Increasing the Granularity of Parallelism and Reducing Contention in Automatic Differentiation

Brad N. Karp
Christian H. Bischof

CRPC-TR90075
November, 1990

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439-4801

ANL/MCS-TM-142

Increasing the Granularity of Parallelism and Reducing Contention in Automatic Differentiation*

by

*Brad N. Karp** and Christian H. Bischof*

Mathematics and Computer Science Division
Technical Memorandum No. 142

November 1990

*This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

**Participant in the Summer 1990 Student Research Participation Program. This program is coordinated by the Division of Educational Programs.

Contents

Increasing the Granularity of Parallelism and Reducing Contention in Automatic Differentiation

Brad N. Karp and Christian H. Bischof

Abstract

The automatic differentiation package ADOL-C of Griewank and Juedes, traces the computation of a function whose derivative is to be computed, and then subsequently propagates adjoint values along these traced paths according to the chain rule. While a sequential implementation of the reverse mode can utilize this trace by traversing it in reverse order, a parallel implementation must build the entire computational graph, as different processors will each simultaneously be working on different sections of the trace. In a sequential implementation, the linearized trace inherently obeys the *dependencies* of the function evaluation. In a parallel implementation, however, there must be a mechanism to determine whether a node's dependencies have been resolved yet, meaning that the node's adjoint value is computable. A node in the graph must represent a quantity of arithmetic operations large enough for a processor to do enough computation before it must communicate the result to another processor, but small enough for there to be enough nodes in the graph to allow many processors to work simultaneously. Sinks are bottlenecks for efficient parallel computation, as their many dependencies mean that many processors will contend for them simultaneously. These two factors are both familiar problems in parallel computation; the first is an issue of *granularity*, while the second is an issue of *contention*. As a first step toward achieving an efficient parallel implementation, we present a system for construction of a computational graph from ADOL-C's computational trace, as well as two transformations for this graph, *hoisting* and *splitting*, which improve its computational granularity and reduce contention, respectively.

1 Origins, Introduction, and Motivation

Griewank showed in 1988 [1] that when the reverse mode of automatic differentiation is used on a sequential processor, "the evaluation of a gradient requires never more than five times the effort of evaluating the underlying function by itself." This observation led to interest in the implementation of an automatic differentiation package that would produce gradients for existing coded functions (ideally in C or Fortran) with minimal modifications to these preexisting functions. Juedes and Griewank [2] soon thereafter produced ADOL-C, a sequential implementation of the reverse mode of automatic differentiation using the operator overloading features of C++. Their package overloads all standard binary arithmetic operators, assignment, and other commonly used univariate mathematical functions and produces a complete record of each individual operation carried out in the course of a function's computation, via this overloading mechanism, called a *tape*. This tape is then used on a sequential processor to propagate adjoint values backward through the function's computation, in the end producing a gradient value.

A logical next step in the efficient implementation of automatic differentiation is the creation of a system that carries out the reverse mode on a parallel processor. Indeed, Juedes

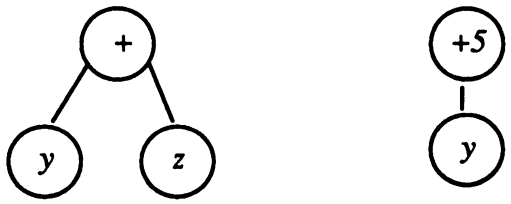
and Griewank [2] produced just such an implementation for the Sequent Symmetry. Their parallel implementation generates a directed acyclic graph of intermediate values along the course of the function's evaluation, whose edges represent the *dependencies* of these values on one another. The Sequent architecture supports local memory access for each individual processor as well as access to a shared memory pool by each processor. This shared memory has a flat hierarchy and must be accessed via locks in order to ensure mutual exclusion of access by processors to a single memory location. The evaluation of this dependency graph is a fine-grained problem, as there is no obvious partitioning method for the division of the evaluation into equal-sized subproblems for distribution among the processors. Their strategy for allocation of work to processors is simply to begin traversing the graph at the dependent variable, placing nodes whose dependencies have all been resolved onto an *evaluation queue.* When a processor needs work, it attempts to "take" a node from this queue for itself. A major concern when using such a system is the amount of overhead generated by locking shared-memory locations, especially when there is heavy contention among processors for a single such location. Juedes and Griewank attempt to circumvent this potential difficulty by using a two-tiered queue system: each processor has its own private queue in local memory, and every pair of processors shares a common queue in shared memory. When a processor resolves a node's last dependency, it adds this node to its own local queue, unless its local queue is full, in which case it places the node on its shared queue, which is of variable size. Similarly, when a processor's local queue is empty, it attempts to find work in its shared queue and, if necessary, accesses shared queues of other pairs of processors to find work if its own shared queue is empty. Certainly, this scheme reduces shared-memory access from the amount that a more naive single shared-memory queue implementation would use. However, it is clear that this implementation still employs extremely fine-grained parallelism, as a queue access (either in local or, sometimes, in shared memory, incurring an "expensive" lock operation) is made for each individual arithmetic operation.

