Utilities for Building and Optimizing a Computational Graph for Algorithmic Decomposition

Christian Bischof
James Hu

CRPC-TR91142
1991

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

Contents

Utilities for Building and Optimizing a Computational Graph for Algorithmic Decomposition

Christian Bischof
James Hu

Abstract. This document describes a utility to construct and evaluate an optimized execution graph from the tapefile generated by the ADOL-C automatic differentiation software. It describes the format of the ADOL-C tapefile, the data structures used in building and storing the graph, and the optimizations performed in transforming the computation trace stored in the tape into an efficient graph representation. In particular, we eliminate assignments, increase granularity by "hoisting" chains of unary operations, and remove so-called dead roots – intermediate values that have no influence on the dependent. Examples show that the optimized graphs contain up to 50% fewer nodes than a graph that would be an exact analogue of the ADOL-C tape. We also describe an attempt at generating compiled code for the graph evaluation as an alternative to interpretative approaches to evaluating the graph.

1 ADOL-C and the Tapefile

1.1 ADOL-C

ADOL-C [3] is a collection of utilities that, given a C program for the evaluation of a function $f : R^m \rightarrow R^n$, can compute *exact* derivatives of arbitrary order in a fashion that is transparent to the user. Automatic differentiation in ADOL-C is based on the numerical use of the chain rule, in contrast to symbolic differentiation packages like MAPLE or REDUCE, or finite difference approaches. An explanation of chain-rule based automatic differentiation can be found in [2].

Given a C code for the computation of a function f and an input value x_o, ADOL-C uses the *operator overloading* of C++ to produce a trace of the elementary operations performed in computing $f(x_o)$. If $f(x) = y$, we call x the independent variables, and y the dependent variables. The operator overloading approach has the following advantages and drawbacks.

Advantages:

1. Only minimal modifications are required in the user program. Essentially one has to identify the dependent variables, the dependent variables, and redeclare the types of all intermediates.
2. Subprograms are handled without difficulties.

3. The trace produced by ADOL-C be used to compute derivatives $\frac{d^i}{x} f(x_o)$ for arbitrary order i.
4. Both the forward and reverse mode can be employed to compute derivatives. The forward and reverse mode of automatic differentiation are explained in detail in [2,1], but here it suffices to say, that in the forward mode we maintain at every intermediate value its derivative with respect to the independent values, whereas in the reverse mode we maintain the partial derivatives of the dependent values with respect to every intermediate value.

Drawbacks:

1. The tape is good only for a set of values in the neighborhood of x_o. That is, if loop bounds change or different branches of a conditional are taken, a new "tape" is needed.
2. The size of the tape is proportional to the number of floating-point operations performed, so it can be rather large. It should be noted, however, that the ADOL-C utilities access the tape in a completely sequential fashion and that the RAM storage requirements are a small multiple of the RAM requirements of the original code [2].
3. Since only elementary arithmetic operations are overloaded, the structure of the user program is not reflected in the tape. That is, if a subprogram g is called twice in the course of evaluation $f(x_o)$, the arithmetic operations performed in g will be recorded twice on the tape.

It should be noted that, except for the first point, the "drawbacks" are not an inherent limitation.

1.2 The Tape

ADOL-C introduces the *adouble* data type for differentiable quantities. Usual C `double` variables are treated as constants with respect to differentiation. For the *adouble* data type, ADOL-C currently overloads the elementary arithmetic operations `+, -, *, /, =, +=, -=, *=, /=`; elementary functions like `cos, sin, exp, sqrt`, and `tan`, and variable constructors and destructors and provides a mechanism to exploit the fact that a user-defined function is defined as an integral $f(x) = \int g(t)dt$. Whenever one of the elementary arithmetic operations or functions is executed, it is recorded on the tape.

To hold the values of the intermediate values arising during the computation, ADOL-C employs a so-called live variable table which contains the values that are still needed at this point in the computation. Which intermediate values are needed is determined through the overloading of the C++ variable constructors and destructors. This should become clear with an example:

```
adouble f(adouble x,adouble y, double z)
{ adouble t;
t = x + y;
if (2 * t > 2) {
   return(x*z)
   } else {
```

```
  return(sqrt(z/x))
} }
```

Thus, for the purposes of differentiation, f computes a function from R^2 into R (the parameter z is a constant with respect to differentiation since it is not an adouble). When we compute $f(1,2,3)$, the following actions are 'caught' by the operator overloading:

```
x = 1; y = 2; z = 3;
construct(t);
construct(temp1); /* allocate a temporary */
temp1 = x + y; t = temp1;
destruct(temp1);
construct(temp2);
temp2 = 2 * t;
destruct{temp2};
construct{temp3}
temp3 = z/x;
temp3 = sqrt(temp3); /* temp3 is returned as function value */
destruct(t); /* local variable dies when function exits */
```

An easy way to allocate space for the recording of the computation would be to give every variable (whether compiler-allocated or user-declared) a unique index in an array of doubles where one would store the value of the variable during the course of the computation. For this little example, we would use eight "slots" corresponding to the eight flops and assignments being performed, and the tape would look as follows:

Operation Code	Result	Input1	Input2	Constant
assign_independent	1			1.0
assign_independent	2			2.0
assign_double	3		3.0	
plus_adouble_adouble	4	1	2	
assign_adouble	5	4		
mult_adouble_constant	6	5		2.0
div_constant_adouble	7	1	3	
sqrt_op	8	7		
assign_dependent	8			

Here **x, y, z, temp1, t, temp2**, and **temp3** have been allocated slots 1 – 7 in that order. Tape record 7,for example, encodes the operation **store[7] := store[1]/store[3]**, when **store** is the array allocated for the storage of intermediate variables. The first column lists the operation code (see the Appendix), the second the entry in the live variable table indicating where to store the result, the third the index of the first input, and the last the index of the second input or a constant value.

We note that in order to perform the reverse mode of automatic differentiation, we have to store the numerical values of each intermediate quantity. This is easily done when we give each intermediate quantity a unique index in the table holding intermediate values.

One can, however, reduce the storage requirements for intermediate quantities, by exploiting the scoping information contained in the constructors and destructors. Loosely speaking, a "construct" call means that we need a new slot, and a "destruct" call means that we can reuse the slot that was assigned the variable that is being destructed. Using this information, we can now get by with four table locations, since `temp1`, `temp2`, and `temp3` can reuse the same storage location (`store[5]` in this case). The tape then stores the following information:

Operation Code	Result	Input1	Input2	constant
assign_independent	1			1.0
assign_independent	2			2.0
assign_double	3			3.0
plus_adouble_adouble	5	1	2	
assign_adouble	4	5		
death_notice	5			
mult_adouble_constant	5	4		2.0
death_notice	5			
div_constant_adouble	5	1	3	
sqrt_op	5	5		
assign_dependent	5			

A "death notice" indicates that this slot can be reused. In ADOL-C it also has the side effect of storing the value of this temporary onto an auxiliary tape of "death values" so that we can restore the value that was stored in a given slot when the corresponding variable died, and perform the reverse mode of automatic differentiation. Experience has shown that this technique usually reduces the size of the table for storing intermediate quantities to about 10% of the number of floating-point operations performed. The length of the tapes storing the operations and the death values is still proportional to the number of floating-point operations, but the tapes are accessed in a completely sequential fashion and can hence be stored out of core, whereas the table of variables currently in use has to be in RAM storage. Let us then introduce the following terminology: A "live" variable (which is being stored in the live variable table) corresponds to a value that may be an input argument to another arithmetic operation later on, whereas a "dead" variable (whose slot in the live variable table can be reused) will not be used any more in any further operations.

