute derivatives in various numerical methods. Automatic differentiation [1] is a method to automatically generate a program that computes derivatives, from the code that evaluates the function value. With that, people can focus on the core of their scientific problems, and avoid manually writing the code for derivative evaluations which is a tedious and time-consuming job.

The way automatic differentiation computes derivatives is essentially through chain rule, and therefore it does not lose any accuracy. As a result, it does not have the drawbacks in numerical differentiation [2] which is normally implemented using finite difference scheme where the step-size needs to be carefully chosen to avoid round-off error as well as truncation error. It is also not as complicated as symbolic differentiation [3] where the analytical symbolic expression of the function derivative is derived.

There are two modes of automatic differentiation – the forward mode and the reverse mode – they use different chain rules to propagate derivative information. There are two main implementations of automatic differentiations: operator overloading and source transformation. The details about these are sketched in Section 2.

In this project, we aim to parallelize the reverse mode automatic differentiation, and we focus on the implementation using operator overloading. We present two parallelization strategies: one is to parallelize the loops in the original code, and the other is to parallelize the directed acyclic graph (DAG) traversal of the computational graph in the reverse mode.

To parallelize the loops, we present a method that is implemented using operator overloading to build up a vectorized computational graph that recognizes the loops/vector operations in the original code. The nodes in this graph can represent vector variables as opposed to scalar variables in the conventional automatic differentiation algorithm. This gives us a way to parallelize the vector operations in modern computer architectures.

To parallelize the DAG traversal in the reverse mode, we adapt the level-synchronous method in (breadth-first search) BFS [4]. The major difference between our algorithm and the normal BFS level-synchronous method is that in the reverse mode of automatic differentiation, when accessing a node, one has to read/write the predecessors of that node, and therefore, the definition of the “independent” frontier needs to be changed. We perform an independence check before distributing nodes to processors/threads.
In this report, we will only focus on the reverse mode, and apply it to functions that have multiple independent variables and one dependent variable since that is the kind of application where the reverse mode is superior to the forward mode.

The rest of the report is organized as follows. Section 2 reviews the basics of automatic differentiation. Section 3 presents key ideas to parallelize loops in automatic differentiation, and Section 4 presents the adaptation of the level-synchronous method.

## 2 Background

Throughout the report, we will stick to the following notation: \( z = f(x) \) is a dependent variable that is a function of independent variables \( x \), and \( y \) are intermediate variables that are also functions of \( x \).

### 2.1 Forward Mode and Reverse Mode

To see how forward mode works, we consider an intermediate variable

\[
y = g(h_1(x), h_2(x), \cdots, h_n(x)).
\]

By chain rule, we have

\[
\frac{\partial y}{\partial x} = \sum_i \frac{\partial y}{\partial h_i} \frac{\partial h_i}{\partial x}.
\]

Since we can compute \( \frac{\partial y}{\partial h_i} \) given that we know the numerical values of \( h_i \) and that \( g(\cdot) \) is a known function with derivative formula defined, (2) tells us that if we know \( \frac{\partial h_i}{\partial x} \), then \( \frac{\partial y}{\partial x} \) can be computed. This leads to the forward mode of automatic differentiation – we start with \( \frac{\partial y}{\partial x} = 1 \), and we propagate the derivatives \( \frac{\partial y}{\partial x} \) until we get the derivative of the dependent variable to the input variable \( \frac{\partial z}{\partial x} \).

One can also think of the computational graph of a function where the top nodes correspond to dependent variables and bottom nodes correspond to independent variables. In forward mode, we basically propagate the derivative information \( \frac{\partial y}{\partial x} \) bottom-up in the computational graph.

As the name tells, the reverse mode propagate the derivative information top-down in the computational graph. Different from the forward mode, the reverse mode propagates \( \frac{\partial z}{\partial y} \) which is also known as adjoints. To do that, we need to utilize the chain rule in another way. Consider

\[
z = g(h_1(y), h_2(y), \cdots, h_n(y)),
\]

then we have

\[
\frac{\partial z}{\partial y} = \sum_i \frac{\partial z}{\partial h_i} \frac{\partial h_i}{\partial y},
\]

i.e., if we know \( \frac{\partial z}{\partial h_i} \), then \( \frac{\partial z}{\partial y} \) can be computed. Therefore, starting at the top node with \( \frac{\partial z}{\partial y} = 1 \), we can propagate the adjoints top-down, and finally obtain \( \frac{\partial z}{\partial x} \).

