Issues in Parallel Automatic Differentiation*

Christian H. Bischof

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439–4801
bischof@mcs.anl.gov

Argonne Preprint MCS–P235–0491
Published in Automatic Differentiation of Algorithms, A. Griewank and G. Corliss, Eds., SIAM, Philadelphia, pp. 100-113, 1991.

Abstract. This paper shows how first-order derivatives can be computed in parallel by considering the computational graph that underlies the evaluation of the target function. The graph can be generated efficiently from the ADOL-C computational trace and can be used to automatically deduce the structure of the Jacobian matrix and compute the Jacobian using the reverse mode of automatic differentiation. By employing well-known graph-coloring techniques, one can dramatically decrease the number of reverse passes required. The resulting implementation performs well on the Sequent Symmetry and BBN Butterfly TC2000 shared-memory multiprocessors. Lastly, we look at the problems that must be tackled to make automatic differentiation a commonplace computing tool, and to allow for efficient implementations on high-performance computers. In our view, the key lies in finding better ways to incorporate user and/or compile-time information about the behavior of the program into the automatic differentiation approach.

1. Introduction. In this paper, we are mainly concerned with the evaluation of first-order derivatives on parallel machines. These techniques can easily be generalized to higher derivatives [12]. That is, given a function

$$F = \begin{pmatrix} F_1 \\ \vdots \\ F_M \end{pmatrix} : \mathbf{R}^N \rightarrow \mathbf{R}^M$$

and an input argument vector $x_o \in \mathbf{R}^N$, we wish to compute

$$\frac{\partial}{\partial x_i} F_j(x)|_{x=x_0}, i = 1, \ldots, N, j = 1, \ldots, M.$$

To this end we assume that we have a computational graph G, which represents the computation of $F(x)|_{x=x_o}$ in terms of the elementary arithmetic operations (like $+, -, *, /$) and standard library functions (like $\sin, \cos$). Figure 1 shows a sample program and the corresponding computational graph for the input values $x1 = 1$ and $x2 = 1.5$. This view of computation is a natural one, as it captures common subexpressions that can be exploited in the course of the computation of F. This point is demonstrated convincingly in [13]. This paper as well as [1] also gives an intuitive explanation of the forward and reverse mode of automatic differentiation within this framework. Let us assume that G contains T nodes n_k, with the first N nodes corresponding to the independent variables x_i, and the last M nodes corresponding to the dependent variables $F_j(x)$. In the forward mode, we associate with each node n_k the values

$$t_k = \nabla n_k(x)|_{x=x_0}$$

* This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, and through NSF Cooperative Agreement No. CCR-8809615.

```
if ((x1 - 2) > 0) then
   a = x1
else
    a = 2*x1
end if
b = 1
for i = 1:2 do
    b = b + sqrt(b)*a
end for
y0 = b/x2
y1 = a*x2
```

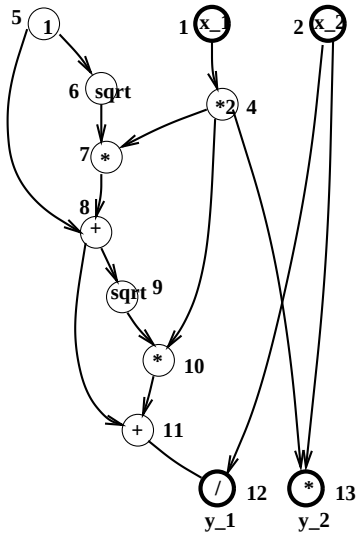

FIG. 1. *Sample Program and Corresponding Computational Graph*

which contain the partial derivatives of this intermediate value with respect to the independent variables. In the reverse mode, we associate with each node the values

$$\bar{t}_k = \frac{d}{d\, n_k} F(x)|_{x=x_o}$$

which measure the sensitivity of the dependent variables with respect to the intermediate quantity n_k. For simplicity we denote by n_k both the computational node and the intermediate value that it stands for.

The information needed to generate a computational graph corresponding to the computation of $F(x)|_{x=x_o}$ can easily be generated through operator overloading. The computational trace that is generated by packages like ADOL-C [14] corresponds to one particular topological ordering of G. In the next section we describe our approach for representing the computational graph generated from an ADOL-C tape, as well as the optimizations to reduce size of the graph and increase the computational granularity of the graph nodes.

The computational graph contains exactly all the synchronization constraints one must satisfy in the computation of $F(x)|_{x=x_o}$. That is, if there is a path from n_k to n_l, then n_l cannot be computed before n_k, but otherwise there is no restriction on the order in which graph nodes are evaluated. In Section 3 we show how we can exploit this freedom to compute F and its derivatives in parallel on a shared-memory multiprocessor. We also use the computational graph to *automatically* deduce the sparsity structure of $F'(x)|_{x=x_o}$ and then use graph coloring to identify component functions F_i which depend on disjoint subsets of independent variables. The gradients of component functions of different colors can then be evaluated in the same reverse pass over the graph, a procedure that can greatly enhance computational efficiency.

