AIAA 94–4261

PARALLEL CALCULATION OF SENSITIVITY DERIVATIVES FOR AIRCRAFT DESIGN USING AUTOMATIC DIFFERENTIATION

C. H. Bischof
MCS Division, Argonne National Laboratory
Argonne, Illinois 60439

L. L. Green, K. J. Haigler
MDOB/FMAD, NASA Langley Research Center
Hampton, Virginia 23681–0001

T. L. Knauff, Jr.
MCS Division, Argonne National Laboratory
Argonne, Illinois 60439

Presented at the
5th AIAA/NASA/USAF/ISSMO Symposium on
Multidisciplinary Analysis and Optimization Conference

Panama City, Florida
September 7–9, 1994

PARALLEL CALCULATION OF SENSITIVITY DERIVATIVES FOR AIRCRAFT DESIGN USING AUTOMATIC DIFFERENTIATION

C. H. Bischof*, L. L. Green,†, K. J. Haigler,‡, and T. L. Knauff, Jr.¶

*, ¶ MCS Division, Argonne National Laboratory
Argonne, Illinois 60439

†, ‡ MDOB/FMAD, NASA Langley Research Center
Hampton, Virginia 23681–0001

Abstract

Sensitivity derivative (SD) calculation via automatic differentiation (AD) typical of that required for the aerodynamic design of a transport-type aircraft is considered. Two ways of computing SD via code generated by the ADIFOR automatic differentiation tool are compared for efficiency and applicability to problems involving large numbers of design variables. A vector implementation on a Cray Y-MP computer is compared with a coarse-grained parallel implementation on an IBM SP1 computer, employing a Fortran M wrapper. The SD are computed for a swept transport wing in turbulent, transonic flow; the number of geometric design variables varies from 1 to 60 with coupling between a wing grid generation program and a state-of-the-art, 3–D computational fluid dynamics program, both augmented for derivative computation via AD. For a small number of design variables, the Cray Y-MP implementation is much faster. As the number of design variables grows, however, the IBM SP1 becomes an attractive alternative in terms of compute speed, job turnaround time, and total memory available for solutions with large numbers of design variables. The coarse-grained parallel implementation also can be moved easily to a network of workstations.

Nomenclature

Symbols

C_D	Wing drag coefficient
C_L	Wing lift coefficient
C_M	Wing pitching moment coefficient
F	Function viewed as a mapping
F_j	Objective functions
G	Generic geometric variable
J	Jacobian
S	Seed matrix
SD_{ij}	Sensitivity derivative matrix
X	Generic grid coordinate
X_i	Design variables
cmax	Airfoil section maximum camber
e_i	i-th canonical unit vector
f, g, h	Generic functions
r, s, t, u	Generic variables
t/c	Airfoil section thickness-to-chord ratio
xcmax	Airfoil section location of maximum camber
x, y, z	grid coordinates

* Computer Scientist
† Research Scientist, Senior Member AIAA
‡ Aerospace Technologist
¶ Student

Acronyms

AD	Automatic differentiation
.AD	AD generated code
ADIFOR	AD of Fortran
CFD	Computational fluid dynamics
DD	Divided differences
MDO	Multidisciplinary design optimization
MIMD	Multiple instruction multiple data
MW	Megawords of computer memory
NDV	Number of design variables
NP	Number of processors
SD	Sensitivity derivative(s)
TLNS3D	3-D Navier-Stokes CFD program
WINGDES	Linear aerodyamics program
WTCO	Wing grid-generation program

Introduction

The realistic multidisciplinary design optimization (MDO) of advanced aircraft using state-of-the-art computers is a challenging problem from both a physical modeling and a computer science point of view. To produce an efficient aircraft design, many trade-offs are necessary among the various physical design variables. Similarly, to produce an efficient design scheme, many trade-offs are necessary among the various MDO implementation options. In this paper, we examine the effects of vectorization and coarse-grained parallelization on a sensitivity derivative (SD) calculation using a representative sample taken from a transonic transport design problem.

A typical MDO problem involves the minimization or maximization of one or more objective functions F_j by changing many design variables X_i, subject to a number of constraints. The design variables are usually inputs to one or more computer programs that must be executed to simulate the complex phenomenon characterized by the objective functions. Many MDO schemes require the repeated calculation of a matrix of terms, called SD's,

$$SD_{ij} = \frac{dF_j}{dX_i},$$

which describe how the objective functions change with respect to the design variables.