One would hope that some sort of clustering of operations into a single "package" to be computed in its entirety by the same processor could improve the efficiency of automatic differentiation in parallel. This hope is justified, *as long as the choices of which operations to cluster together are made intelligently.* To this end, consider the following fragment of a function being differentiated:

$$\sin(\cos(\tan(\sqrt{x+y}))).$$

It is noteworthy that the graph representation of this calculation will have single edges (or *links*) between nodes, with the exception that the bottom + will have two links, emanating to x and y. (See Figure 1.) Because of the single links connecting this fragment, parallelism can contribute no benefits whatsoever to the fragment's evaluation; no node will be ready to be evaluated in this chain until its one parent has been evaluated. Yet, in Juedes and Griewank's implementation, these nodes will each be fetched individually from evaluation queues. Here we have the motivation for our first graph transformation, *hoisting*, which attempts to take advantage of the above mentioned construct to improve the granularity of the computation.

It is also noteworthy that the result of a single arithmetic operation can be used infinitely many other places later in the computation. Consider the simple case of $x = y + z$. The code for the function being differentiated may use x arbitrarily often thereafter, which in turn means that the node representing x in the graph can have arbitrarily many links emanating "upward" to other nodes. (See Figure 2.) Thus, in the reverse mode of automatic



(a) (b)

Figure 3: In (a), $y + 5$ is inefficiently represented as + with two children, y and 5; in (b), generated by our implementation, $y + 5$ is represented as +5 with one child, y.

this scheme generates an acyclic graph from the cyclic graph that ADOL-C's live variable analysis produces. (For example, when ADOL-C stores the result of a x += 5 in the same array location where the original value for x was stored, we have the array location's value depending on its own old value–a cycle. When a location is reused, this graph-building scheme doesn't deallocate the node referenced by the old pointer in the array location; it simply overwrites the pointer, leaving the old node intact. Thus, by creating separate nodes in the graph for each time an array location was reused by ADOL-C, these cycles are eliminated.) It is important that the graph remain acyclic if it is to be suitable for parallel evaluation. Also, the graph complexity is reduced somewhat during the graph's construction by including constants used as operands inside of the operation's node itself, rather than creating a separate node containing the constant and linking it to the operation. In other words, a construct such as $y + 5$ will be represented in the graph as a node containing the preexisting value of y linked upward to a node containing the information +5 rather than linked upward to a node containing the information + which would have 5 as as second child. (See Figure 3.)

As different opcodes have varying parameter types and counts, the tape comprises records of varying C structures. The different structures used in the tape can be found in `template.h`. In an effort to keep the code complexity of routines that work on the graph to a minimum, a single data type for nodes, regardless of their opcodes, was adopted. This single type **node** contains a union to maximize space efficiency, as certain quantities associated with one opcode are mutually exclusive of other quantities associated with other opcodes. (A binary operator that involves a constant must store a constant and a pointer to the node containing the other operand, whereas a binary operator that involves no constants must store two pointers to nodes containing operands, for example.) In actuality, there is a two-tiered organization. When the graph is initially built, each node created consists of a **gennode** half and a **node** half. These graph data structures can be found in `graph.h`.

As a footnote to this description of the graph's construction from the tape, it should be mentioned that the error condition in which a tape generated by ADOL-C references a location in the live variable array before any value has been assigned to the location is handled by the graph construction code. The uninitialized location is set to be an assignment opcode whose value is one, so that a segmentation violation will not occur when the location is referenced, and an error message containing the exact location of the faulty reference is printed. This error trapping allows code with improper or insufficient variable initializations to successfully build a graph for test purposes, and it also gives the programmer of the code being differentiated a precise indication of where the fault in the code lies.

3 Hoisting

Now that the computational graph has been generated, we wish to put this graph into a form where it is most readily suitable for parallel processing. Our first step toward this goal is the elimination of the chain construct pictured in Figure 1. Ideally, such chains should be coalesced into single, encapsulated units, so that a processor will be assigned the entire length of the sequential chain to compute rather than the first node on the chain's end. *Hoisting* accomplishes exactly this task.