While promising, automatic differentiation schemes are still quite a ways from being serious

contenders for hand-coded derivatives. We survey current approaches to automatic differentiation in Section 4 and examine the issues that in our view must be tackled to make automatic differentiation a commonplace computing tool and to allow for efficient implementations on high-performance computers. As will be seen, the key lies in finding better ways to incorporate user and/or compile-time information about the behavior of the program into the automatic differentiation approach.

2. An Efficient Computational Graph Representation. We used the computational trace produced by the ADOL-C package [14] to generate a computational graph representing the computation of $F(x)|_{x=x_o}$. For example, the code fragment

```
t = plus(a,b);
z = 2*t;
```

would produce the following kind of trace if `plus(a,b)` was defined as `a+b`, and `a` and `b` were stored in s(1) and s(2), respectively.

```
construct(s(3), s(4));              /* allocate formal parameters
s(3) = s(1); s(4) = s(2);           /* assign actual to formal parameters
s(5) = s(3) + s(4); s(6) = s(5);    /* addition using one temporary variable
destruct( s(3), s(4), s(5) );       /* plus has gone out of scope
s(3) = s(6);                        /* assignment to t,
                                    /* reusing storage location 3
s(4) = 2.0*s(3); s(5) = s(4);       /* multiplication using temporary variable
```

Here the `s` array is used as RAM storage for the intermediate values that arise in the course of the computation. The simplest strategy of assigning storage locations to intermediate values would be to assign a unique location to every intermediate value, but the RAM storage cost needed for differentation utilities would be $O(T)$, in addition to the $O(T)$ cost for recording the T operations. In contrast, ADOL-C overloads the `C++` constructors and destructors, and hence reuses the storage locations assigned to a variable when this variable goes out of scope. This can be seen above, where we reuse `s(3), s(4)` and `s(5)` once the `plus` function has gone out of scope. Using this technique, the RAM requirements usually are rather modest. We also note that an operation like `z = 2*t` results in two assignments, one assigning `2*t` to a compiler-generated intermediary, which is then assigned to `z`. While this is clearly a compiler-specific phenomenon, the GNU `C++` compiler handles arithmetic operations this way.

In the computational graph, each node corresponds to some intermediate value computed during the execution of the program to compute $F(x)|_{x=x_o}$. 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, in Figure 2, 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.

To represent the computational graph efficiently, we use our own memory management to store graph nodes, opcodes, and pointers between children and parents in contiguous locations in memory. In this way, we avoid memory fragmentation and enhance data locality, since the information pertaining to neighboring nodes is likely to be stored in adjacent memory locations.

To decrease the size of the graph and increase computational granularity, we perform several optimizations on the fly as we construct the computational graph from the ADOL-C tape. In doing so, we incorporate some suggestions made in [3]. First, we eliminate assignments nodes. Recall that

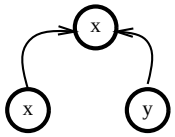

FIG. 2. *Graph Representation of* `x+=y`

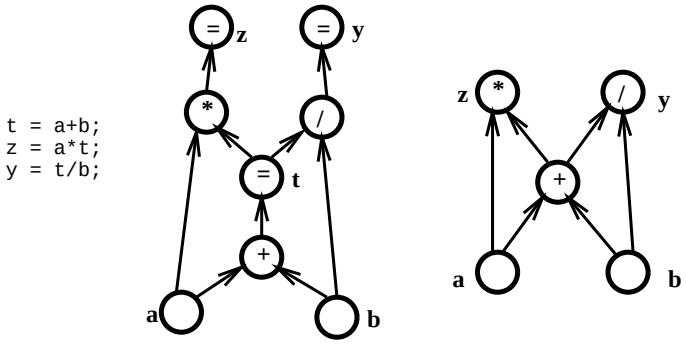

FIG. 3. *Sample Code Fragment and its Representation with and without Assignment Nodes*

a node in the graph corresponds to a value, not a memory location. Thus, if we use a variable `t` on the right-hand side of an arithmetic operation, we can simply generate a pointer to the result of the last arithmetic operation that was represented by `t`. An example is shown in Figure 3.

Second, we collapse chains of unary operations into one node, an operation that we call *hoisting*. As depicted in Figure 4, 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, a so-called supernode. 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.

Lastly, we remove so-called dead roots; 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. An example is shown in Figure 5. Upon encountering a dead root, we check recursively whether children of this node have become dead roots themselves. The implementation of those optimizations is nontrivial, since all these optimizations are performed on the fly; for details the reader is referred to [2]. On the other hand, the effect of those optimizations can be quite noticeable. We generated computational graphs for the ADOL-C tapes of the following three application codes:

Shallow: This code solves the shallow-water equation to simulate the development of the atmosphere in a rectangular region [21]. 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 partial differential equation model of the exothermic reaction in a section of a cylindrical combustion chamber [23]. 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.