1.3 Format of the ADOL-C Tape

The exact format of the operation trace tape is as follows: The first eight fields contain integers printed in ASCII format in 10 bytes each (see macro LENGTH_TAPEHEAD_ENTRY in `macros.h`) denoting the following information:

1. the number of times the internal tape buffer was dumped to file,
2. the number of independent variables,
3. the number of dependent variables,
4. the size of the internal buffer used for writing to the file,

5. the maximum number of live variables,
6. the length of the tape in bytes,
7. the total number of operations recorded (not counting death notices), and
8. the number of nodes destroyed (while building the live variable table).

When generating the tape, ADOL-C buffers output to the file to speed up I/O handling.

To save space, the rest of the tape is stored in binary format and hence is readable only on the machine that generated it. Each tape entry consists of three parts:

1. An opcode. The C type for this entry is **optype**.
2. A variable length record, containing the information needed for this opcode. The C type for this entry is union type **t_type_record**, defined in **readtape.h**.
3. The length of the variable length record. The type for this entry is **rlentype**.

For example, the record for an immediate operator (i.e., x += y) involving another **adouble** is

```
struct a_same_arg {
  locint result,loc_1;
};
```

containing the index of x and y in the live variable table. A special type **locint** (usually defined to be a **short int**) is used to identify locations in the live variable table. All the various structures corresponding to the various operations are defined in **template.h**. The Appendix lists which opcodes correspond to which tape entry types. Some types that are not obvious are as follows:

Death Notices: **struct death_not_rec {**

```
    locint loc_1,loc_2;
  };
```

To avoid cluttering up the tape with death notices whenever a variable goes out of scope, a death notice is only generated when a certain number of variables in contiguous life variable locations (from location **loc_1** up to including **loc_2**, where **loc_2** is the highest currently used slot) have died. Thus, the live variable table behaves like a stack, in that we add and delete live variables only on the top.

Quadrature Functions: **struct quadrature {**

```
    locint result;
    double r_val;
    locint old_loc;
    locint deriv_loc;
  };
```

As mentioned before, the derivative of a function $f(x) = \int_a^x g(t)dt$ is easy to compute: it is simply $g(x)$. For example, $\arctan(x) = \int_0^x \frac{1}{1+x^2}$, and So $\frac{d}{dx}\arctan(x)|_{x=x_o} = \frac{1}{1+x_o^2}$. The fields in the quadrature record have the following meaning:

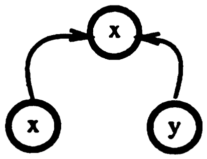


Figure 1: Graph Representation of **x+=y**

result: live variable slot where $f(x_o)$ is stored.

r_val: $f(x_o)$. We have to store $f(x_o)$ with the record, because $f(x_o)$ will be computed with some numerical quadrature routine and when we read the tape in the forward mode, we have to be able to restore the value at an intermediate quantity (but we may not have access to the quadrature routine any more).

old_loc: slot where x_o is stored.

deriv_loc: deriv_loc is the live variable slot where $g(x_o)$ is stored. In computing the derivatives, we have to evaluate $g(x_o)$, so there will be records on the tape to do just that.

2 Building the Execution Graph

In the execution graph, each node corresponds to some intermediate value computed during the execution of the program to compute $f(x_o)$. A graph node represents both an operation and the value that is the result of performing that operation with the given input values. We say that node c is a "child" of node p, if the value computed at c is an input value to the operation performed at p; p is called the "parent" of c. Since all operations are at most binary, each node has at most two children, but may have many parents. The computational graph is acyclic, with the leaves representing the independent variables and the roots representing the dependent variables.

It is important to understand that *a node in the computational graph represents a value,* not *a storage location.* For example, the operation `x+=y` will be translated into the graph shown in Figure 1. The child node labeled x represents the value in storage location x *before* the addition; the parent node labeled x represents the value in storage location x *after* the addition has been performed.

We can draw an analogy between ADOL-C's live variables and our graph nodes as follows: An ADOL-C `store` location is alive if its value may still be read as an input to another operation; this corresponds to the graph node that may still acquire parents. When a `store` location dies, its value will not be used any more; this corresponds to a graph node who will not acquire any more parents.

2.1 Overall Framework

The overall framework for building and representing the computational graph is shown in Figure 2. We have three main components:

Live Node Table: Each entry in this table points to a "live"graph node, that is, a node that may still acquire parents as nodes still to be constructed use its value as input. Further, each entry