For realistic design problems, the requirements for the MDO scheme may change as the design proceeds. During the conceptual design, for example, low-fidelity physics modeling and low-accuracy SD calculations may be acceptable or desirable, so that the effects of a large number of design variables can be examined over a broad domain. Later, as the design approaches an optimum point, fewer variables may be permitted to change, but the calculations may require extremely high degrees of accuracy. In addition, as the design proceeds, new design variables or objective functions may be identified and others eliminated. Thus, the SD calculation scheme should embody a high degree of flexibility and predictable levels of accuracy. Furthermore, the SD calculation scheme must be efficient, because this operation represents a large portion of the total optimization time. Several recent research efforts[1–5] have demonstrated the potential for using automatic differentiation (AD) to compute the SD required for MDO from a variety of codes. The AD approach meets the requirements for versatility, accuracy, and speed in an MDO scheme.

Other methods for computing the SD include divided differences (DD), hand coding of derivatives from analytic relationships, and symbolic manipulators. Each of these methods, however, has one or more severe drawbacks with respect to the stated requirements for flexibility, accuracy, and speed. Moreover, these methods become unwieldy as the complexity and resolution of the physical simulation or the number of design variables

increases. However, as demonstrated in this paper, the use of AD provides the basis for a scalable SD technology suitable for today's parallel computers.

The SD calculation relevant to the aerodynamic design of a transport-type aircraft is considered in this paper. An important consideration that must be made early in the design process is the number and type of design variables and constraints that will be allowed, because this determines the domain in which the design can occur. For example, an extremely simple wing can be described by just a few parameters: the thickness and chord specified at the root and tip sections, the span, and the leading-edge sweep angle. Similarly, a fuselage with circular cross sections can be described by the three coefficients of an elliptic or parabolic radius distribution along the fuselage axis. Such simple wings and fuselages may be useful in preliminary design studies, but they lack much of the detail of commercially available configurations. A more realistic wing-body configuration can easily require about 200 design variables: 8 variables at each of 15 wing sections, plus 3 or 4 variables at each of 20 fuselage cross sections. Additional geometric design variables (perhaps a total of 1000) can be required to describe the location, shape, three-axis orientation, and the size of the nacelles, pylons, tails, canards, and winglets. Note that the effects of any of these aircraft components on the aerodynamic properties, such as the lift and drag, are realized within a computational fluid dynamics (CFD) code by the flow solution dependence upon the grid (x,y,z) coordinates, and thus any of these components may be treated in the same way within the CFD code itself. However, as the increase in the number of design variables and constraints allows for a more refined geometric configuration, the optimization problem also becomes more difficult and the storage and compute time requirements increase. Although thousands of variables may ultimately be used in future design problems, current practice (except for some inverse methods) usually restricts the design space to just a few; this work employs a design domain of up to 60 variables and examines the storage and compute time issues relevant to the SD calculation via AD on vector and parallel computers.

AD and Derivative Strip-Mining

Automatic differentiation[6–8] is a chain-rule-based technique for evaluating the derivatives of functions defined by computer programs. Automatic differentiation techniques rely on the fact that every function, regardless of how complicated, is executed on a computer as a (potentially long) sequence of elementary operations such as addition, multiplication, and elementary functions (e.g., sine and cosine). By applying the chain rule

$$\frac{\mathrm{df}[\mathrm{g}(\mathrm{t}),\mathrm{h}(\mathrm{t})]}{\mathrm{dt}} = \frac{\partial \mathrm{f}(\mathrm{s},\mathrm{r})}{\partial \mathrm{s}} \times \frac{\mathrm{dg}(\mathrm{t})}{\mathrm{dt}} + \frac{\partial \mathrm{f}(\mathrm{s},\mathrm{r})}{\partial \mathrm{r}} \times \frac{\mathrm{dh}(\mathrm{t})}{\mathrm{dt}}$$

repeatedly to the composition of those elementary operations, the derivative information of the function f can be computed exactly (to machine precision) in a completely mechanical fashion.[7–9]

The ADIFOR (Automatic DIfferentiation of FORtran)[9–12] tool has been developed jointly by the Mathematics and Computer Sciences Division at Argonne National Laboratory and the Center for Research on Parallel Computation at Rice University. To apply ADIFOR to a given Fortran 77 code, the user need only specify those variables that correspond to independent and dependent variables with respect to differentiation; ADIFOR then determines which variables require associated derivative computations, formulates the derivative expressions, and generates new Fortran 77 code for the computation of both the original simulation and the associated derivatives. For the remainder of this paper, an ADIFOR-generated code version will be designated with a .AD suffix.