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

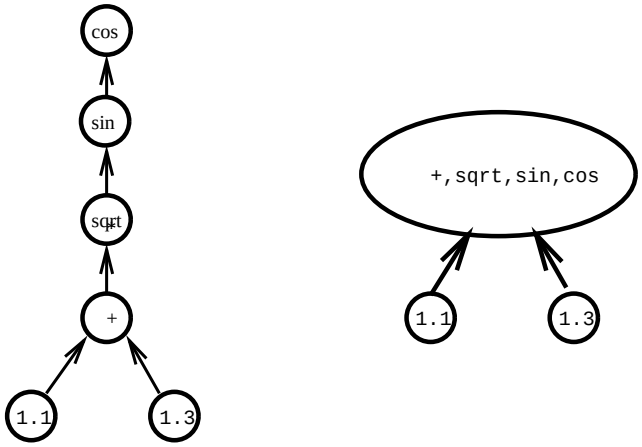

FIG. 4. *Sample Code Fragment and its Graph Representation before and after Hoisting*

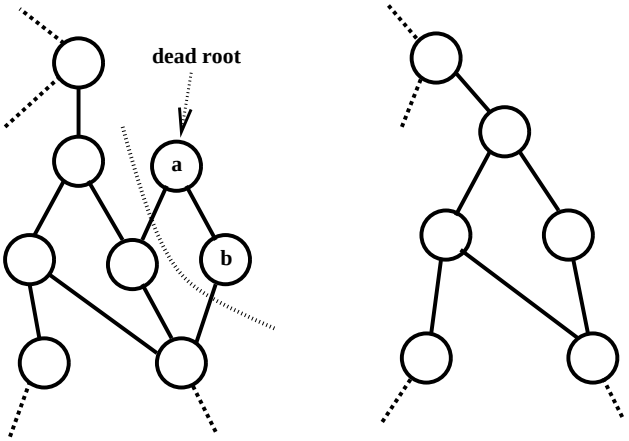

FIG. 5. *Before and After Removal of Dead Roots*

TABLE 1
Effect of Graph Optimizations

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

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

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

31×31 grid, yielding 961 independent variables and 961 dependent variables.
The total 'tape' storage required 4.1 Mbytes for "Shallow", 3.4 Mbytes for "Bratu", and "Cavity", 1.8 Mbytes for "Cavity". In contrast, the RAM requirements, ad indicated by the maximum number of storage locations, were rather modest. The maximum number of storage locations was 18,365 for "Shallow"; 6,413 for "Bratu"; and 2,355 for "Cavity".

In Table 1 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 removing dead roots, the number of operations 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 see that our optimizations have quite a noticeable effect. Between removing assignments and dead roots, we eliminate 35.0%, 13.8%, and 45.9% of the operations to be performed. By incorporating these on-the-fly techniques into the ADOL-C tape generation mechanism, we would have saved 29.7%, 9.3%, and 39.8% for the tape storage of "Shallow", "Bratu", and "Cavity" , respectively. The effect of hoisting was problem-dependent: 19.4%, 17.0%, and 8.8% of all operations were stored in supernodes for "Shallow", "Bratu", and "Cavity" , respectively, and usually a supernode contained two or three arithmetic operations.

3. Exploiting Parallelism. The execution graph now can be used to perform a forward or reverse sweep employing several processes. In a forward sweep, we start with the nodes representing the independent variables (the leaves of the computational graph), and make sure that we evaluate all children of a node before we evaluate this node itself. The situation is the opposite for the reverse pass; here we start with the dependent variables (the roots of the computational graph) and evaluate all parents of a node before we evaluate this node itself.

If the computational graph is stored in shared memory, we can compute first derivatives using

```
put roots on queue
repeat
    lock queue. Pick a node n from the queue. t̄_n = 0. Unlock queue.
    foreach parent p of n do
        [ t̄_n = t̄_n + (∂g_p/∂x_n) t̄_p ]  (*)
    end for
    foreach child c of n do
        lock c;
        if c has not been visited yet then
            initialize counter(c) to the no. of parents of c.
        end if
        counter(c) = counter(c) - 1;
        unlock c.
        if (counter(c) == 0) then
            lock queue; put c on queue; unlock queue;
        end if
    end for
until queue is empty
```

FIG. 6. *Dynamically Scheduled Evaluation of First-Order Derivatives Using the Reverse Mode*

the reverse mode of differentiation with the algorithm shown in Figure¡ 6. If we wish, for example, to compute the gradient $\nabla F_3(x)|_{x=x_o}$, we initialize the adjoint values in the root corresponding to F_3 to 1, and to zero in all other roots. See [1] for a detailed example.

With each graph node, we associate a counter which counts how many parents of a given node still have to be computed. When all parents of a given node have been evaluated, we can evaluate this node; to this end, we put it on a global queue, that contains nodes that are ready to be evaluated. The evaluation is then the summation of the adjoint values (*). Here g_p is the elementary function associated with node p.

