Implementing Automatic Differentiation Efficiently

David Juedes
Andreas Griewank

CRPC-TR90074
October, 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, Illinois 60439-4801

ANL/MCS-TM-140

Implementing Automatic Differentiation Efficiently

by

David Juedes and Andreas Griewank*

Mathematics and Computer Science Division
Technical Memorandum No. 140

October 1990

*Permanent address: Iowa State University, Ames, IA 50011. This author was a participant in the Spring 1990 Science and Engineering Research Semester Program, which is coordinated by the Division of Educational Programs, Argonne National Laboratory.

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.

Contents

Implementing Automatic Differentiation Efficiently

by

David Juedes and Andreas Griewank

Abstract

The automatic differentiation of computer arithmetic has been investigated since before 1960. Most of this effort has been centered on the *forward* mode of derivative evaluation. Speelpenning, Iri and Kubota, and Horwedel et al. have all implemented the more efficient *reverse* mode of evaluating derivatives in their respective Fortran precompilers. The *reverse* mode requires information about a computation to be stored in order that derivatives may be calculated after the function evaluation has been completed. This additional storage cost may be prohibitive. The goals of our research have been to implement an automatic differentiation package that gracefully handles the storage issue and to generate a parallel implementation of derivative evaluation in the reverse mode. This paper discusses results of both research efforts, specifically those involving the ADOL-C package.

1 Introduction

The automatic differentiation of computer arithmetic was first investigated by Beda et al. [Beda59] and Wengert [Wen64]. Since then there have been various implementations of automatic differentiation. Most of these implementations have concentrated on the simple *forward* evaluation of derivatives. For scalar functions of the form $y = F(x_1, ...x_n)$, the forward evaluation of partial derivatives requires $O(n)$ times the execution time of the original function. Speelpenning [Spe80] mentioned and Baur and Strassen [BS83] later published a proof that the number of operations required to compute a scalar function and its partial derivatives is bounded above by a fixed constant times the number of operations required to compute the function. This theoretical result leads to the more efficient *reverse* mode of derivative evaluation. Speelpenning [Spe80], Iri and Kubota [IK87], and Horwedel et al. [Hor88] have all implemented the *reverse* mode of evaluating derivatives in their respective Fortran precompilers.

The reverse mode of derivative evaluation is of interest because the gradient of a function can be obtained for approximately the same cost as the original function evaluation. Unfortunately, the reverse mode requires information about a computation to be stored in order that derivatives may be calculated in the reverse fashion. This extra cost can be prohibitive on large problems.

The reverse mode presents another challenge. In a multiprocessing environment, we would like to execute the derivative evaluation in parallel. The forward mode of automatic differentiation can be done trivially in parallel, but it is inefficient for problems with a large number of independent variables (significantly more independent variables

than processors). Indeed, in such cases, the reverse mode run sequentially is superior to the forward mode run in parallel.

Our goal in studying the reverse mode of automatic differentiation has been twofold: (1) to find a solution to the storage problem, and (2) to develop an efficient parallel implementation. This paper is, accordingly, organized as follows. In Section 2, we briefly review the techniques of automated differentiation. In Section 3, we discuss the sequential version of the automatic differentiation package ADOL-C and its management of the storage problem. In Section 4, we compare the results and execution time of a sample problem using ADOL-C and divided differencing to produce gradients. In Section 5, we present our parallel implementation of reverse mode of derivative evaluation. In Section 6, we examine the efficiency of our parallel implementation. Finally, in Section 7, we summarize the benefits of the ADOL-C implementation.