The ADIFOR program provides a flexible interface that allows for the computation of arbitrary linear combinations of the columns of the Jacobian matrix. That is, if a Fortran code is considered as a mapping F: x $\rightarrow$ y, with associated Jacobian

$$\mathrm{J} = \frac{\mathrm{dF}}{\mathrm{dx}},$$

ADIFOR generates code that allows one to compute J×S, where the so-called seed matrix S is initialized at run time. The user is hence able to choose, at run time, to either compute all sensitivities (i.e., the full Jacobian matrix J), any subset of columns of J, an arbitrary Jacobian-vector product, or a compressed Jacobian to exploit sparsity.[13]

Automatic differentiation is a powerful technique for obtaining accurate and efficient sensitivities of design codes, but can be demanding in terms of resources. If we are interested in the computation of number-of-design-variable (NDV) sensitivities, then the ADIFOR-generated code replaces each statement involving derivative-related variables with possibly some assignment statements and a loop of length NDV. Hence, as a rule of thumb, the ADIFOR-generated code for the computation of NDV sensitivities can require up to NDV times the memory of the original simulation. The increase in run time is more difficult to predict, because ADIFOR need not augment every statement in the original code; however, experience has shown that the run time of the derivative code can be expected generally to increase by a factor of somewhat less than NDV.

Because of memory and computational time limitations, a single run of the sensitivity-enhanced version of a simulation code may compute fewer than the number of sensitivities desired overall. These limitations greatly slow down the design/sensitivity analysis cycle time with a “human in the loop”, because both the designer and a complete set of design information must be available at the same time to permit intelligent processing of the data and to allow the cycle to proceed. To alleviate this problem, parallel processing is employed to quickly and temporarily increase the resources available for sensitivity analysis. That is, even though the simulation code is a serial code, coarse-grained parallelism is used in the computation of derivatives.

To illustrate, assume that we wish to compute the Jacobian

$$\frac{\mathrm{dF}}{\mathrm{dX}_{i=1,17}}$$

of a function depending on 17 parameters. The ADIFOR-generated code allows us to compute any subset of the entries of this gradient through proper choice of the seed matrix. Hence, we can decrease turnaround time by spawning several copies of the ADIFOR-generated derivative code to run simultaneously on multiple processors, as shown in Figure 1. Here, e_i denotes the i-th canonical unit vector. Hence, the first replication of the ADIFOR-generated code will compute the first six entries of the gradient, the second one will compute the next six, and the third one will compute the remaining five.

This strip-mining approach is simple; it can be compared to running several simulations in parallel for DD approximations of derivatives, decreasing wall-clock time with minimal human effort. It can easily be mapped onto any collection of compute nodes, from a MIMD-type parallel computer to a heterogeneous workstation network.

To implement this approach, we employed the Fortran M system.[14,15] Fortran M is a small set of extensions to Fortran primarily geared toward coordination of relatively large-grained parallel applications. Loosely speaking, Fortran M can be considered an object-oriented version of Fortran, because Fortran M tasks exchange information by sending and receiving data across channels. Fortran M provides constructs for creating tasks and channels, for sending messages on channels, for mapping tasks and data to processors, and so on. Fortran M was chosen for two reasons. First, only minimal modifications were required to the original serial Fortran 77 code generated by ADIFOR because Fortran 77 is a subset of Fortran M. Secondly, Fortran M supports parallel execution on heterogeneous workstation networks, the most ubiquitous parallel-computing resource.