In this simple form, the global queue will clearly become a bottleneck in the computation. Griewank and Juedes [17] avoided this problem by employing a hierarchical queue structure. Our implementation was inspired by their promising results but in addition, we wished to satisfy the following criteria:

- The parallel code had to be portable across shared-memory architectures.
- The mechanism for maintaining pools of "runnable" nodes had to be simple, yet efficient.
- The number of "locking" calls had to be minimized.

We achieved portability by implementing our code using the P4 portable communication library of Lusk et al. at Argonne. This library has been implemented on a variety of machines, and in particular on the Sequent Symmetry and the BBN Butterfly TC2000 shared-memory multiprocessors. To maintain "runnable" nodes, we collected them into packets, typically of size 20, and then transferred only packets of nodes between local process queues and one global queue. During graph evaluation, a node then consumes nodes from its local queue, and inserts newly generated runnable nodes into its local queue. If the local queues grow beyond a high-water mark, a certain number of packets are transferred to the global queue. If a process runs out of nodes in the local queue, it looks for a packet in the global queue. This scheme is simple, yet we found that it does the job of avoiding contention on the global queue. The number of "locking" calls is decreased through the use of supernodes generated

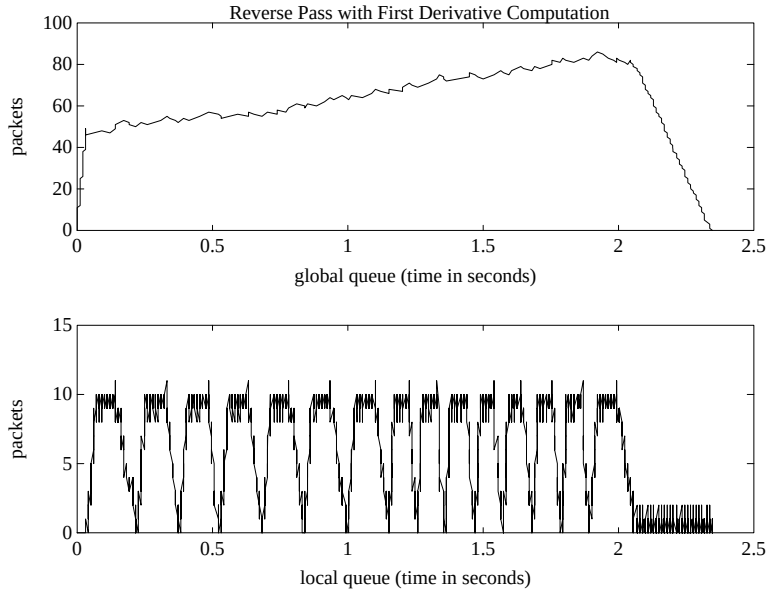

FIG. 7. *Queue Behavior during First Derivative Computation Using Reverse Mode*

through hoisting. Since the locking overhead required for evaluating a node is independent of the number of arithmetic operations associated with a node, hoisting improves computational efficiency by increasing the computational grain size. We also perform some straightforward optimizations that ensure that a node is locked only if another process could also access it.

We implemented two versions of our graph evaluator, which differ only in the way data is transferred between the local and global queues. In the first version, which we call "the pointer version", data is transferred between local and global queues simply by switching pointers. That is, all packets reside in shared storage, but they may be temporarily viewed as local in that only one process will access them. This version seems suited for a "flat" memory hierarchy as in the Sequent Symmetry, where all memory accesses essentially take the same amount of time. On the other hand, on the BBN Butterfly TC2000, shared-memory accesses may be up to ten times slower than local-memory accesses, since shared-memory accesses have to be routed through a switch, whereas local-memory accesses are immediately satisfied. In addition, local-memory accesses are cached, but shared-memory accesses are not. So in the second "copying version", we make sure that local queues do in fact reside in local memory, and we explicitly copy information between global and local queues.

It is instructional to consider the behavior of the local versus the global queue. In Figure 7 we show the total number of packets in the global queue and the local queue of a processor (as it turned out, the patterns of all local queues were virtually identical) as we evaluate the Jacobian of the "Cavity" test problem. We are using the "copying" version with three processes on the BBN TC2000. The packet size is 20, a local queue must not contain more than 11 packets; and if that limit is about to be exceeded, four packets are copied to the global queue. We observe a slow growth phase as packets are pushed onto the global queue. During that phase, there is substantial local activity between accesses to the global queue, as local processes seem to consume a "neighborhood" of the computational graph. As a result, there is virtually no contention on the global queue.