Perhaps the most serious implication of hoisting for the graph and its evaluation is the introduction of a second data type for nodes in the graph; homogeneity of the graph is lost. After hoisting, the graph contains *regular* nodes, which encode single operations as before, and *supernodes*, which encode singly linked chains of operations (with either a leaf or a binary operation at the chain's end). From the standpoint of data type, three data types make up a hoisted graph. The first is **gennode**. A **gennode** contains all information that must be stored both in nodes and in supernodes, as well as identification of the node type. The second and third are **node** and **snode**. A **gennode** is associated with either a **node** or an **snode**. A **node** contains the data specific to a regular node, while an **snode** contains data specific to a supernode. Thus, an **snode** encodes several operations that were originally in a singly linked chain, while a **node** encodes only a single operation. A **gennode** can be thought of as a sort of "wrapper" for the two different node types that occur in the hoisted graph. (Once again, the details of the data structures can be found in `graph.h`.)

As mentioned previously, the graph generated initially from the tape consists only of nodes, with no supernodes. This fact simplifies the work to be done in the hoisting process, as it guarantees that during a top-down traversal of the graph, any node encountered that has not yet been visited will not be a supernode, *as long as supernodes are expanded maximally when first created.* In other words, when two nodes can be collapsed into a supernode, the algorithm for collapsing the two should not stop at just those two unless necessary; rather, it should attempt to collapse as many nodes as possible (as determined by the criterion below) in a single sweep. The algorithm for hoisting works just this way; it creates the largest supernodes possible in a depth-first fashion when it discovers that a supernode can be created.

The formal rule for determining the "hoistability" of a node is as follows:

> *A child c can be hoisted into a node n if n has only one link downward, which points to c, and c has only one link upward, which points to n.*

Note that this rule does not specify the number of children that *c* may have; a supernode may have arbitrarily many parents and either zero or two children. (Note that a supernode cannot have one child except during the time while it is being built, because supernodes are maximally expanded when built.) Also note that the current implementation allows for arbitrarily large supernodes, (i.e., supernodes that encode arbitrarily many operations), as the data structure used to store operations and operands inside a supernode is a dynamically allocated linked list. It might be worthwhile in the future to implement hoisting where a supernode would have statically allocated storage for operations and operands, fixing a limit on the number of operations contained inside one supernode. Such a statically allocated supernode data structure would be simpler than the current implementation's dynamically allocated data structure. However, further information about the numbers of nodes coalesced into single supernodes must be obtained before this question can be evaluated properly.

```
loop
  pop a node n off of traversal stack
  u = uplinks(n)
  if u > threshold
       n2 = n
       r = u modulo threshold
       q = (u integer divide threshold) - 1
       temp = uplink ptrs of n2 after first threshold uplink ptrs
       remove all but first threshold uplink pointers from n2
       while q > 0
            a = new += 0 node, marked as visited
            attach a above n2
            move the first threshold uplink ptrs from temp to a
            n2 = a
       if r > 0
            a = new += 0 node, marked as visited
            attach a above n2
            move all remaining uplink pointers in temp to a
  push n's non-visited children onto traversal stack
  mark n's non-visited children as visited
until traversal stack empty
```

Figure 6: The splitting algorithm

problem employing nonlinear least squares methods, yielded more encouraging results. The computational graph we produced from ADOL-C's tape of this function was 108,686 nodes in size when no transformations were applied to it. When hoisting alone was applied to this graph, the resulting graph was only 67,242 nodes in size, indicating that a great many nodes were collapsed into supernodes—a signnificant improvement in granularity. When splitting and hoisting were both applied to the graph, the resulting graph contained 70,896 nodes. Thus, for this particular test function, hoisting and splitting have a profound effect on the graph's organization. Perhaps the most important observation to be made from these two examples is that the effectiveness of our transformations is rather problem dependent; the graph of the Helmholtz energy function was not changed nearly as much as that of the shallow water test problem by our transformations.

6 Conclusions

We have presented a method for generating a computational graph from the tape produced by ADOL-C, and two transformations of this graph to improve its suitability for parallel evaluation: by improving its granularity (hoisting), and by eliminating one-node hot spots of contention (splitting). Our results are encouraging but indicate that the potential benefits reaped from these transformations are problem dependent.

While hoisting and splitting may not always have a profound effect on a graph's size and structure (as in the Helmholtz energy function example), they can make a great difference (as in the shallow water problem), and the resulting graph is likely to provide a useful base for the construction of an efficient parallel implementation of automatic differentiation.

Acknowledgments

BK is indebted to Chris Bischof, Andreas Griewank, David Juedes, and Jay Srinivasan for their friendly assistance and support (in both technical and social capacities) during his stay at Argonne.