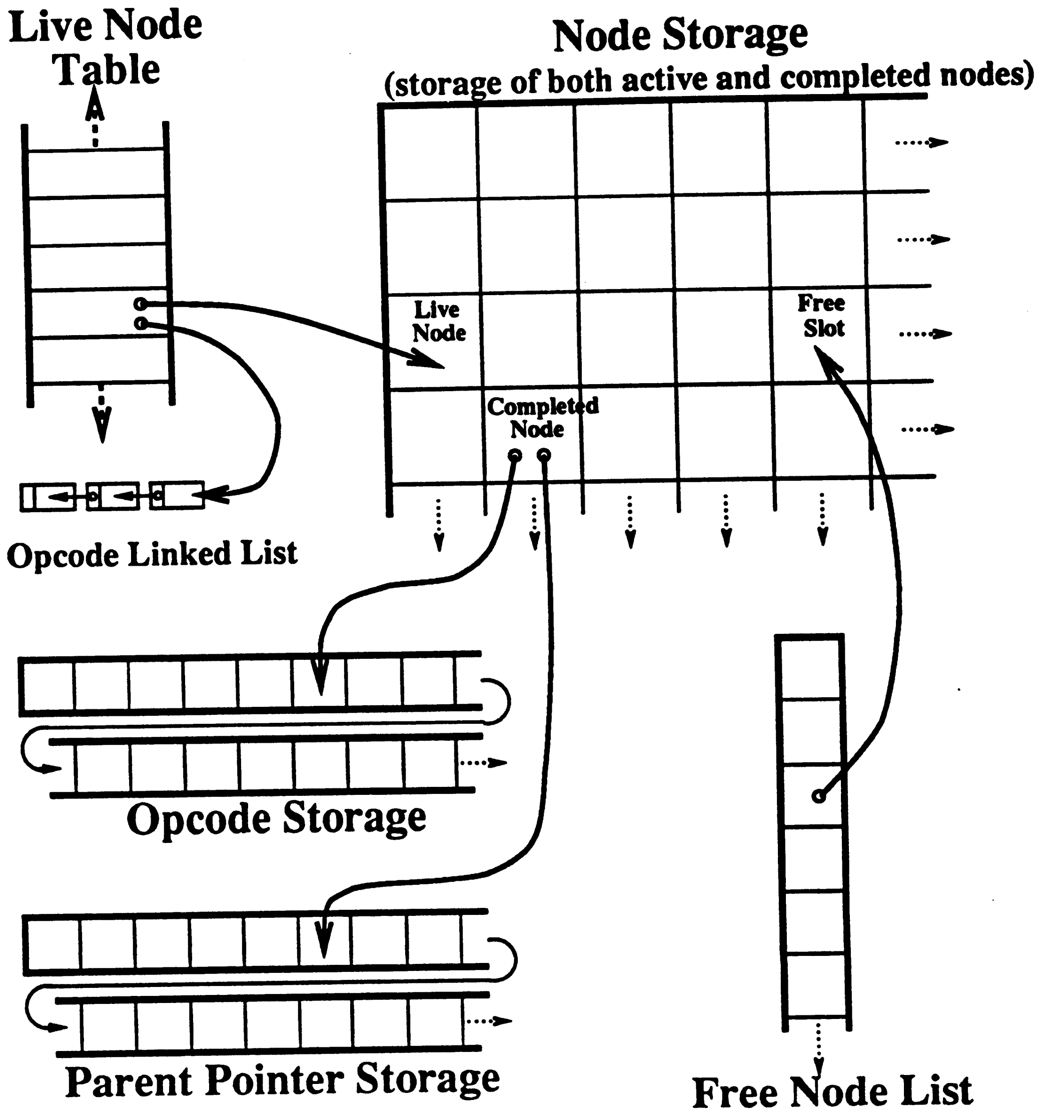


Figure 2: Implementation of Computational Graph

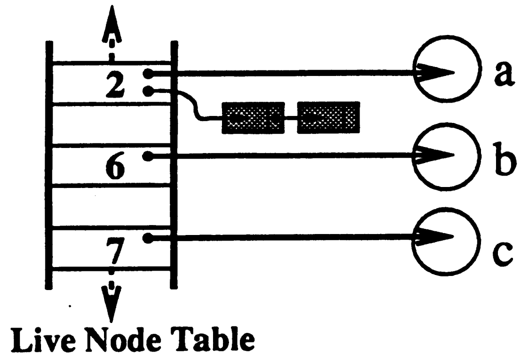


Figure 3: Situation before Inserting a New Node in Location 2

points to a linked list of (opcode,constant) pairs. While on the ADOL-C tape each tape entry corresponds exactly to one operation, we will perform an operation called "hoisting" (see §2.3) which may associate a chain of operations with an opcode.

Node, Opcode, and Parent Pointer Blocks: In order to minimize memory requirements and memory fragmentation, the graph structure is stored in three separate arrays. The node storage blocks store the core node structure `t_node`, described in more detail below. The opcode block stores (opcode,constant) pairs, and the parent pointer blocks store pointers to parents. Both opcode structures and parent addresses are stored consecutively for maximum memory usage.

Node Freelist: Nodes identified as dead roots (i.e, they have no influence on the values of the dependent variables, see §2.4) are removed from the graph by hoisting them into other nodes (see §2.3). As a result, the corresponding slot in a node storage block can be reused, and its address will be put on the freelist. The freelist is a stack implemented as a doubly-linked list of fixed-sized arrays. All stack operations can be performed in constant time, and the stack can grow as needed.

The rationale for this setup was both to minimize memory fragmentation and to exploit the locality that the creation of graph nodes is likely to exhibit. In other words, we wished to store the information pertaining to neighboring nodes close together in memory.

When we read a new record from the tape, we write out the graph node corresponding to the live variable slot that is being reused, and construct a new graph node that will be hooked up to that slot. For example, when we read the record corresponding to `store[2] := store[6] + store[7]`, we know that the value represented by the node (a, say) hanging off location 2 in the live node table (call this location `live[2]`), will not be used any more. This situation is shown in Figure 3. We copy the opcode list of node a into the opcode storage array (the opcode fields in the opcode list hanging off `live[2]` are set to UNUSED, indicated by NOP in the figure, so that they can be reused). (At this point we could try to perform some optimizations on node a; We return to this

```
t = a+b;
z = a*t;
y = t/b;
```

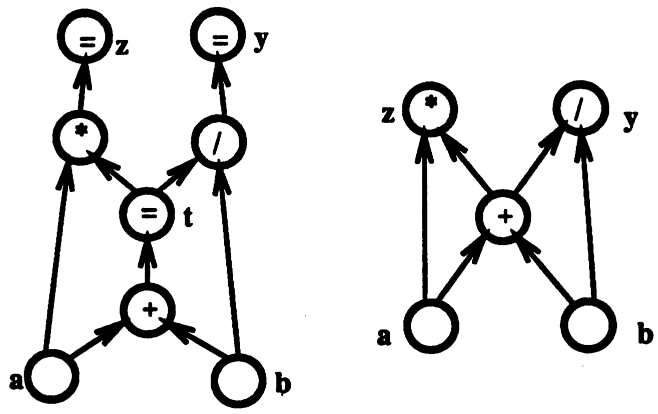


Figure 5: Sample Code Fragment and its Representation with and without Assignment Nodes

message, but allows the construction of the graph to continue.

2.2 Graph Data Structure

The data structure representing a graph node is as follows:

```
typedef struct node_type {
   unsigned int visited:1;     /* flag to mark during a traversal */
   unsigned int id;            /* unique identifier */
   struct node_type *left;     /* pointer to left child */
   struct node_type *right;    /* pointer to right child */
   unsigned int num_parents;   /* number of parents */
   unsigned int num_opcodes;   /* number of opcodes */
   unsigned int height;        /* max distance from a leaf */
   union {
     unsigned int blink;       /* back link to table entry */
     unsigned int slot;        /* index in array of intermediate values during
                                  serial evaluation */
   } b;
   union {
     unsigned int aliascount;  /* number of aliases to a node */
     unsigned int depth;       /* max distance from a root */
   } d;
   union {
```