In our implementation we also exploit the fact that we can deduce the structure of the Jacobian matrix from the computational graph in an automatic fashion. If there is a path from the ith

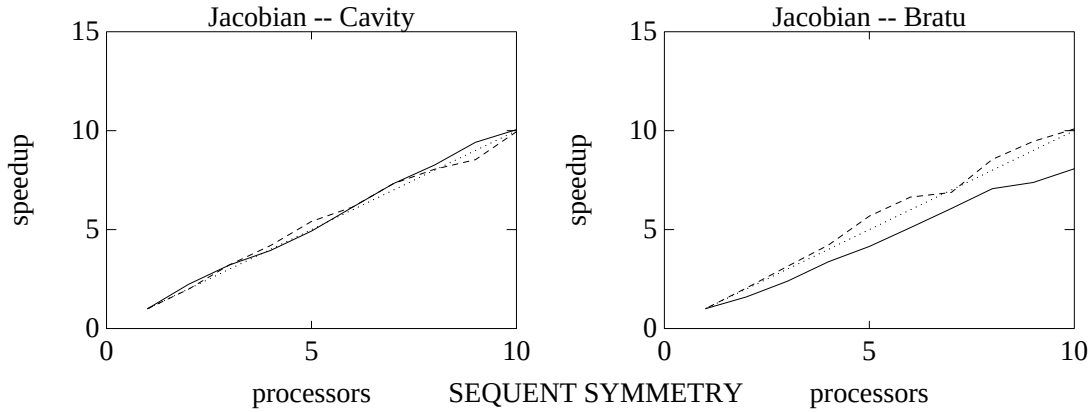

FIG. 8. *Speedup of "Pointer" (solid line) and "Copying" (dashed line) Version on Sequent Symmetry*

leaf (corresponding to the ith independent variable x_i) to the jth root (corresponding to the jth dependent variable F_j), then $\frac{\partial F_j}{\partial x_i} \neq 0$ — in the absence of numerical cancellation. This is one of the important fringe benefits of automatic differentiation. While current software for solving nonlinear least squares problems (see, for example, [6, 19, 20]) requires the user to specify the sparsity structure of the Jacobian — a tedious and error-prone process unless the structure of the Jacobian is very regular — the dependency analysis that is performed as a by-product of automatic differentiation automatically takes care of that.

Given the structure of the Jacobian, we can now identify rows of the Jacobian that can be computed independently. In essence, if F_i and F_k depend on different sets of input values X_i and X_j, respectively ($X_i, X_j \in \{1, \ldots, n\}, X_i \cap X_j = \{\}$), then ∇F_i and ∇F_j can be evaluated at the same time with the reverse mode, by seeding the adjoint values in the nodes corresponding to the dependent variables F_i and F_j with 1, and all other adjoint values in the dependent variables with 0. Upon completion of the reverse pass, the adjoint values corresponding to the independent variables in X_i will be the nonzero gradient values for F_i, and the adjoint values in the independent variables in X_j will be the nonzero gradient values for F_j. This structure has been exploited in finite-difference approximations of the Jacobian, and graph coloring algorithms have been used to identify maximal sets of component functions that depend on mutually disjoint sets of input values [5, 4, 9, 18, 22]. In our implementation we use the sequential coloring algorithm by Coleman, Garbow, and Moré [5], but we also mention that recent research has shown that this step can be efficiently implemented in parallel as well [15]. For example, in the cavity problem we require 21 "colors", that is, we can compute the gradients for all 961 component functions in only 21 reverse passes through the computational graph. For the "Bratu" problem, which employs a nine-point stencil in its PDE discretization, we get by with nine colors as expected.

The speedups obtained for the "Cavity" and "Bratu" problems on up to 10 processors of the Sequent Symmetry and BBN Butterfly TC2000 are shown in Figures 8 and 9, respectively. The solid line corresponds to the "pointer" version, the dashed line to the "copying version". We see that on the Sequent we obtain very good speedup, with the "copying version" performing somewhat better for the "Bratu" test problem. We seem to obtain superlinear speedup only because these experiments were not performed on dedicated machines. On the other hand, on the BBN Butterfly TC2000, we observe a noticeable degradation of performance, with the "pointer" version being clearly inferior to the "copying" version. This degradation is due to the TC2000 architecture, which has a archical memory structure in which shared data (in particular our graph structure) is more costly to

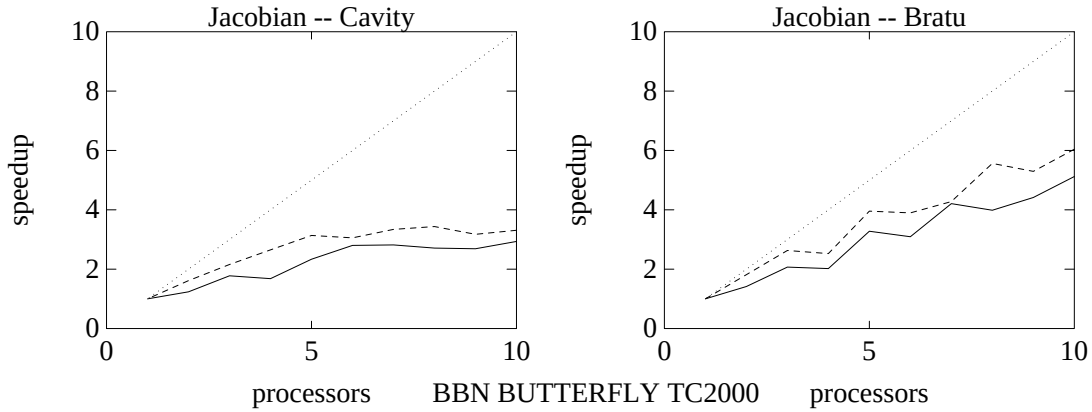

FIG. 9. *Speedup of "Pointer" (solid line) and "Copying" (dashed line) Version on BBN Butterfly TC2000*

access. In addition, our efforts to enforce locality on our graph data structures are a disadvantage, since the Butterfly's memory allocation routines (which are called by P4) currently do not scatter data across different memory modules (this will be remedied in the next software release).