References

[1] A. Griewank, "On Automatic Differentiation," in *Mathematical Programming: Recent Developments and Applications,* ed. M. Iri and K. Tanabe, KTK Scientific/Kluwer Academic Publishers, 1989.

[2] D. Juedes and A. Griewank (1990). *Implementing Automatic Differentiation Efficiently,* Mathematics and Computer Science Division Technical Report ANL/MCS-TM-140, Argonne National Laboratory, Argonne, Illinois.

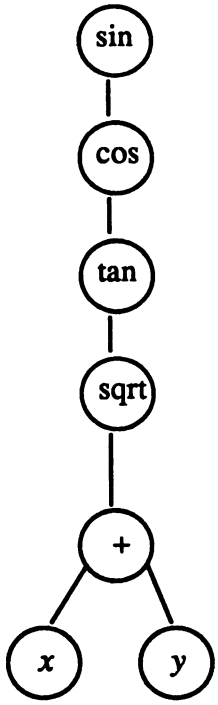


Figure 1: The graph representation of $\sin(\cos(\tan(\sqrt{x+y})))$

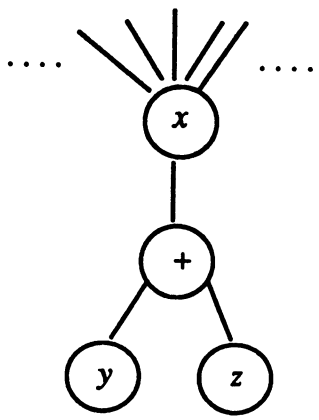


Figure 2: A graph where x is likely to be a candidate for contention

differentiation, such a node with a high upward branching factor will be a "hot spot" of *contention* among processors, as the probability is high that several processors on the other sides of these many upward links will simultaneously attempt to access the node to determine whether it is ready to be evaluated. Here we have the motivation for our second graph transformation, *splitting*, which attempts to restructure such nodes with high upward branching factors so that contention for them will be reduced.

The starting point for the work presented here is the tape of a function evaluation produced by ADOL-C, Juedes and Griewank's automatic differentiation package.

2 The Tape and the Graph

Before the trace produced by ADOL-C can be transformed into a form more suitable for parallel evaluation, the more basic transformation from the tape into the dependency graph must be accomplished. This transformation is quite simple, as the tape produced by ADOL-C is merely a linearization of the graph in question.

ADOL-C stores all values generated during the course of a computation in an array. All overloaded variables in the user's ADOL-C program are assigned array locations by ADOL-C, and results of overloaded arithmetic operations are also assigned array locations by ADOL-C. ADOL-C performs *live variable analysis* when managing this array, meaning that when an array location's value will no longer be used for further computations (this situation occurs when an overloaded variable is destructed in the user's ADOL-C program, for instance), it will reuse this array location for a newly generated value in an effort to keep the number of array locations used at any given time to a minimum. (This reuse includes the trivial cases of $+=$ and other such overwriting operators; $x \mathrel{+}= y$ will be stored in the same array location as the original value of x.) When ADOL-C places a value into the array, it writes out a record onto the tape detailing the information about the operation that generated the value. Each record on the tape thus represents a single operation and contains the operation code (*opcode*) (each arithmetic operation overloaded by ADOL-C has a symbolic integer constant associated with it for this purpose), the array subscript where the operation's result is to be stored, and the array subscript(s) of the operation's operand(s). (All ADOL-C operators are either unary or binary, so each record will contain either one or two operand locations.) The very beginning of the tape contains the tape's parameters stored in ASCII format, including the buffer size used to create the tape; the maximum number of live variables simultaneously in use (needed to allocate storage as simply as possible when reading the tape to build the graph); and the numbers of records stored on the tape, independent variables, and dependent variables.

Thus, the procedure used to build the dependency graph from the tape is to read the tape's parameters and then read each record from the tape. An array of pointers to nodes, the size of the maximum number of live variables, is allocated. This array serves as an analog to ADOL-C's intermediate value storage array. When a record is read from the tape, a node is allocated and initialized to contain the record's opcode. The pointer to this node is stored in the location of the array where the "result" of the operation is located. This new node is then connected to the nodes on which it is built, by storing the pointers located in the pointer array at the locations specified by the operand indices of the tape record in it. Similarly, the nodes on which the new node was built are updated to contain pointers to the new node. (The edges of the graph are all bidirectional.) This process repeats until the entire graph has been constructed, when each tape record has been processed. Note that

```
loop
  pop a node n off of the traversal stack
  c = child(n)
  if n has one child
       while c has one parent
            n = supernode containing n and c together
            c = child(n)
            if n has more than one child or no children
                 break from while
  push n's non-visited children onto traversal stack
  mark n's non-visited children as visited
until traversal stack empty
```

Figure 4: The hoisting algorithm

The transformation is implemented as a depth-first traversal of the computational graph. Each node in the graph has a `visited` flag in its `gennode`, and all nodes are marked as not having been visited at the start of the traversal. The root of the computational graph is pushed onto the traversal stack, and then the algorithm in Figure 3 is executed (after which the entire graph has been traversed) When this loop terminates, the hoisting process has been completed. The code for hoisting can be found in `transform.c`.