Given a serial code for the computation of sensitivities, a Fortran M master process was created which (given the number of parallel nodes NP and the number of sensitivities NDV to compute) spawns *ceiling*[‡](NDV/NP) clones of the se-

‡ The ceiling function is an integer division that rounds up instead of down.

rial derivative code and communicates to each node the particular subset of sensitivities it must compute. Each slave process then reads the appropriate subset of the total seed matrix from the disk, computes its share of the sensitivities, and sends the result back to the master. This computational harness is simple and adaptable to any other ADIFOR-generated derivative code with minimal modifications.

Description of Computer Codes

The WTCO Wing Grid-Generation Code

The WTCO code is a batch-mode, algebraic, transfinite interpolation wing grid-generation program. The code can generate grids around wings of at least two airfoil sections that are described by ordinates or NACA four-digit airfoils. In this case, the usual NACA four-digit airfoil family has been expanded by allowing the maximum camber (cmax), the location of maximum camber (xcmax), and the thickness ratio (t/c) to be specified as real number inputs, rather than deduced from an integer designation. A true NACA 2412 has a cmax = 0.02, xcmax = 0.4, and t/c = 0.12; the WTCO program has been modified to allow for small perturbations to this shape, such as cmax = 0.02002, xcmax = 0.4004, and t/c = 0.12012. These modifications were necessary to perform the DD runs used to verify the SD calculations by AD.[3] The user specifies spacing constants at several points on the grid boundaries, the type of boundary surface, the interior interpolation, and the amount of smoothing required. The WTCO code was chosen for use in this study for the following reasons. The code has been used reliably with the flow solver on many previous occasions, is noniterative and does not include coding that is extraneous to the grid generation (such as graphic representation). The source code necessary for AD application was readily available. In addition, the SD calculations for up to four geometric design variables had previously been calculated and verified.[3] The grid coordinates were differentiated using ADIFOR with respect to four section parameters: maximum camber, location of maximum camber, thickness to chord ratio, and the section twist angle. The total number of design variables can be changed by changing the number of input sections that describe the wing; in this study, up to 15 sections with a total of 60 design variables were used.

The TLNS3D Aerodynamic Analysis Code

The TLNS3D code[16] is a high-fidelity aerodynamic computer program that solves the time-dependent three—dimensional (3–D) thin-layer Navier-Stokes equations with a finite-volume formulation. The code employs grid sequencing, multigrid, and local time stepping to accelerate convergence and efficiently obtain steady-state high Reynolds number turbulent flow solutions. When temporally converged to a steady-state solution, the method is globally second-order accurate. The TLNS3D code is a central-difference code that employs second-order central differences for all spatial derivatives and employs a blending of scalar second-difference and fourth-difference artificial dissipation to maintain numerical stability. The solution is advanced explicitly in time with a five-stage Runge-Kutta time-marching algorithm. The code includes the Baldwin-Lomax (B-L) turbulence model. This code has been used successfully in a number of applications across the flight-speed range from low subsonic to hypersonic and for a number of flight vehicle types.

The wing lift, drag, and pitching moment coefficients (C_L, C_D, and C_M) are differentiated with respect to the grid coordinates with ADIFOR. Coupling between WTCO.AD and TLNS3D.AD is accomplished by passing as files both the grid and the grid SD matrix with respect to the above geometric parameters. An application of the chain rule within TLNS3D.AD is used to calculate an aerodynamic SD with respect to a geometric design variable (for example, the wing lift coefficient) as

$$\frac{dC_L}{dG} = \frac{dC_L}{dX} \cdot \frac{dX}{dG}$$

In this equation, G is a generic geometric variable, $\frac{dC_L}{dX}$ is a new code seg-

ment produced by an ADIFOR application to TLNS3D, and $\frac{dX}{dG}$ is the grid SD array that is input to TLNS3D.AD and used to initialize the seed matrix. As discussed in Refs. 1 and 2, both the function and its derivatives are computed in an iterative fashion with the multigrid algorithm, although other additional current research is under way to develop a more efficient paradigm for AD application to iterative algorithms.