4. Outlook. Automatic differentiation using approaches based on the chain rule based approaches is a very convenient software tool. Given just the code for the function, we can compute derivatives of any order (as well as the structure of derivative matrices) exactly (in contrast to finite differences) in a fashion that is transparent to the user. A survey of currently available packages is given in [16]. This property makes chain-rule based automatic differentiation a natural candidate for packages that require derivative values. In addition, our work as well as that of Christianson [3] and Dixon [7] has shown that there is scope for exploiting parallelism in the automatic computation of derivatives, again in a fashion that is transparent to the user.

On the other hand, even though this technique is used many different fields (see the papers in these proceedings), it has by no means become a standard computing tool yet. If one asks the typical computational scientist about automatic differentiation, he is likely to associate this term with finite difference approximations or symbolic techniques. The reason is that in all likelihood he has software employing finite-difference approximations or symbolic techniques on his computer — be it a PC, workstation, mainframe or supercomputer — but no software that supports chain-rule based automatic differentiation.

For this technique to become the commonplace computing tool that it ought to be, we need software that fulfills the following criteria:

Ease of Use: In particular, Fortran subroutines should be easy to interface with, since most scientific applications are implemented in this language.

Speedy Execution: Theoretically, the evaluation of the gradient with the reverse mode of automatic differentiation should require no more than five times the effort of evaluating the underlying function itself [10], and observed running times should be close to that factor.

Moderate Storage Requirements: The storage requirements for performing the forward as well as the reverse mode should be a modest multiple of the storage required for evaluating the functions.

Unfortunately, as useful as they are in many respects, all packages surveyed in [16] are suboptimal with respect to some of these criteria. Fortunately, the method is not at fault. It is our belief that one can develop automatic differentiation software that satisfies these criteria, if one can get

```
c$parallel
      do 10 i = 1,n
         y(i) = 0
c$vector
         do 20 j = 1,n
            y(i) = y(i) + a(i,j)*x(j)
         20 continue
      10 continue
```

FIG. 10. *Code fragment for computing* $y = Ax$

compiler support for automatic differentiation. For example, through the use of operator overloading techniques in `C++` or Ada, or PASCAL-SC, we can easily deal with functions written in those languages. On the other hand, Fortran functions require the use of precompilers, which usually support only a subset of the language. If we were to build the restricted set of operator overloading techniques needed for our purposes into a Fortran compiler, we could deal with Fortran functions with the same ease. Fortran90 will have most of that functionality, but at the moment it is unclear when this language will be as commonplace as Fortran77 is now.

The compiler issue also comes into play when we are concerned about running times. For example, the main computation inside the Helmholtz energy function

$$f(x) = RT \sum_{i=1}^{n} x_i \log \frac{x_i}{1 - b^T x} - \frac{x^T A x}{\sqrt{8} b^T x} \log \frac{1 + (1 + \sqrt{2}) b^T x}{1 + (1 - \sqrt{2}) b^T x}$$

is the matrix-vector product Ax, which might be computed in Fortran as shown in Figure 4 The Fortran directives instruct the compiler to vectorize the `j` loop and indicate that the various iterations of the `i` loop can be scheduled concurrently if the hardware supports it. With current techniques, this information, which can be so crucial for performance, would simply be ignored. The `j` loop would be evaluated at serial speed at best; and, if we were to use the graph scheduling techniques we described in the preceding section, we might rediscover the instructions in the parallel loop, but at significant cost. Again, if we were to incorporate differentiation arithmetic into a compiler, it would be easy to generate vectorized differentiation or parallel automatic differentiation code for such source code.

Lastly, there is the issue of storage requirements. In order to perform the reverse mode of automatic differentiation, all current implementations store a trace of the computation; and as we have seen in our examples, this trace is usually rather large. Even though the trace is accessed sequentially, the large storage requirement can quickly make this technique infeasible on small computer systems and/or large problems. There is, however, a way around this problem. Let us consider the following code fragment:

```
x = f(a);
y = g(x);
```

To implement the reverse mode of automatic differentiation, we currently evaluate f and g, recording all the arithmetic operations performed, and then run backwards over the computational trace. If $T(f)$ and $T(g)$ are the number of arithmetic operations required for evaluating f and g, this technique requires $T(f) + T(g) + \gamma(T(f) + T(g))$ arithmetic operations ($1 \leq \gamma \leq 5$ being the factor by which the reverse mode is more expensive than the function evaluation) and $O(T(f) + T(g))$ storage for recording the trace. If, on the other hand, we

1. record a;

TABLE 2
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

2. evaluate $f(a)$ *without tracing*;
3. evaluate $g(x)$ *with tracing* ;
4. do reverse pass on trace of g, until we come to the point where x was assigned;
5. re-evaluate $f(a)$ *with tracing*; and
6. complete the reverse pass on f's trace,

the arithmetic cost is $2 * T(f) + T(g) + \gamma(T(f) + T(g))$, but we require only $O(\min(T(f), T(g)))$ storage for tracing. Of course, we could now further reduce storage by applying this same technique in a recursive fashion to f and g themselves.