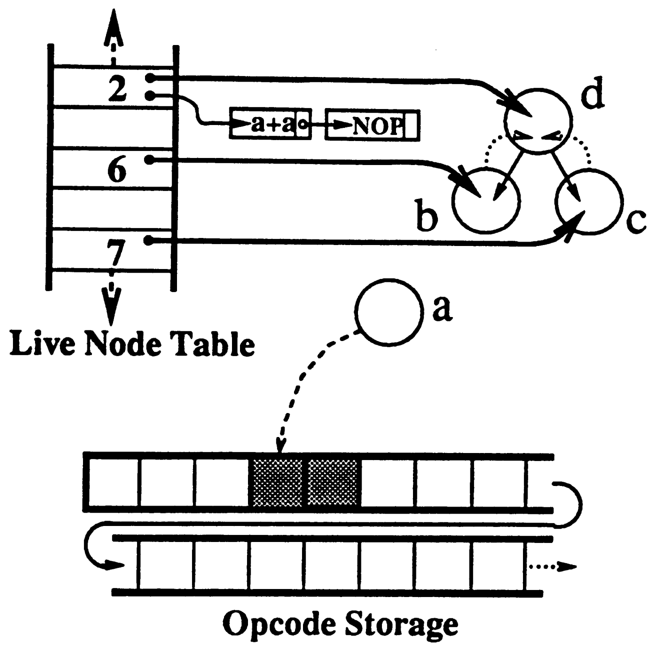


Figure 4: Situation after Inserting a New Node in Location 2

subject later.) We then allocate a new graph node (*d*, say), hook it up to `live[2]`, copy its opcode into the opcode list off `live[2]`, and make the nodes *b* and *c* pointed to by `live[6]` and `live[7]` the children of the new node. This is shown in Figure 4. Assignments between adoubles (as recorded with the `assign_a` opcode) are a special case, in that there is no reason to explicitly represent them in our graph. For example, an assignment of the form `store[4] := store[7]` can be represented by having `store[4]` point to the same node as `store[7]`. While this saves us storage, it creates complications when we wish to remove superfluous nodes. For example, if the value in `store[7]` on the ADOL-C tape had no influence on the dependent variables, we would assume that we can remove the node off `live[7]` at the point when `store[7]` is reused. But this is the case only when there is no live node entry pointing to the node currently pointed to by `store[7]`. Otherwise the removal of this node will create a dangling pointer. To avoid this problem, we associate a *reference counter* with every node, and we remove a node only when the reference counter indicates that there is only one live node still pointing to this node. Removing assignments does reduce the size of the graph that ultimately has to be represented. An example is shown in Figure 5.

In transforming the ADOL-C tape into a computational graph, we also exploit simple mathematical facts to reduce the number of opcodes that we will have to deal with later. For example, we have no `div_a_d` opcode denoting a division of an adouble by a constant. Since we store constants directly with the opcodes, we can represent this operation as a `mult_a_d` opcode (multiplication of an adouble with a constant) and store the reciprocal of the constant. These transformations reduces the number of opcodes from 40 to 19 and greatly simplifies the further system design.

Currently we also do not take into account the special structure of quadrature opcodes. We simply store them as unary operations, with `live[result]` being the parent of `live[old_loc]`. The subtree for the computation of the integrand that is hanging off `live[deriv_loc]` will later be removed as a spurious root.

Uninitialized variables create problems with building the graph, in the sense that if `store[2]` is an uninitialized variable, then `live[2]` does not point to any node, but nonetheless it can happen that in the user program this value is used as input for a later operation. If such a situation occurs, we call the routine `bogus_node`, which allocates a node, its value initialized to 0, and prints an error

```
        struct node_type **upptr;  /* pointer to array of parents */
        struct node_type *parent;  /* pointer to last referenced parent */
    } p;
    union {
        t_op opdef;                /* a single operation */
        t_op *opptr;               /* an array of operations */
    } o;
} t_node;
```

The fields have the following significance:

visited: A Boolean flag used during graph traversals.

id: A unique integer number identifying this node in the range from 0 to (total number of nodes - 1); used in debugging.

left, right: Pointer to left and right child, UNUSED if no such child exists. By convention, a unary operator has right==UNUSED.

num_parents: The number of parents.

num_opcodes: The number of arithmetic operations represented by this node.

height: The maximal distance from a root.

Union structures are used to allow for space sharing between variables that will never be used at the same time. Fields used in 'live' nodes (i.e. nodes that are still hanging off the `live` table) are

blink: If this node is pointed to by `live[i]`, then `blink == i`. If this graph node is completed, `blink == UNUSED`.

aliascount: If this node is pointed to from *k* entries, then `aliascount == k-1`.

parent: This is a pointer to the last parent of this node. In particular, this will be THE parent if the node had only one parent (this is important for hoisting).

Fields used in completed nodes are

slot: The slot in the storage area that this node will be evaluated in; defined in `live_analysis` (see §3.1).

depth: The maximal distance from a root.

upptr, parent: If a node has only one parent, its address is stored directly in parent. If a node has more parents, upptr points to the address in a parent storage block where the parents are stored in consecutive order.

opdef, opptr : If a node represents only one operation, it is stored directly in opdef. Otherwise, opptr points to the address in an opcode storage block where the opcodes are stored in consecutive order.

The core data structure representing the graph is as follows:

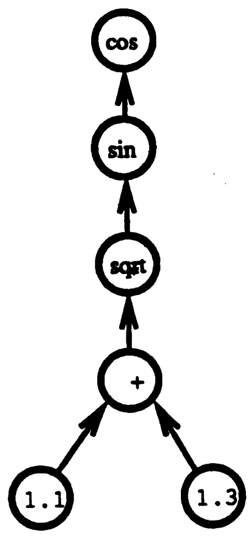
```
typedef struct graph_type {
   /**** core structure ****/
   t_nodelist *roots;           /* linked list of nodes that are roots */
   t_nodelist *rootcur;         /* root list cursor (points to the end of the list) */
   t_nodelist *leaves;          /* linked list of nodes that are leaves */
   t_nodelist *leafcur;         /* leaf list cursor (points to the end of the list) */
   unsigned int max_live;       /* max number of live variables */
   unsigned int slot;           /* storage slot in our storage allocation scheme */
   unsigned int num_roots;      /* number of dependent variables */
   unsigned int num_leaves;     /* number of independent variables */
   unsigned int num_nodes;      /* number of nodes in the graph */
   /**** for statistics ****/
   unsigned int num_opcodes;    /* total number of opcodes encountered */
                                /* (not counting assignments) */
   unsigned int num_assign;     /* number of assignments encountered */
   unsigned int num_hoisted;    /* number of nodes that have been hoisted */
   unsigned int num_deleted;    /* number of nodes that were dead roots */
   struct {
     int *hoists;
     int *parents;
     int *children;
   } histogram;
   unsigned int n_size;         /* memory (in bytes) used by node_blocks */
   unsigned int o_size;         /* memory (in bytes) used by op_blocks */
   unsigned int p_size;         /* memory (in bytes) used by parent blocks */
   /**** for debugging ****/
   t_node **lookup;             /* lookup array for the nodes */
} t_graph;
```

In the `histogram` structure, and the `lookup` pointer. `hoists[i]` contains the number of nodes that represent `i+1` operations, `parents[i]` contains the number of nodes having `i` parents, and `children[i]` contains the number of nodes having `i` children. `lookup` points to a table that, given the identifier number of a node, returns its address.