2 Preliminaries

Automatic differentiation is the technique of applying differentiation rules to a computation, producing derivatives. Computational arithmetic is performed by executing a sequence of assignments, unary and binary operations, and univariate functions. The result is that a computation can be expressed as a composite function. For example, the sequence of instructions

```
x = a*b;
y = b*b;
z = x+y;
```

computes the function $z = b^2 + ab$. By applying differentiation to arithmetic operations and univariate functions and then applying the chain rule to each operation, we can differentiate the composite function. Applying the standard differentiation rules with respect to b in the above example, we get the following result.

$$\begin{aligned}
\frac{\partial x}{\partial b} &= \frac{\partial b}{\partial b}a + \frac{\partial a}{\partial b}b = a \\
\frac{\partial y}{\partial b} &= \frac{\partial b}{\partial b}b + \frac{\partial b}{\partial b}b = 2b \\
\frac{\partial z}{\partial b} &= \frac{\partial x}{\partial b} + \frac{\partial y}{\partial b}b = 2b + a
\end{aligned}$$

As the example illustrates, the chain rule can be applied in a *forward* mode during a sequence of computations. The chain rule may also be applied in a *reverse* fashion.

The reverse mode of derivative evaluation is essentially done by back substitution. One dependent variable is nominated for evaluation. The derivative values are then calculated in terms of previously calculated results. For example, if we nominate z as

the dependent variable in the example calculation, then we have the following:

$$\begin{aligned}
\frac{\partial z}{\partial z} &= 1 \\
\frac{\partial z}{\partial x} &= \frac{\partial z}{\partial z}\frac{\partial z}{\partial x} \\
\frac{\partial z}{\partial y} &= \frac{\partial z}{\partial z}\frac{\partial z}{\partial y} \\
\frac{\partial z}{\partial a} &= \frac{\partial z}{\partial x}\frac{\partial x}{\partial a} + \frac{\partial z}{\partial y}\frac{\partial y}{\partial a} \\
\frac{\partial z}{\partial b} &= \frac{\partial z}{\partial x}\frac{\partial x}{\partial b} + \frac{\partial z}{\partial y}\frac{\partial y}{\partial b}
\end{aligned}$$

If we now let $\bar{x}$ represent the previously calculated value of $\frac{\partial z}{\partial x}$, the sequence of equations becomes

$$\begin{aligned}
\bar{z} &= 1 \\
\bar{x} &= \bar{z}\frac{\partial z}{\partial x} = \bar{z} \\
\bar{y} &= \bar{z}\frac{\partial z}{\partial y} = \bar{z} \\
\bar{a} &= \bar{x}\frac{\partial x}{\partial a} + \bar{y}\frac{\partial y}{\partial a} = \bar{x}b \\
\bar{b} &= \bar{x}\frac{\partial x}{\partial b} + \bar{y}\frac{\partial y}{\partial b} = \bar{x}a + \bar{y}2b.
\end{aligned}$$

An evaluation of this sequence produces $\bar{a} = \frac{\partial z}{\partial a} = b$ and $\bar{b} = \frac{\partial z}{\partial b} = a + 2b$.

For each binary operation of the form $x = y \otimes z$, a simple set of rules governs the adjoint quantities $\bar{x}$, $\bar{y}$, and $\bar{z}$. For example, if $x = y * z$, then the sequence

$$\begin{aligned}
\bar{y} &+= \bar{x} * z \\
\bar{z} &+= \bar{x} * y
\end{aligned}$$

would be executed during the reverse evaluation. This set of rules allows the reverse evaluation to proceed at a fixed constant of the time required to compute the actual function.

Using the previously described reverse mode, we can calculate the derivative of the dependent variable with respect to every variable in the computation. For scalar functions of the form $y = F(x_1, x_2, ..., x_n)$ the reverse mode of automatic differentiation is $O(n)$ times faster than either the forward mode or divided differencing. See Griewank [Grie89] or Rall [Rall81] for a more detailed overview of the techniques for automatic differentiation.

We use a standard set of notation when dealing with the automatic differentiation of computer arithmetic. The essential components of any computation are a set of independent variables $\{x_1, ..., x_n\}$, a set of dependent variables $\{y_1, ..., y_m\}$, and a function $\mathcal{F} : \{x_1, ..., x_n\} \Rightarrow \{y_1, ..., y_m\}$. The forward mode of automatic differentiation computes the derivatives of all dependent variables with respect to one independent variable. The reverse mode of automatic differentiation calculates the derivatives of one dependent variable with respect to all independent variables. Each pass of automatic differentiation requires approximately the same amount of time to execute as the original function. It follows that, for $n < m$, the forward mode is superior. For $m < n$, the reverse mode is superior. This paper will primarily consider the reverse mode of automatic differentiation.