Such an approach has been used in hand-coded implementations of the reverse mode. Griewank [11] recently analyzed such a scheme, showing that if one accepts an increase in the number of operations by a fixed factor k, the storage required for tracing is limited essentially by the k-th root of the original run-time T. For the particular choice $k = \ln(T/R)$, where R is the RAM storage requirement for evaluating the original function, the tracing storage requirement and computing time both grow logarithmically in the ratio T/R. In his analysis, Griewank assumed that one could insert checkpoints arbitrarily during the computation; but, as suggested by the example above, subroutine boundaries might be the natural places to put those checkpoints. Again, compiler support would be crucial here.

In summary, we believe that automatic differentiation can become the generally accepted computing tool it deserves to be. For this to happen, we have to make the compiler community aware of this opportunity, and educate ourselves to understand in more detail exactly what functionality (and in what form), we could expect from compiler-supported automatic differentiation.

Lastly, let us return to the issue of parallelism, which is of crucial importance for the long-term viability of automatic differentiation. We essentially have two (not mutually exclusive) choices:

- We can exploit parallelism through *splitting of the computational graph*. This is what we have done in the implementation we reported on in the preceding sections.
- Alternatively, we can exploit parallelism through *parallelizing the computations associated with each graph node*. This has been done by Dixon [7], where the gradient computation associated with each node in the forward mode is done elementwise in parallel (see also [8]).

Even though by no means trivial, the latter alternative is comparatively straightforward, since it is clear which parts of the problem can be done in parallel. On the other hand, extracting parallelism in the evaluation of the computational graph is more difficult.

The main problem is one of computational granularity. The dynamically scheduled algorithm in Figure 6 requires considerable overhead in comparison with the main task, which is the accumulation of the adjoint values in (*). In an attempt to avoid this locking and scheduling overhead, we generated a statically-scheduled version for computing $F(x)|_{x=x_o}$ in the following fashion: First we grouped the computational graph nodes by their maximum distance from a root (this is commonly called the "height" of a node). For each node in the graph, we then generated a line of C code, and we grouped lines corresponding to nodes of the same height in batches of 1000. Between each height we placed a barrier, thereby forcing synchronization of all processes. On the "Cavity" problem, we thus generated 67,924 lines of C code and required 47,727 intermediate storage locations. We compiled this code on the Sequent Symmetry and obtained the (rather disappointing) results shown in Table 2. We see that the parallel code hardly does better than the serial version. The main reason is that our execution schedule is rather unbalanced, as is shown by Figure 11 where shows

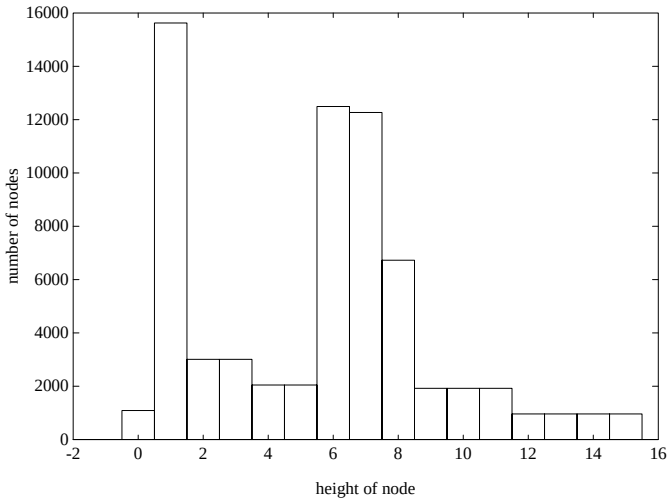

FIG. 11. *Level Distribution of 'Cavity' Graph*

the node-height distribution is plotted. 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.

The crucial issue is again one of granularity. In our current computational graph, we associate only a few flops with each graph node, which is simply not enough to lead to a parallel implementation that could seriously compete with a hand-coded version. What we need is a coarser computational graph, where each node corresponds to a subroutine invocation or a loop nest, say. By being able to integrate supernodes that correspond to a function g (say), we can then integrate the storage saving schemes mentioned above, and the overhead for scheduling the evaluation of this node would be amortized over many floating-point operations. In addition, we would be able to optimize the evaluation of $g(x)$ or $\frac{dg}{dx}$. For example, we could exploit the following:

Known derivatives: We might know the derivative explicitly, either by computing it by hand or by applying symbolic techniques.