The main disadvantage of the forward mode is exhibited when we compute the derivative of a scalar variable to \( n \) variables with \( n \) being large. Since at each node in the computational graph, one needs to store \( \frac{\partial z}{\partial x} \) which is a vector of size \( n \), this can cause a memory problem when \( n \) is
large. Although this problem is less catastrophic in practice since it is not necessary to build up the complete computational graph and therefore not necessary to store the derivative information in all nodes, it at least need to store vectors at starting nodes, which amounts to $O(n^2)$ memory.

In contrast, reverse mode is much better in this situation, because at each node, only one scalar number $\frac{\partial z}{\partial y}$ needs to be stored. However, the main disadvantage of the reverse mode is that it has to build up the computational graph in order to traverse it, and the computational graph itself may be large. The benefits usually comes when this derivative function is to be called for many times. In that case, the computational graph, or the detailed ordered access of all nodes (also called a tape, can be stored, and be reused many times.

There are other situations such as computing the derivatives of a vector function to a scalar variable and compute the derivatives of a vector function to a vector (Jacobian), etc., where forward mode and reverse mode have their own advantages and disadvantages.

### 2.2 Operator Overloading and Source Transformation

There are two mainstream implementations of automatic differentiation: operator overloading and source transformation.

Operator overloading implementation is straightforward and easy to understand – one overloads the operators to not only compute function values but also compute derivative information or build up the computational graph. For example, one can define a new class `ad_double`, and declare original `double` variables as `ad_double` variables. Then the program will use the operators defined for `ad_double` which compute derivative information or build up the computational graph.

Source transformation works like a compiler, which reads in the code that computes the function, analyzes the code and exports a source code that computes the derivatives. This implementation normally generates more optimized programs but involves a lot of implementation efforts.

### 3 Parallelizing Loops

#### 3.1 Motivation

It is very common in scientific computing that the code for function evaluations involves loops. For example, any vector or matrix operations are coded using loops. Except for linear operations for which one can call BLAS library [5], a wider variety of nonlinear functions of vectors/matrices are implemented using loops.

This poses a big problem for the reverse mode of automatic differentiation – because of the loop, the size of the computational graph increases a lot. For example, consider the following C code.

```c
double func1(double* x, int n){
    int i;
    double* g = (double*) malloc (n * sizeof(double));
    double f;

    for (i=0;i<n;i++){
        g[i] = 2 * x[i] + 1;
    }

    f = 0.0;
    for (i=0;i<n;i++){
        f = f + g[i];
    }
    return f;
}
```
The computational graph for this simple example is shown in Fig. 1. We see that the first loop generates \( n \) parallel branches, each having 5 nodes, and the second loop, which does a reduce operation, creates \( n - 1 \) nodes. Hence, there are \( 6n - 1 \) nodes in total.

Figure 1: Computational graph of \( \text{func1} \).

However, what this function really does is to sum up all the elements in vector \( 2x + 1 \). In MATLAB notation, it is simply \( f = \text{sum}(2x+1) \). Therefore, a much simpler vectorized computational graph can be derived, and it consists of only 6 nodes, as shown in Fig. 2.

Figure 2: Vectorized computational graph of \( \text{func1} \).

Since some nodes in this DAG represent vectors of potentially large size, it provides possible
parallelization when propagating the derivative information. In other words, this DAG “recog-
nizes” the loops in the source code, and therefore provides a way for parallelizing the loops.

We also notice that in this example, there are two kinds of loops: the first loop operates on a
vector and generates another vector, and is data-parallel; the second loop operators on a vector
and generates a scalar, and indeed implements a reduce operation. Accordingly, the construction
of vectorized graphs for these loops differ a little bit, and we will discuss these in the following
sections.

3.2 Constructing Vectorized Computational Graphs using Operator Over-
loading: an Example

Vectorized computational graphs (such as the one in Fig. 2) is great for parallelization and for
alleviating memory constraints. However, codes are normally written using loops and vectors with
subscripts, instead of vectors directly. Therefore special techniques must be used to detect vectors
in original codes.

The fact that codes are written using loops and vectors with subscripts gives us the key idea to
overload the subscript operator [ ] as an extension to the conventional automatic differentiation
implementation using operator overloading.

To do that, two more classes are needed. Firstly, the subscript operator has to operate on a vector,
and therefore, we need a new class ad_double_vec which represents a vector. Secondly, the
subscript operator needs to know the expression of its index, and we use a new class ad_int_idx
which represents the index in the subscript operator.