3 The ADOL-C Package

Rall [Rall84] used *operator overloading* in PASCAL-SC to implement the forward mode of automatic differentiation. Rall's package demonstrates that automatic differentiation can be done with relative ease in a language that supports operator overloading. We have developed an automatic differentiation package, ADOL-C, which uses operator overloading extensively. This section discusses ADOL-C and its facilities for producing gradients efficiently in the reverse mode. ADOL-C also has the ability to produce a truncated Taylor series in the forward mode, as well as higher derivatives.

The ADOL-C package defines a new class in C++ to provide automatic differentiation facilities. The new class *vfloat* is a class of virtual floating-point numbers. The following standard operations are defined for the class vfloat.

- The assignment operator $=$.
- The unary operators $+, -$.
- The binary operators $+, -, *, /$.
- The standard C operators $+=, -=, *=, /=$.
- The trigonometric functions sin, cos, tan, asin, acos, atan.
- The C functions exp, log, log10, sqrt, pow.

These operations are overloaded specifically for the class vfloat.

The elements of the class vfloat may be used as any floating-point number or double variable would be used. For example, the program segment

```
vfloat x,y,z;
y=z=5;
x=y+z;
```

produces the result $double(x) = 10.0$. The elements of the class vfloat emulate standard floating-point arithmetic in all respects except one. The operations executed on elements of the class vfloat may optionally be traced. The trace of each operation in a computation provides enough information to calculate first or higher derivatives of the current evaluation.

Elements of the class vfloat are not simple floating-point numbers. The only data item associated with each vfloat variable is an integer location. This location is used to change the corresponding element of a storage array during arithmetic operations. In essence, each vfloat variable is an indirect address of a floating-point number. A vfloat's location is used to accurately trace its use in a computation throughout its lifetime.

When a vfloat element is constructed, it is assigned a unique location among all currently live vfloats. Each vfloat retains its unique location throughout its lifetime. At destruction, a vfloat's location is marked as unused. This unused location may then be reused at some later construction. Our live variable analysis allows the size of the storage array to remain manageable.

The facilities provided to trace a computation are relatively simple. By enclosing the execution of a computation by the function call $trace(ON)$ and $trace(OFF)$, all numeric and data operations of the computation are traced. For example, the program segment

```
vfloat x,y,z;
y=10;
z=12;
trace(ON);
x=y+z;
x+=z;
trace(OFF);
```

traces the execution of the function $x = y + 2z$. The execution occurs sequentially, and thus the trace of the computation is stored sequentially. A large buffer holds the trace of the computation. If this buffer becomes full, then it is written to a file. Tracing the computation in this purely sequential manner keeps the file (and possibly virtual memory) access to a minimum. This is the key to handling the storage problem efficiently.

After creating a trace of the computation, our ADOL-C package allows questions to be asked about the computation. By calling the function *gradient_pass*, the derivatives of specified dependent variable with respect to all of the independent variables in the computation are calculated. The function *gradient_pass* uses the reverse mode of automatic differentiation; thus each variable in the computation requires an associated adjoint quantity. By calling the function *set_memory(k)*, an adjoint array of degree k is associated with each vfloat variable in the computation. For calculating gradients, each vfloat requires only one adjoint quantity. Thus the sequence

```
set_memory(0);
gradient_pass(x);
```

is sufficient to calculate the variables $\frac{\partial x}{\partial y}$ and $\frac{\partial x}{\partial z}$. To access the partial derivatives of the dependent variable with respect to a given independent or intermediate variable, we use the member function *adj*(). In conjunction with the previous examples, the sequence

```
double dxdy,dxdz;
dxdy = y.adj();
dxdz = z.adj();
```

places the values 1.0 and 2.0 in *dxdy* and *dxdz*, respectively.

ADOL-C employs rather simple constructs to produce gradients. These same constructs are used to produce higher derivatives and a truncated Taylor series.