Library software: Common mathematical operations are usually supplied in assembler libraries by the vendors, and great increases in speed can be achieved through the use of those libraries. Take again the Helmholz energy function. The main kernel is $y = x^T Ax$, and $\nabla y = 2Ax$. By inserting the matrix-vector multiplication subroutine, we are certain to compute this derivative significantly faster than through any of the current automatic derivation techniques.

Parallelism within g: We might know an efficient way of parallelizing g, which we can also exploit in the derivative computation.

Again, we are trying to capture user insight in the automatic derivation system. As has been shown through our prototype implementation, we can compute derivatives completely automatically, but at a cost. By capturing what user intuition is available, we stand to greatly reduce this cost.

Acknowledgments. We thank Ted Gaunt and James Hu for their dedicated effort in this project, and Andreas Griewank for many helpful discussions.

REFERENCES

[1] Christian Bischof, Andreas Griewank, and David Juedes. Exploiting parallelism in automatic differentiation. In Elias Houstis and Yoichi Muraoka, editors, *Proceedings of the 1991 International Conference on Supercomputing*, pages 146–143, Baltimore, Md., 1991. ACM Press.

[2] Christian H. Bischof and James Hu. Creating and optimizing a computational graph for algorithmic decomposition. ANL/MCS-TM-148, Argonne National Laboratory, Mathematics and Computer Sciences Division, 1991.

[3] Bruce Christianson. Automatic Hessians by reverse accumulation. Technical Report No. 228, The Hatfield Polytechnic, Hatfield, U.K., 1990.

[4] T. F. Coleman and J. J. Moré. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM Journal on Numerical Analysis*, 20:187–209, 1983.

[5] Thomas F. Coleman, Burton S. Garbow, and Jorge J. Moré. Software for estimating sparse Jacobian matrices. *ACM Transactions on Mathematical Software*, 10(3):329–345, 1984.

[6] John Dennis and Robert Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinjear Equations*. Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

[7] L. C. W. Dixon. Automatic differentiation and parallel processing in optimization. Technical Report No. 176, The Hatfield Polytechnic, Hatfield, U.K., 1987.

[8] Herbert Fischer. Automatic differentiation: Parallel computation of function, gradient and Hessian matrix. *Parallel Computing*, 13:101–110, 1990.

[9] D. Goldfarb and P.L. Toint. Optimal estimation of Jacobian and Hessian matrices that arise in finite difference calculations. *Mathematics of Computation*, 43:69–88, 1984.

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

[11] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. Preprint MCS–P228-0491, Argonne National Laboratory, Mathematics and Computer Sciences Division, 1991.

[12] Andreas Griewank. *Automatic Evaluation of First- and Higher-Derivative Vectors*, volume 97, pages 135–148. Birkhäuser Verlag, Basel, Switzerland, 1991.

[13] Andreas Griewank. The chain rule revisited in scientific computing. Preprint MCS–P227–0491, Argonne National Laboratory, Mathematics and Computer Sciences Division, 1991.

[14] 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.

[15] Mark T. Jones and Paul E. Plassmann. A parallel graph coloring heuristic. Preprint ANL/MCS-P246-0691, Argonne National Laboratory, Mathematics and Computer Sciences Division, 1991.

[16] David Juedes. A taxonomy of automatic differentiation tools. In Andreas Griewank and George Corliss, editors, *Proceedings of the Workshop on Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, Philadelphia, 1991. SIAM. to appear.

[17] David Juedes and Andreas Griewank. Implementing automatic differentiation efficiently. ANL/MCS–TM–140, Argonne National Laboratory, Mathematics and Computer Sciences Division, 1990.

[18] Jorge J. Moré. On the performance of algorithms for large-scale bound constrained problems. In *Large-Scale Numerical Optimization*, pages 31–45, Philadelphia, 1990. SIAM.

[19] Jorge J. Moré, Burton S. Garbow, and Kenneth E. Hillstrom. Implementation guide for MINPACK-1. Technical Report ANL–80–68, Argonne National Laboratory, Mathematics and Computer Sciences Division, 1980.

[20] Jorge J. Moré, Burton S. Garbow, and Kenneth E. Hillstrom. User guide for MINPACK-1. Technical Report ANL–80–74, Argonne National Laboratory, Mathematics and Computer Sciences Division, 1980.

[21] I. M. Navon and U. Muller. FESW — a finite-element Fortran IV program for solving the shallow water equations. *Advances in Engineering Software*, 1:77–84, 1979.

[22] Paul E. Plassmann. Sparse Jacobian estimation and factorization on a multiprocessor. In T. F. Coleman and Y. Li, editors, *Large-Scale Optimization*, pages 152–179, Philadelphia, 1990. SIAM.

[23] K. H. Winters and K. A. Cliffe. A finite element study of driven laminar flow in a square cavity. Technical Report AERE – R 9444, AERE Harwell, Theoretical Physics Division, 1979.