As a result, the vectorized computational graph for func1 that involves subscript operators is
shown in Fig. 3. Similar to the conventional computational graph, each node has its operator and
operands. The differences are:

- Some nodes represent vectors, instead of scalars. Therefore, they store a vector of values
  (vals) and adjoints (adjs).

- The vector node has a vector of pointers expr that point to the values of the vector. The
  values are stored in itself if expr is NULL.

- Some nodes represent indices, and they do not need to store vals and adjs. They are used
to track the vectors in a loop.

- There is a new operator [ ] which operates on a vector node, and stores the indices that are
  referenced.

We now go through the procedure that builds up this vectorized computational graph, and explain
the techniques we use in the algorithm.

3.2.1 Rewrite the Function using New Classes

To use the overloaded operators, one simply needs to redefine the types of double variables and int
variables in the original code. For example, the modified func1 for operator overloading is

```c
ad_double func1_ad(ad_double_vec& x, int n){
    ad_int_idx i;
    ad_double_vec g(n);
    ad_double f;
```
for (i=0;i<n;i++){
    g[i] = 2 * x[i] + 1;
}

f = 0.0;
for (i=0;i<n;i++){
    f = f + g[i];
}
return f;

3.2.2 Overloading Subscript Operator

Suppose the func1 is called with n=4, then we basically have a loop-unrolled equivalent function func1_loopunrolled:

ad_double func1_loopunrolled(ad_double_vec x, int n){
    ad_int_idx i;
    ad_double_vec g(n);
    ad_double f;
    
    g[0] = 2 * x[0] + 1;
    g[1] = 2 * x[1] + 1;
    g[2] = 2 * x[2] + 1;
    g[3] = 2 * x[3] + 1;
    
    f = 0.0;
    f = f + g[0];
    f = f + g[1];
    f = f + g[2];
\( f = f + g[3]; \)

\( \text{return } f; \)

When \( g[0] = 2 \times x[0] + 1; \) is called, the program calls the operators in the following order: \( x[], \ast, +, g[], =. \) When \( i=0 \) and \( [] \) is called, we create a new node with operator being \( [] \) and operands being the vector and the index. It also stores the current index to itself and references the current value to be the subscripted value.

### 3.2.3 Use Structural Hashing to Avoid Repeating Nodes

When \( i=1 \) and \( [] \) is called, however, we do not want to create a new code, since we know it is another element in the vector. To avoid that, we keep a global unique_table which is a hash table with keys being the operator and operands, and the value being the node the operator is assigned. Using the unique_table, we will never have repeating nodes in the graph. Therefore, in the second loop, we simply add the index \( i = 1 \) to the node and change the current value of the node to the new value.

Accordingly, when \( \ast \) and \( + \) are called for the second time, we simply add the new value into the corresponding node. When \( = \) is called for the second time, we change the expr pointer of the \( g \) vector to the \( + \) node.

### 3.2.4 Reduce Operation

The second loop in func1 basically implements the reduce operation. However, since it sums up \( n \) variables, it creates \( n \) nodes in the computational graph. However, we have seen in Fig. 2 that only one node is needed. This is because that the \( + \) operation is associative. To utilize this property, we combine the \( + \) nodes whenever one of its operand node has operator \( + \). An illustration of this procedure is shown in Fig. 4. This applies to all associative operators such as \( + \) and \( \ast \).

![Figure 4: Illustration of the reduce operation.](image-url)
3.2.5 Storing Indices Efficiently

We have shown that each node with \([\text{operator}]\) being the operator has to store a vector of indices that are referenced. This is a potentially huge vector and can again lead to memory problems. However, since the indices appear in a loop, and therefore generally have an expression associated, we can simply use a three-tuple \((\text{istart}, \text{istep}, \text{istop})\) to store the index range, and use the other operand as the expression of the index. Thus we can store the indices efficiently using only three integers.

3.2.6 Storing Expression Pointers Efficiently

In each vector node, we have a vector of expr nodes that are used to point to the values of that vector. Similarly this can easily consume a lot of memory. To avoid that, we maintain a table which maps indices to expressions. Since indices are stored efficiently using the trick mentioned before, we expect to use very few expr pointers. This is also the case in many practical problems, and is certainly the case for \(\text{func1}\) shown above.

3.2.7 Parallelizing the Access of a Vector Node

Since a vector node of size \(n\) corresponds to \(n\) adjoints to be computed, this task can be partitioned and distributed to several processors/threads. We have implemented this using OpenMP. Initial results show that using 2 threads, 1.64X speedup is obtained when the size of the vector is large enough.