4 A Comparison with Divided Differencing

Recall that the derivative of a function $f'(x)$ is defined by

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$

We can approximate the gradient of a function at a point by selecting a small enough h in the definition of derivative. This technique requires $n+1$ evaluations of the original n variable function.

The technique of approximating the gradient of a function by using divided differencing is widely used. We compare the time required to compute the gradient of the Helmholtz energy function by divided differencing with an implementation using the ADOL-C package. The Helmholtz energy function is examined with respect to various differentiation techniques by Griewank [Grie89]. We give a brief description of the function and compare the resulting execution times of both techniques.

4.1 The Helmholtz Energy Function

As Griewank [Grie89] mentions, the Helmholtz energy at the absolute temperature T of a mixed fluid in a unit volume is

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

R is the universal gas constant, and

$$0 \leq x, b \in \mathbf{R}^n, A = A^T \in \mathbf{R}^{n \times n}.$$

This function and its gradient are used extensively in the simulation of oil reservoirs. Differentiating this function is nontrivial. For this reason it was chosen as our test problem.

4.2 Timing Results

In our comparison, we wrote two C++ implementations for calculating the Helmholtz energy function and its gradient. The first implementation used double-precision arithmetic to calculate the function and divided differencing to calculate the gradient. The second implementation used the package ADOL-C. A combination of vfloat and double-precision arithmetic was used to compute the function; all numeric vfloat values were calculated by using double-precision arithmetic. The gradient of the function was then calculated by using the reverse mode of automatic differentiation. The results of our timing comparisons on a Sun 3 workstation are shown in Fig. 1.

In all comparisons the values from divided differencing and our analytically calculated derivatives differed by at most 0.001. These differences can be attributed largely to the truncation error caused by divided differencing. The ADOL-C implementation used 25 megabytes of file storage on the Helmholtz energy function with 1000 independent variables. This evaluation took approximately 10 minutes to complete. The divided differencing implementation required over 4 hours to complete on the same problem.