2.3 Hoisting

We already mentioned that assignments are not represented as nodes in our computational graph. Another optimization we perform is that we collapse chains of unary operations into one node, an operation that we call *hoisting*. As is depicted in Figure 6, hoisting allows us to represent a chain of unary operations much more succinctly. Instead of using five graph nodes with one opcode each, we represent this chain of operations in one graph node with five opcodes. Apart from saving memory, this operation increases the granularity of the graph in that more floating-point operations are associated with the evaluation of that particular graph node.

We can hoist a particular node n when its location in the `live` table is to be overwritten and the following conditions are all true:



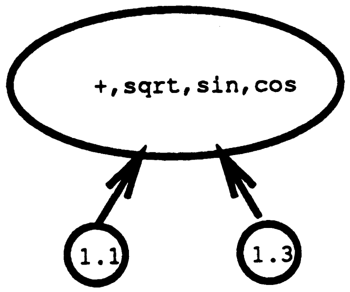


Figure 6: Sample Code Fragment and its Graph Representation before and after Hoisting

- *n has exactly one parent.*

- *We still have the address of the parent of n.* This may not be the case for the following reason: Assume that *n* had two parents, *p*1 and *p*2, and that *p*2 was the last parent added to *n*. So **p.parent** in *n* contains the address of *p*2. Then we remove *p*2 because it is a spurious root, decrementing *n*'s parent counter, and setting **p.parent** to NULL in *n*, since this parent was removed. We would have to traverse the graph in order to find out that *p*1 is the remaining parent of *n*, since we store only child pointers when we build the graph, and we considered the potential payoff not worth the effort. As a result we may not perform some hoists that would have been made possible through a dead root removal.

- *The parent of n does not represent a dependent variable.* These nodes represent the output values of the graph, and we wished to keep them uniform.

- *n's reference count indicates that only one link is pointing to this node.* If more than one pointer is pointing to *n*, it could still be used as an input value later on, and removing it would create a dangling pointer.

- *n is not a leaf.* Most leaves represent independent variables (the other alternative is constant initializations) and are pointed to from the leaf list. Hoisting them would create dangling pointers (unless we would update the leaf list, which we chose not to do).

- *n's parent has only one child.* We cannot hoist into a binary operation.

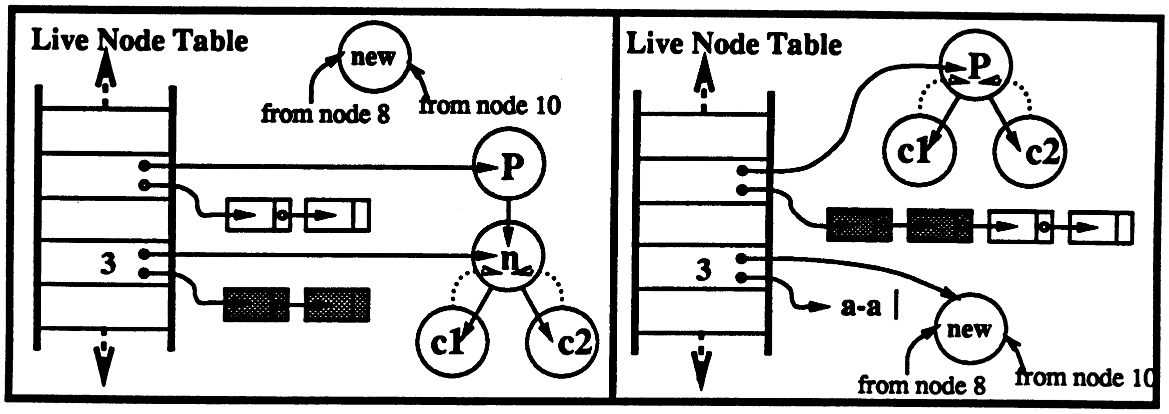


Figure 7: Normal Hoisting Process

- *The parent is still alive.* Live nodes have their opcodes in a linked list that hangs off the live node table, and so it is no problem inserting new opcodes. Completed nodes have their opcodes stored either in the node (if they only have one opcode), or in an array of fixed size, so we cannot easily insert a new opcode. If we wished to hoist into that node, we would have to find more space for the expanded opcode list, which we could do by copying it to the end, but again we considered this an unnecessary complication.

The mechanism is easiest to understand with an example as shown in Figure 7. Assume that the operation we read off the tape is a binary operation, e.g., `store[3] = store[8] - store[10]`. We then generate a new node *new* to represent this operation and have its children pointers point to the nodes off `live[8]` and `live[10]`. For the nodes off `live[8]` and `live[10]`, we make their parent pointer point to *new*.

Before having `live[3]` point to *new*, we check node n (which is about to be unlinked from `live[3]`) for hoistability. Now assume that the node n currently pointed to by `live[3]` is hoistable into node p as is shown in the left half of Figure 7. Hoisting involves updating p's children pointers, updating the number of children p has, splicing n's opcode list in at the beginning of p's opcode list, increasing p's opcode count, and updating the parent pointer for n's children (since they now have p instead of n as a parent). Thereafter n can be removed; that is, its address in a node storage area will be put on the free list. Lastly, we connect *new* to `live[3]`. The resulting situation is shown on the right side of Figure 7.

The situation may be more complicated with unary operators since the new node that is being generated can be the parent of the node to be hoisted (this cannot happen for binary operators since we will never hoist into a binary node). For example, assume that we read `store[3] = store[3] * 2.0` off the ADOL-C tape. We create a new node *new* to represent the new operation, and make its left child pointer point to the node pointed to by `live[3]` (call it n, say), and set n's parent

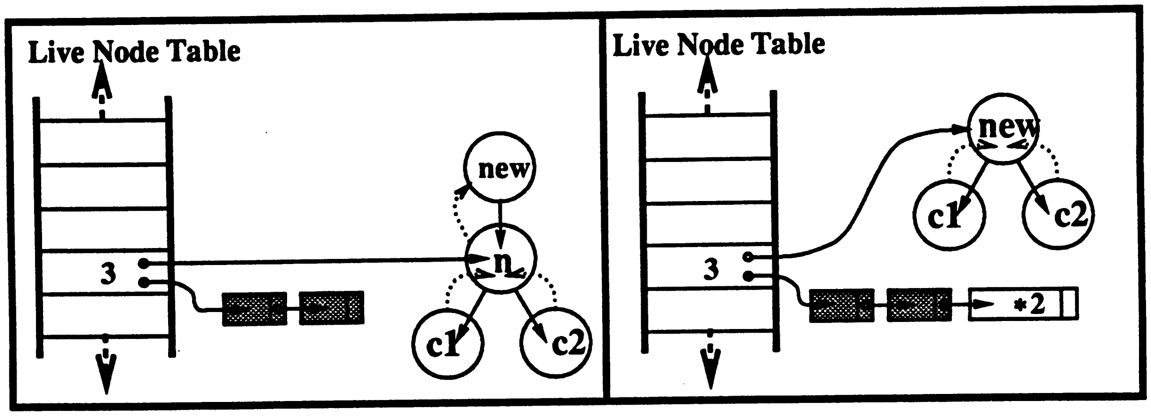


Figure 8: Special Case in Hoisting Unary Nodes

pointer to *new*. This situation is shown on the left side of Figure 8. Now we want to hoist *n* into *new*. Since *new* is not installed in the live node table yet, it does not have an opcode list yet, so in this case we insert the opcode corresponding to *new at the end* of *n*'s opcode list, and then proceed as in the case for the hoisting of a binary node. The resulting situation is shown on the right half of Figure 8.