4 Splitting

While hoisting improves the *granularity* of the graph evaluation problem, splitting reduces the likelihood of *contention* during the graph's evaluation. As stated earlier, when the result of an arithmetic operation is used later in many places in the computation, the node that encodes this particular operation will have an extremely high upward branching factor. Splitting, as its name suggests, breaks a node of a high upward branching factor into several nodes. A threshold for the maximum number of upward links of a node is selected, and any node that has more than this number of upward links is split into several nodes, each with an upward branching factor equal to or less than this maximum. For an example of this simple transformation, see Figure 5. Of paramount importance is that the new nodes produced by the splitting process be computationally equivalent to the original node with a high upward branching factor. The opcode and operand of the "dummy" nodes created in the splitting process therefore encode += 0. Note that the minimum number of nodes needed to meet the maximum branching factor constraint will be created; if the original node is not evenly divisible by the maximum branching factor, then all nodes created will be of maximum branching factor except for one, which will have the remainder of the upward links; and if the original node is evenly divisible by the maximum branching factor, all nodes created will be of maximum branching factor. Note also that both nodes and supernodes alike can be split in this fashion; the new nodes produced are not dependent on the type of the node being split.

Like hoisting, splitting is implemented as a depth-first traversal of the computational graph, using the `visited` flag in `gennodes` to prevent repeated processing of the same node, as it is possible to reach the same node from more than one upward path in a directed acyclic

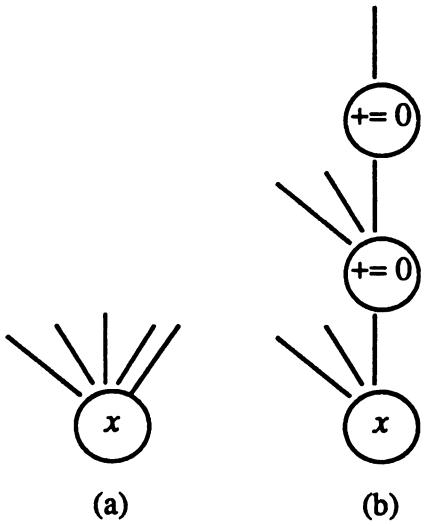


Figure 5: Assuming the maximum branching factor selected is two, we have a node that exceeds this maximum in (a), and its split equivalent in (b).

graph. Note that the same `visited` flag is used for all graph traversals, so that space is not wasted in each node for separate flags. It is possible to reuse this same flag because all graph traversals will have reversed the flag from its previous value in all nodes of the graph when they have completed. Since all flags in the graph initially have the same value, the value of the flag in the root node at the start of the traversal can be taken to mean, for the current traversal, that a node has not been visited. The change in a flag's value, rather than its absolute value, is all that is significant. (See the `polarity` variable in `transform.c` for the implementation of this single flag for multiple traversals.) Splitting first pushes the root of the computational graph onto the traversal stack and then executes the algorithm in Figure 4 (`threshold` below denotes an integer that is the maximum number of permissible upward links for a node to have): When this loop terminates, the splitting transformation has been completed. The code for splitting can be found in `transform.c`.

5 Results

The statistics on graph size obtained after performing hoisting and splitting on computational graphs from two real automatic differentiation problems are encouraging. The first problem on which we tested the transformations was the Helmholtz energy function [2], [1], previously used by Juedes and Griewank to test their sequential and parallel automatic differentiation implementations. The computational graph generated from ADOL-C's tape trace of a one-hundred-variable execution of the Helmholtz energy function was 21,115 nodes in size without application of our transformations. After hoisting was applied to this graph, without splitting, the graph was 21,010 nodes in size, indicating that 105 nodes were collapsed into supernodes. When only splitting, and no hoisting, was applied to the graph, the resulting graph contained 22,125 nodes. Lastly, when both splitting and hoisting were applied to the graph, the graph that resulted contained 22,020 nodes. While not necessarily striking in any respect, the results from this test function indicate that these two transformations can produce a graph more suitable for parallel evaluation without a substantial increase in the graph's size.

The second function on which we tested the transformations, a large shallow water test