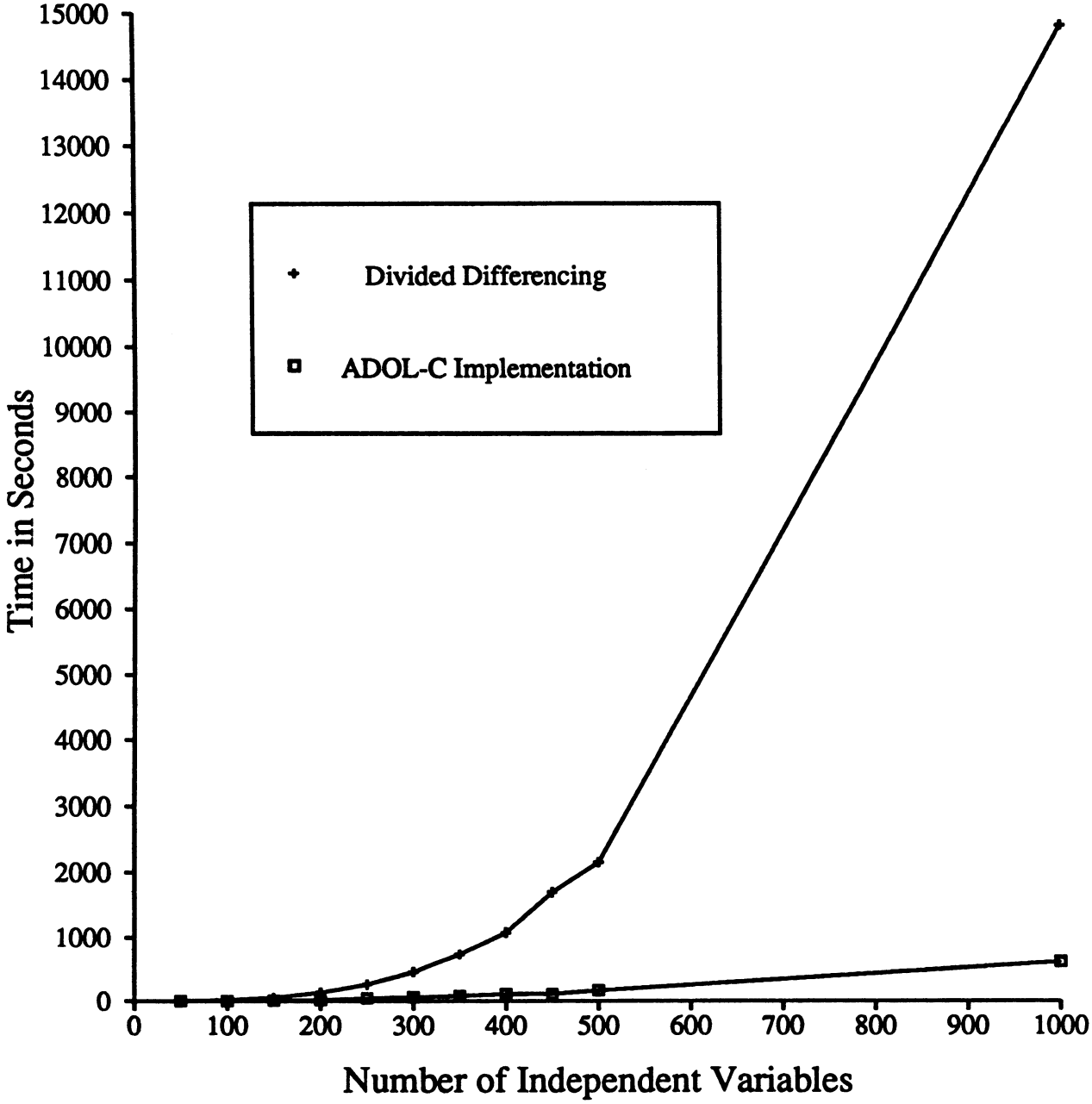

Figure 1: ADO-C vs. Divided Differencing

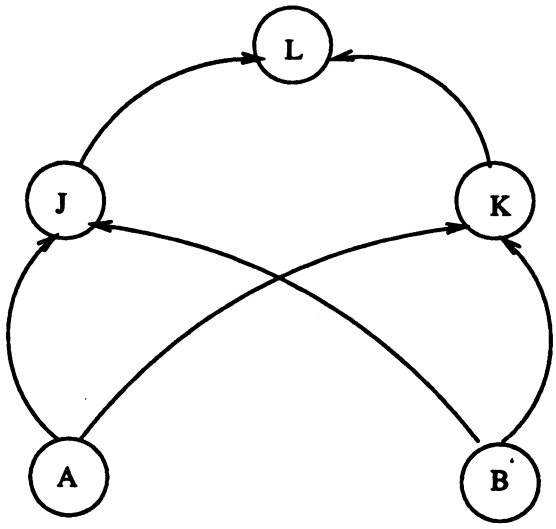

Figure 2: The Dependency Graph

locally or be addressed in a shared fashion. In order to ensure the consistency of a section of shared memory, locks are used to surround critical sections of code. The extensive use of locking mechanisms can be a drain on the performance of any parallel program; thus we minimized the use of locking mechanisms. We saw the evaluation queue to be the main bottleneck of our implementation; we therefore chose to use a multiple-layered approach to simulate a single evaluation queue. Our approach is as follows.

- Each processor uses a local evaluation queue. This queue is accessed locally and does not need to be locked. If the local queue has an element, it is evaluated first. This queue is of fixed length.
- Each pair of two processors has a local/shared queue. If a processor's local queue is full, it places elements ready for evaluation on its local/shared queue. When a processor's local queue is empty, it first searches its local/shared queue for the next element to be evaluated. This queue is shared and accessed via locking mechanisms.
- If both a processor's local and local/shared queues are empty, then the local/shared queues of the remaining processors are searched in a round robin fashion.

This scheme is illustrated in Figure 3.