2.4 Dead Root Removal

Another optimization that we perform is the removal of "dead roots", which are nodes whose value has no influence on the dependent variables. Most commonly, those nodes arise as a by-product of the evaluation of some control flow condition. As with hoisting, this optimization is tried when a node is about to be unhooked from the live node table.

A node can be removed from the graph with impunity if the following conditions are met:

- It does not have any parents.
- Only one live variable entry is pointing to it.
- It does not represent a dependent variable.
- It is not a leaf. The last condition ensures that we do not remove independent variables that are not used in the course of the computation (a more common occurrence than one would think).

If a node is a dead root, then we recursively descend to the nodes that are reachable from it, and remove all those *completed* children that have become dead roots themselves. If children are not completed yet (i.e., they are still pointed to from the `live` table), we quit, since they may still

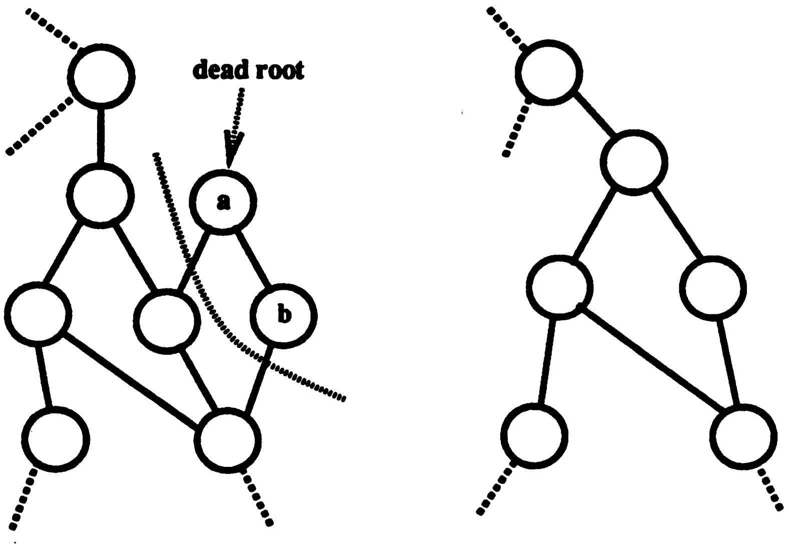


Figure 9: Removal of dead roots

acquire parents. If a node n is removed, we also set the parent pointer in its children to NULL if it had pointed to n before. An example is given in Figure 9, where the removal of dead root a triggers the removal of node b. Note that dead root removal is oblivious to the number of opcodes associated with a node; that is, supernodes are removed in the same fashion as normal nodes.

3 Related Utilities

In addition, we provided some utilities for evaluating the function defined by the computational graph, generate C code for evaluating the function, and to print out the graph structure for debugging purposes.

3.1 A Storage Allocation Scheme

To evaluate the function represented by the graph, we must allocate storage slots to the values represented by the graph nodes. The simplest procedure, of course, is to allocate a storage slot for every graph node, but we tried to do better. Our goal was to divide the graph into several *levels*, such that for all nodes the level of a child was less than the level of a parent. If all nodes on a level have different storage slots assigned to them, the following schedule will compute the function correctly:

```
for $i = 1$ to $max\_level$ do
   evaluate all nodes on level $i$ (in any order)
end for
```

The `live_analysis` routine implements an attempt to reuse storage slots while assigning them in such a way that the level scheme above works. Our approach is implemented using a double stack scheme:

```
put leaves on stack1
for $current\_level = 0$ to $max\_level$ do
   while (stack1 is not empty) do
      pop a node $n$ from the stack1
      if (level($n$) == $current\_level$) then
         assign a storage slot to $n$
         push all unvisited children of $n$ on stack2
         and mark them as visited
      else
         push $n$ on stack2
      end if
   end while
   stack1 $\leftrightarrow$ stack 2
end for
```

To determine when we can reuse storage slot k (say) that was assigned to a node n (say), we keep track of how many of n's parents have been assigned a slot. Once the last parent of n has been assigned a slot, we can reuse n's slot since its value will not be needed any more. It should be noted that we can reuse n's slot only at *the next level*, since we do not want to assume any particular order in the evaluation of the nodes on the same level.

There are several possibilities to define the "level" of a node in a fashion that is consistent with the child-parent partial ordering of the nodes. In the absence of any superior alternative, we chose the height of a node to define its level. The height of a node is defined in the routine `node_height`, so one could easily try out new numbering schemes.

3.2 A Code Building Utility

To test the validity of our storage allocation scheme and to get a feeling for how much overhead we encur when evaluating the tape in an interpretative fashion, we implemented a routine called `buildcode` which prints out C code to compute $f(x_o)$. Code for statements is written out by levels, with the storage locations determined by the live variable analysis described above. An example is given in Figure 10. The code in (a) yields the graph shown in (b). The generated code in (c) then results when the node levels are defined by heights.

We then used this code to develop a parallel execution schedule by placing the statements at each level into a parallel do-loop and putting a barrier synchronization point between each level.

3.3 Debugging Utilities

To help in debugging our codes, we implemented a utility that, given the unique number `node->id` that we assigned a graph node, prints out the subgraph that has this node as root. It also allows one to print information stored in subnodes. Another useful utility is the `eval_graph` routine, which evaluates the graph serially, using the storage locations determined by the scheme described in §3.1. If `eval_graph` computes the same function value $f(x_o)$ as ADOL-C, then we have a rather high degree of confidence that our graph structure and storage allocation scheme are correct.

4 Experimental Results

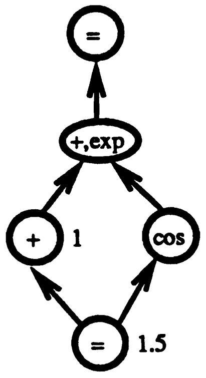
In this section we report on experimental results that we obtained by applying our graph generation, transformation, and evaluation utilities to ADOL-C tapes generated for the following three application codes:

Shallow: This code solves the shallow-water equation to simulate the development of the atmosphere in a rectangular region. We had 243 independent variables, corresponding to an initial state defined by a 9×9 grid with 3 variables at each node. Starting from this initial state, we integrated over 31 time steps. There is only one dependent variable, corresponding to the sum of squares between the measured and computationally predicted values.