Since the computation of the derivative/adjoint of an elementary function is cheap, it may cause the overhead of parallelization to be too expensive for the parallel code to outperform the serial code. In that case, we should employ a coarser-grained parallelism where the propagation of the derivative information through several nodes are parallelized. Another way to implement a coarser-grained parallelism is to ask the user to provide the derivative code for a sub-function, and the parallelism is applied on propagating the derivative information of that sub-function.

3.3 More Examples

The above example explains the main ideas we use to construct the computational graph, although there are lots of subtle implementation details not mentioned.

Here we show the vectorized computational graphs for two more example.

The first example is \(\text{func2}\) as shown below. In this example, we have more complicated sub-script indexing in the code, and the corresponding vectorized computational graph is shown in Fig. 5.

```c
double func2(double* x, int n){
    int i;
    double* g = (double*) malloc (2 * n * sizeof(double));
    double f;

    for (i=0;i<n;i++){
        g[2*i+1] = 2 * x[2*i] + 1;
        g[2*i]  = x[2*i+1] - 1;
    }

    f = 0.0;
}
```
for (i=0;i<n;i++){  
    f = f + g[i];  
}  
return f;

Figure 5: Vectorized computational graph of func2.

The second example is func3 as shown below.

double func3(double * x, int n){  
    int i;  
    double * g = (double*) malloc (2 * n * sizeof(double));  
    double f;  
    for (i=0;i<n;i++){  
        g[i] = g[i] + x[i];  
        g[i+n] = 2 * g[i];  
    }  
    f = 0.0;  
    for (i=0;i<2*n;i++){
        f = f + g[i];
    }  
    return f;
}

In this example, the RHS of an assignment has the same component as LHS. In the conventional automatic differentiation code, a new node will be generated and be designated as the variable on the LHS. However, we cannot do this since we hope to use the unique_table to avoid repeating nodes. Instead, we create a new vector node, and point the expr pointer of the LHS vector node to that new node. When this new node is referenced later, however, we will create a new subscripting node with operator [], and also add that node to the unique_table. Hence, the vectorized computational graph is constructed and is shown in Fig. 6.
4 Parallelizing DAG Traversal

Another parallelization strategy is to parallelize the DAG traversal in the reverse mode. To do that, we adapt the level-synchronous method for BFS [4]. The basic idea is to maintain a list of nodes of the current frontier of the breadth-first search, and distribute the access to all these nodes to different processors/threads, and run them in parallel. This has assumed, however, that the access to nodes in the frontier are independent.

However, in reserve mode of automatic differentiation, when we access a node in the computational graph, we also read/write its predecessors in order to propagate the derivative information. Therefore, it is highly possible that two nodes in the frontier have the same predecessor, leading to a race condition.

To avoid that, we pre-scan the nodes in the frontier before distributing them to difference processors/threads. We check if they have the same predecessor nodes. If so, we remove some of them iteratively until we have a set of nodes that do not share any predecessor nodes. This may increase the complexity of the code. However, since this information can be stored, and reused for future function calls, the time complexity can be alleviated.

To show an example of this, Fig. 7 plots the computational graph of func4 which is shown below. It also plots the actual frontier that is used in each iteration of the DAG traversal. It is observed that because we only distribute nodes that are independent, the parallelism achieved is limited at the beginning, and starts to benefit when the number of nodes in the frontier becomes large. This can, of course, be further optimized according to special structures of computational graphs, but we do not discuss these in this report.

```c
double func4(double* x, int n)
{
    double f;
    int i, j;
    for (i=n-1; i>=0; i--)
    {
        for (j=0; j<i; j++)
            x[j] = x[j] + x[j+1];
    }
    f = x[0];
    return f;
}
```
5 Conclusion

In this report, we have presented two parallelization strategies for the reverse mode of automatic differentiation. The first one is to build up a vectorized computational graph, and then parallelize vector operations. The second one is to adapt the level-synchronous method in the DAG traversal in the reverse mode. We have seen from experiments that we need to feed each thread enough work in order to achieve parallelization. Other optimizations such as SIMDization and graph algorithm for specific graphs are also possible. Finally, although our implementation is on the shared memory architecture, other platforms (such as CUDA) may be better choices.

This work has not been fully developed, implemented and tested yet – only simple operators are overloaded at this stage, and the code is still being tested for more examples. I hope to continue working on this and have the tool released if it turns out to be useful.

Acknowledgments

The author would like to acknowledge Baruch Sterin and Sayak Ray for extremely useful discussions.

References