5 A Parallel Implementation of the Reverse Mode

It would be ideal if the advantage in efficiency of the reverse mode over the forward mode of automatic differentiation translated exactly to a multiprocessor environment. Calculating the gradient of a function in parallel using the *forward mode* of automatic differentiation can be efficiently implemented. One simply assigns each processor the task of calculating the partial derivative of all dependent and intermediate variables with respect to one independent variable. In this manner the main problem is broken into p smaller problems, one for each of the p processors. Such problems are said to have *coarse granularity.*

It is not immediately obvious how to break the reverse evaluation of the gradient into p subproblems. The difficulty rests with the wealth of dependency information embedded in any trace of a computation. Using this dependency information to evaluate a function concurrently produces subproblems on the order of a few instructions; however, the *fine granularity* of this technique can create communication problems.

A trace of a computation indirectly stores all of the dependency information we need. Unfortunately, a variable may depend on values computed much earlier in the computation. Thus, it is impractical simply to trace the computation and then search for concurrency on the reverse sweep. Our implementation stores a *dependency graph* instead of simply tracing the computation. For example, the execution of the instructions

```
J=A+B;
K=A*B;
L=J+K;
```

produces a dependency graph with five nodes. The resulting dependency graph is shown in Figure 2.

The dependency graph given by any computation becomes a *directed acyclic graph.* The reverse mode of automatic differentiation is essentially a downward traversal of the dependency graph starting at the node corresponding to the dependent variable. It follows that, given any dependency graph $\mathcal{G}$ of depth d, the reverse mode of derivative evaluation requires at least d time steps to complete.

Our parallel implementation of the reverse mode traverses the dependency graph, evaluating partial derivatives at each node. The embedded dependency information is inverted during the reverse sweep. The node that corresponds to the dependent variable is seeded with the value 1 and placed on an evaluation queue. Each node placed on the evaluation queue will eventually be visited, and its derivative information propagated to the nodes that depend on it. An unevaluated node is placed on the evaluation queue once all of the nodes it depends on have been evaluated. When a processor is available, it accesses the evaluation queue and evaluates the next node. When the evaluation queue is empty, the reverse sweep of derivative evaluation is complete.

Our implementation was done on a Sequent Symmetry configured with 24 processors. The Sequent Symmetry is a shared-memory machine. Memory can either be accessed

6 Parallel Test Results

Our implementation was primarily concerned with the reverse evaluation of derivatives. The ADOL-C package was modified to generate the dependency graph during the original function evaluation. Our modified package contains routines to evaluate the gradient in parallel by using the dependency graph. Several factors influence the performance of our parallel implementation.

- the structure and depth of the dependency graph,
- the size of the given problem, and
- the distribution of elementary operations.

The original function evaluation creates the shape of the dependency graph. No parallel traversal of the dependency graph can execute faster than the depth the dependency graph. Thus the user's function dictates the amount of parallelism available for our package to exploit. The optimal case occurs when a user's function creates a dependency graph G of depth $\log |G|$.

We did several tests of the experimental package on large problems. On the Helmholtz energy function (Section 4), we obtained promising results. Computing the gradient of the Helmholtz energy function with 300 independent variables, we were able to execute it over 11 times faster using 15 processors than using a single processor. We obtained similar results up to 18 processors on the Sequent Symmetry. Figure 4 plots our results with respect to the theoretical linear speedup in the number of processors used.