Bratu: Bratu is a PDE model of the exothermic reaction in a section of a cylindrical combustion chamber. The code assumes radial symmetry and converts the problem to a two-dimensional grid with mixed boundary conditions. These results were obtained with a 40×80 grid of the chamber section, yielding 3,200 independent variables and 3,200 dependent variables.

```
x = 1.5;
t1 = 1 + x;
t2 = cos(x);
t2 += t1;
t3 = exp(t2);
y = t3;
```

(a)



(b)

```
/* code generated by readtape */

#include <math.h>
#include <stdio.h>

#define r_plus(x, y)    ((x) + (y))
#define r_minus(x, y)   ((x) - (y))
#define r_times(x, y)   ((x) * (y))
#define r_divide(x, y)  ((x) / (y))

double t[3];
double root[1];

int main() {
do_level_0();
do_level_1();
do_level_2();
do_level_3();
printf("root(%d) = %e;\n", 1, root[0]);
}

void do_level_0() {
t[0] = 1.5;
}

void do_level_1() {
t[1] = r_plus(1.0, t[0]);
t[2] = cos(t[0]);
}

void do_level_2() {
t[0] = exp(r_plus(t[1], t[2]));
}

void do_level_3() {
root[0] = t[0];
}
```

(c)

Figure 10: A Code Generation Example

Cavity: This problem is a discretization of an incompressible Navier-Stokes equation in a rectangle with constant fluid flow over one end of the rectangle. The rectangle is represented as a 31 x 31 grid, yielding 961 independent variables and 961 dependent variables.

4.1 ADOL-C Tape Statistics

Table 1 contains some statistics on the tapefiles as they were produced by ADOL-C, version 1.0. For each problem, we show how many instances of an operation were on the tape, what percentage of the total number of operations this corresponded to, how much storage (in bytes) operations of this type consumed, and what percentage of the total storage required for the tape this corresponds to. "Death notices" are not shown in this statistic. For "Shallow", we had 29,131 death notices; for "Bratu", 25459; and for "Cavity",12628; each death notice required 10 bytes on the tape. Together with the storage for the opcodes, shown in the "Totals" row of Table 1, the total storage required for "Shallow" is 4.1 Mbytes; for "Bratu", 3.4 Mbytes, and for "Cavity", 1.8 Mbytes. In contrast, the RAM requirements, determined by the maximum number of live variable, were rather modest. The maximum number of live variables was 18,365 for "Shallow"; 6,413 for "Bratu"; and 2,355 for "Cavity". These results correspond to 6.5%, 2.6%, and 1.6% of the numbers of operations performed.

4.2 Effect of the Graph Optimizations

In transforming the ADOL-C tape into a computational graph, we performed the three types of optimization previously described:

Assignment Removal: We deleted nodes containing the **assign_a** opcode by aliasing nodes through pointers and keeping reference counts.

Hoisting: We collapsed chains of operations into supernodes.

Dead Root Removal: We eliminated nodes whose values had no influence on the final results.

In Table 2 we show the effect of those optimizations. We display the total number of operations stored in the original ADOL-C tape, the number of assignments removed, the number of opcodes deleted as a result of dead root removal, the number of opcodes remaining in the final graph that have been amalgamated into a supernode as a result of our hoisting operation, and the remaining number of nodes in the optimized graph. We show both absolute and relative values. We should also note that the dead root elimination process did remove quite a few supernodes as well. For example, in "Cavity", we deleted 3,850 nodes, but eliminated 13,732 opcodes. We see that our optimizations have quite a noticeable effect. Between removing **assign_a** opcodes and dead roots, we eliminate 35.0%, 13.8%, and 45.9% of the operations to be performed. For emphasis these data are listed again in Table 3, where we show the number of opcodes in the optimized graph, the number of bytes that would be required to encode the optimized graph as an ADOL-C tape, and the percentage savings that this represents compared to the original ADOL-C operation count and storage requirements. Since the graph-building optimizations are applied on the fly, and employ ADOL-C's live variable analysis, these results suggest the use of on-the-fly assignment elimination

Table 1: Statistics of Original ADOL-C Tape

Shallow

opcode	# operations	% operations	# bytes	% bytes
assign_ind	3483	1.2	20898	0.5
assign_dep	1	0.0	6	0.0
assign_a	61236	21.7	612360	15.9
assign_d	7955	2.8	143190	3.7
eq_plus_a	24984	8.9	249840	6.5
plus_a_a	43062	15.3	602868	15.7
plus_a_d	10890	3.9	196020	5.1
min_a_a	30537	10.8	427518	11.1
mult_a_a	50823	18.0	711522	18.5
mult_a_d	48834	17.3	879012	22.9
Totals	281805	100.0	3843234	100.0

Bratu

opcode	# operations	% operations	# bytes	% bytes
assign_ind	3200	1.4	19200	0.6
assign_dep	3200	1.4	19200	0.6
assign_a	30578	13.8	305780	9.6
eq_plus_a	24178	10.9	241780	7.6
plus_a_a	36267	16.4	507738	16.0
plus_a_d	12089	5.5	217602	6.8
mult_a_a	12089	7 5.5	169246	5.3
mult_a_d	75734	34.1	1363212	42.8
div_d_a	12089	5.5	217602	6.8
exp_op	12089	5.5	120890	3.8
Totals	246972	100.0	3182250	100.0

Cavity

opcode	# operations	% operations	# bytes	% bytes
assign_ind	961	0.7	5766	0.3
assign_dep	961	0.7	5766	0.3
assign_a	49139	35.9	491390	27.9
assign_d	6284	4.6	87976	5.0
plus_a_a	19220	14.0	269080	15.3
plus_a_d	4933	3.6	88794	5.0
min_a_a	34976	25.5	489664	27.8
min_d_a	961	0.7	17298	1.0
mult_a_a	11532	8.4	161448	9.2
mult_a_d	8072	5.9	145296	8.2
Totals	149667	100.0	1762478	100.0

Table 2: Effect of Graph Optimizations

Shallow	Number	Percent
ADOL-C tape operations	281805	100.0
assign_a opcodes	61236	21.7
dead opcodes	37390	13.3
hoisted opcodes	29694	10.5
graph nodes	153485	54.5

Bratu	Number	Percent
ADOL-C tape operations	221513	100.0
assign_a opcodes	30578	13.8
dead opcodes	0	0.0
hoisted opcodes	48356	21.8
graph nodes	142579	64.4

Cavity	Number	Percent
ADOL-C tape operations	137039	100.0
assign_a opcodes	49139	35.9
dead opcodes	13732	10.0
hoisted opcodes	6144	4.5
graph nodes	68024	49.6

Table 3: Statistics of ADOL-C Tape Generated from Optimized Graph

	Shallow	Bratu	Cavity
number of opcodes	183179	190935	74168
% savings compared to ADOL-C opcodes	34.9	14.1	46.0
tape storage (bytes)	2703106	2876470	1063928
% savings compared to ADOL-C storage	29.7	9.3	39.8

Table 4: Number of Opcodes per Graph Node

# opcodes/node	**Shallow**	**Bratu**	**Cavity**
1	123791 (80.65%)	118401 (83.04%)	62008 (91.16%)
2	29694 (19.35%)	-	5890 (3.84%)
3	-	24178 (16.96%)	124 (0.08%)
4	-	-	2 (<0.01%)

Table 5: Compiled Graph Evaluation Code Scheduled by Height of Nodes

# processors	serial	1	2	4	6	8
execution time (secs)	0.80	0.97	0.57	0.58	0.54	0.6

and dead root elimination in the construction of the ADOL-C tapefile itself in order to reduce both the execution time for evaluating the tape and the storage required to create it.