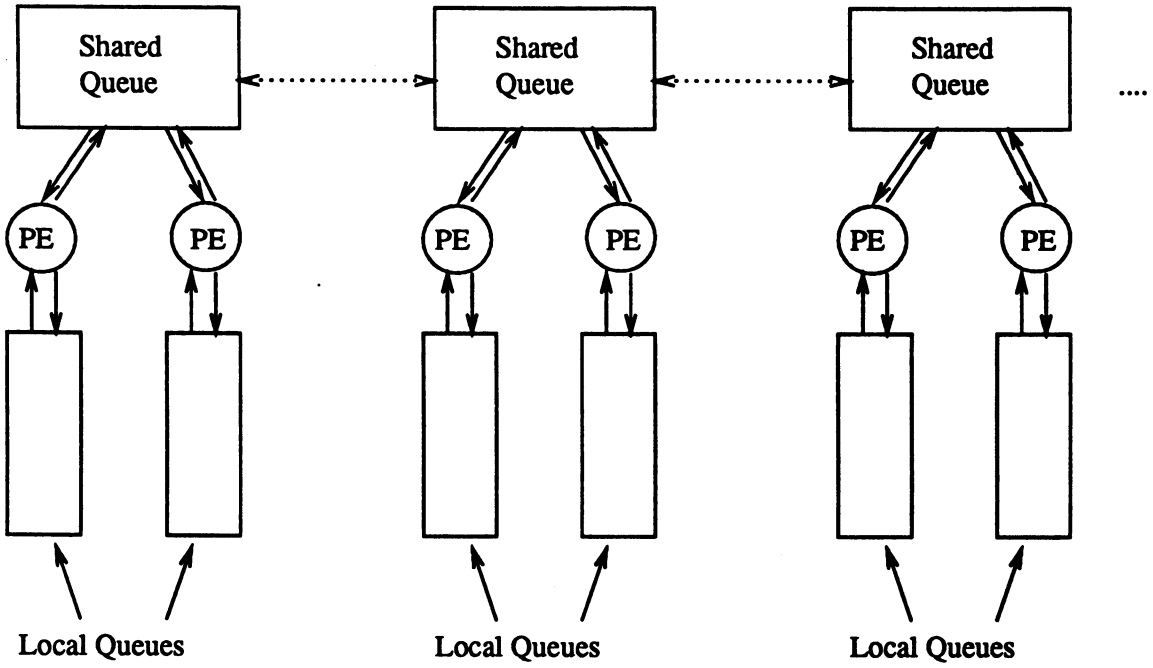

Figure 3: Simulating a Global Evaluation Queue

7 Conclusion

Though the automatic differentiation of computer arithmetic is well understood and practical, it is not widely used. Using divided differencing is perhaps the easiest way of approximating the gradient of a function. Unfortunately, divided differencing is inherently inaccurate and inefficient. We have implemented an automatic differentiation package that is efficient in terms of temporal, spatial, and parallel resources.

ADOL-C provides temporal efficiency by using the reverse mode of automatic differentiation to produce analytical derivatives. It provides spatial efficiency by accessing the trace of the computation in a purely sequential mode. Finally, our experimental parallel version of ADOL-C uses parallel resources efficiently to compute gradients. We believe that this package will be of great use in applications that require more accuracy than approximate methods provide.

References

[BS83] W. Baur and V. Strassen (1983). "The Complexity of Partial Derivatives," Theoretical Computer Science, Vol. 22, pp. 317-330.

[Beda59] L. M. Beda et al. (1959). "Programs for Automatic Differentiation for the Machine BESM," Inst. Precise Mechanics and Computation Techniques, Academy of Science, Moscow.

[Grie89] A. Griewank (1989). On Automatic Differentiation," in Mathematical Programming: Recent Developments and Applications, ed. M. Iri and K. Tanabe, KTK Scientific/Kluwer Academic Publishers, Amsterdam.

[Hill85] K. E. Hillstrom (1985). "Users Guide for JAKEF," Technical Memorandum ANL/MCS-TM-16, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois.

[Hor88] J. E. Horwedel, B. A. Worley, E. M. Oblow, and F. G. Pin (1988) "GRESS Version 0.0 Users Manual," ORNL/TM 10835, Oak Ridge National Laboratory, Oak Ridge, Tennessee.

[IK87] M. Iri and K. Kubota (1987). "Methods of Fast Automatic Differentiation and Applications," Research memorandum RMI 87-0, Department of Mathematical Engineering and Instrumentation Physics, Faculty of Engineering, University of Tokyo.

[Rall81] L. B. Rall (1981). "Automatic Differentiation: Techniques and Applications," in Lecture Notes in Computer Science, No. 120, Springer-Verlag, Berlin, 1981.

[Rall84] L. B. Rall (1984) "Differentiation in PASCAL-SC: Type GRADIENT," ACM TOMS, Vol. 10, pp. 161–184.