The storage requirements for our graph and computational granularity are then further improved by hoisting. The size of the graph nodes is displayed in more detail in Table 4. We see that in the cases tested our supernodes usually have 2 or 3 opcodes and that the distribution is rather problem-dependent.

4.3 Generating Compiled Code from the Execution Graph

We tested our code building and parallel execution utility on the "Cavity"problem. For the purpose of parallel scheduling, we defined the level of a node by its height. All the nodes at a given height were then assigned to parallel processes in batches of 1000. So if we had 4,500 nodes, we generated five parallel processes; if we had 500 nodes, we generated only one process. Altogether we generated 67,924 lines of C-code, and we required 47,727 intermediate storage locations. We compiled this code on the Sequent Symmetry and obtained the (rather disappointing) results shown in Table 5. We see that the parallel code hardly does much better than the serial version. The main reason is that our execution schedule is rather unbalanced, as is shown by Figure 11 which shows the node-height distribution. As can be seen, the scope for exploiting parallelism within a given height is rather limited. Most of the time we have fewer than 3,000 nodes for a given level and cannot sensibly use more than three processes. It also turns out that our code is severely I/O bound. Each line of code is executed only once, so it takes rather long to fetch instructions from memory onto the chip. In particular, there is no locality of reference in the instruction cache.

Acknowledgments

We thank Andreas Griewank and Shawn Reese for many helpful discussions. We also thank Ted Gaunt for his help in gathering and evaluating the statistics.

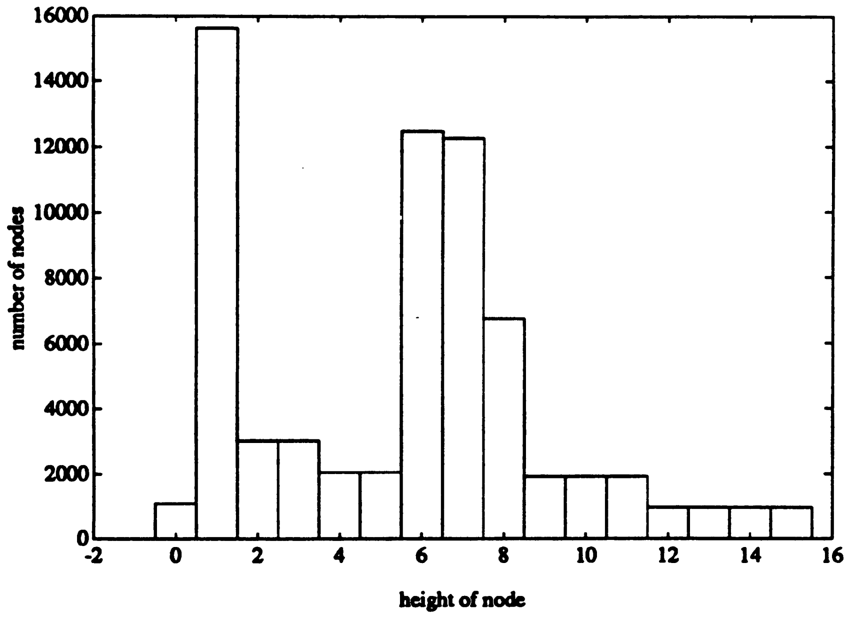


Figure 11: Level Distribution of 'Cavity' Graph

References

[1] Christian Bischof, Andreas Griewank, and David Juedes. Exploiting parallelism in automatic differentiation. Preprint MCS-P204-0191, Argonne National Laboratory, Mathematics and Computer Sciences Division, 1991.

[2] Andreas Griewank. On automatic differentiation. In *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Publishers, 1989.

[3] Andreas Griewank, David Juedes, and Jay Srinivasan. ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. Preprint MCS-P180-1190, Argonne National Laboratory, Mathematics and Computer Sciences Division, 1990.

Appendix: ADOL-C opcodes

This appendix lists the different opcodes (defined in `opcode.h`) and the name of the tape structure (defined in `template.h`) that is used for this opcode.

Assignment operators:

opcode assign_ind (structure d_assign_rec): Assignment to an independent variable

opcode assign_dep (structure dep_assign_rec): Assignment to a dependent variable opcode **assign_a (structure a_assign_rec):** adouble := adouble

opcode **assign_d (structure d_assign_rec):** adouble := constant

Elementary Arithmetic Operations:

Immediate Assignment Operators:
Opcodes of the form **eq_*op*_*type***, where *op* ∈ {**plus, min, mult, div**} and *type* ∈ {**d, a**}, corresponding to a constant (double), or adouble argument. The tape structures used are **a_same_arg** when the second argument is an adouble, and **d_same_arg** when the second argument is a constant.

Binary Operators:
Opcodes of the form *op_type1_type2*, where *op* ∈ {**plus, min, mult, div**} and *type1, type2* ∈ {**d, a, i**}, corresponding to a constant (double, adouble, or integer) argument. The tape structures used are **two_a_rec** for a binary operation involving two adoubles, and **args_i_a, args_a_i, args_a_d, args_d_a** depending on the type of the arguments.

Elementary Functions (structure single_op):
Opcodes **exp_op, log_op, sin_op, cos_op, tan_op, sqrt_op**, corresponding to e^x, $\log(x)$, $\sin(x)$, $\cos(x)$, $\tan(x)$, and $\sqrt{x}$, respectively.

Quadrature Functions (structure quadrature):

Elementary Quadrature Functions: Opcodes **atan_op, asin_op, acos_op**, corresponding to arctan, arcsin, arccos, respectively.

User-supplied Quadrature Functions: Opcode **gen_quad** denotes a user-supplied quadrature function.

Death Notice (structure death_not_rec) Opcode **death_not**.

Padding: opcode **ignore_me** corresponds to a record of type **padtype**. Padding records are needed to fill up the remaining bytes in the tape buffers before writing them out, and for properly aligning structures on word boundaries.