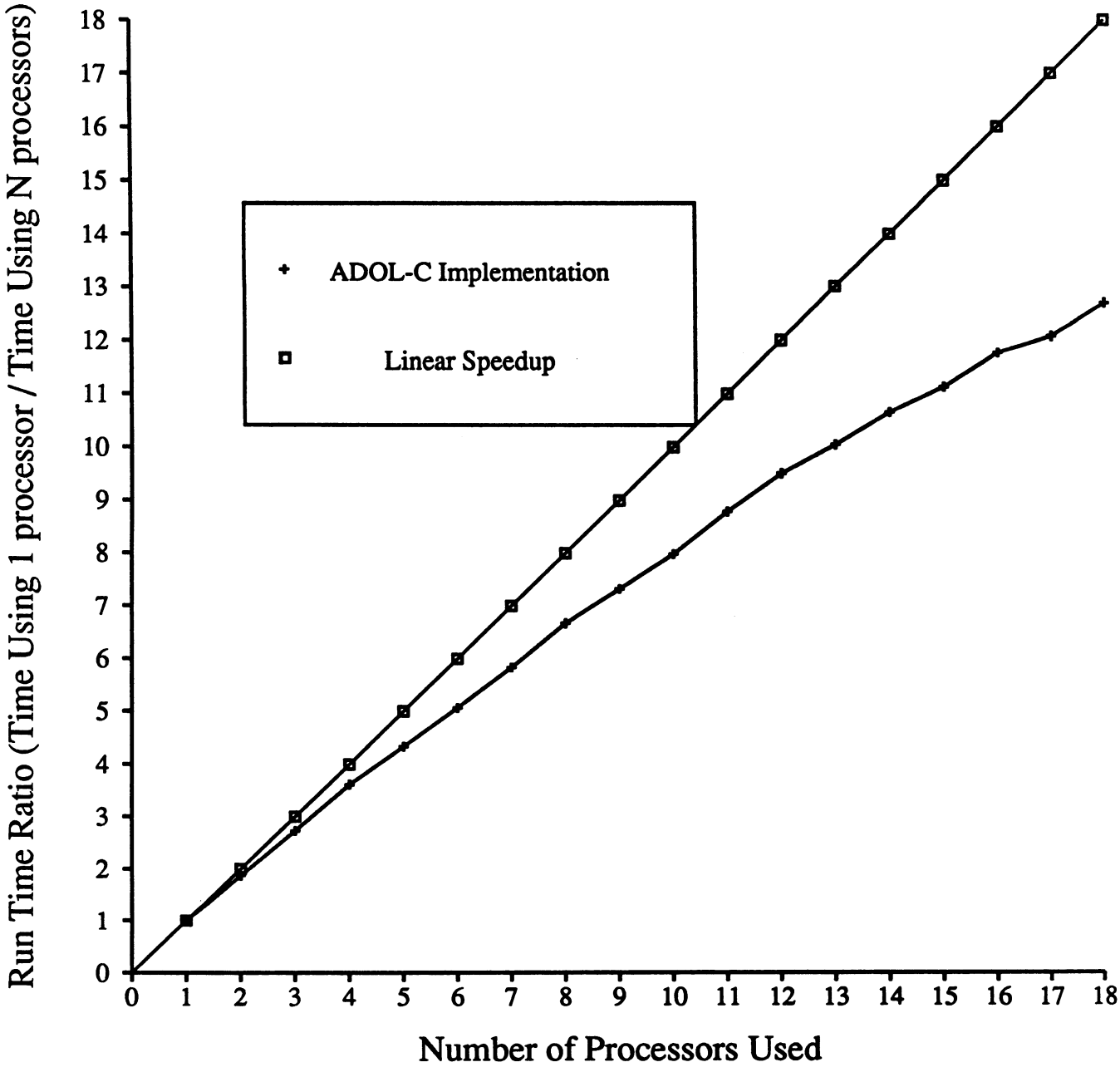

Figure 4: Parallel Test Results vs. Linear Speedup

[Spe80] B. Speelpenning (1980). "Compiling Fast Partial Derivatives of Functions Given by Algorithms," Ph.D. Dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois.

[Wen64] R. E. Wengert (1964). "A Simple Automatic Derivative Evaluation Program," Comm. ACM, Vol. 7, pp. 463–464.