Note that the TLNS3D.AD cases were not run to convergence; such runs may require several hundred to perhaps 1000 fine-grid multigrid iterations. Instead, the code was run only 50 coarse-grid cycles and 50 fine–grid cycles for the timing studies shown. Previous results[2] for such cases have indicated that both the function (i.e., the C_L, C_D, and C_M) and the derivatives converge monotonically and that errors in the forces and moments with such short multigrid runs are usually between 10% and 50%. The runs are, therefore, representative of those that will be made during the conceptual or preliminary design phases. Even runs as short as 10 cycles on the fine grid can be used to obtain meaningful timing comparisons, if care is taken to exclude nonscalable times such as the input/output required at the beginning and end of the program.

The TLNS3D.AD results in Ref. 3 used a 2–section input (root and tip) in WTCO.AD, whereas those reported here used up to a 15–section input in WTCO.AD (root, tip, and linearly interpolated intermediate sections). Although the 15–section input described the same physical wing as the 2-section input, numerical differences in the grid and in the grid sensitivity matrix that were passed into TLNS3D.AD precluded direct comparison between derivatives of the wing lift, drag, and pitching moments with respect to root and tip geometric variables. The implication of this effect for a realistic design problem is that the maximum resolution expected in a simulation should be built into the design effort from the start and used throughout the design process, so that consistent results can be obtained from the simulation codes at different points during the design effort.

The WINGDES Wing Design Code

Sensitivity derivatives were also computed for WINGDES,[17] a low-fidelity (linear) vortex lattice aerodynamic method that calculates pressure, lift, and drag for wings in subsonic and supersonic aerodynamic flows. In addition, for a given planform, WINGDES contains an inner design loop that optimizes the wing camber distribution. The WINGDES method also employs nonlinear corrections to the lift and drag to account for attainable leading—edge thrust. Unger and Hall[5] report on their experiences in applying ADIFOR to the WINGDES code, which was provided to the authors for use in this parallelization study.

Description of Computer Techniques

Cray Y-MP Vector Processor

The original sequential TLNS3D code is an efficient, highly vectorized code. The initial version of TLNS3D.AD, however, was not nearly as efficient on the Cray Y-MP computer. The Cray compiler Flowtrace and Loopmark options were used to identify the time consuming subroutines and functions of the AD-generated code, as well as “do loops” that did not vectorize as well as the corresponding ones in the original code. The derivative code was then manually post-processed, and additional compiler options were used to restore much of the code vectorization.[2]

Table 1 shows the timing for the fully vectorized (sequential) TLNS3D.AD code for NDV ranging from 1 to 60 using a single processor of the NASA Langley Research Center Cray Y-MP. Note that the time per design variable increases sharply from four to six design variables and then slowly decreases as the number of design variables increases beyond six. This limitation is due to the Cray compiler, which will unroll the innermost loops automatically only up to a length of 5. In the original TLNS3D code, the innermost loops are those that involve the flux and residual computations; such loops are rich in opera-

tions and vectorize efficiently. However, in TLNS3D.AD, the innermost loops are those generated by ADIFOR of length NDV to calculate the gradient objects; in TLNS3D.AD, such loops offer little opportunity for efficient vectorization, in particular for small NDV. However, as the time-per-design variable indicates in Table 1, extremely good vectorization was achieved with the 48– and 60–design-variable cases.

Another issue with SD calculation on the Cray Y-MP is the amount of memory required by TLNS3D.AD. With the current queues and memory configuration of the Cray Y-MP, only about 48 design variables on the 97×25×17 grid can be considered as a routine job submission; for the next finest grid (193×49×33), this number would drop to about six design variables. Although memory exists on the Cray Y-MP to accommodate up to perhaps 80 design variables on the 97×25×17 grid and 10 design variables on the 193x49x33 grid, use of this much computer memory requires special permission and a dedicated user mode. Even the 48–design-variable case shown in Table 1, submitted to the 125MW 3–hr queue, required more than a day of elapsed wall-clock time to obtain the SD results because of heavy machine use. Results from such a job can be run for only about 50 fine-grid cycles in this queue. If the job were run to convergence, it could require at least 18 hours of Cray Y-MP execution time in addition to the time required in the input queue. The 60 design variable case actually fared better in the input and executes queues, because it could only be run in a dedicated user mode (usually overnight, or over a weekend). This case required special permission, but executed reasonably quickly, once permission was granted to submit the job.

Parallel Experiments on IBM SP1

For the WINGDES.AD code, sensitivities were computed of pressure, lift, and drag with respect to 7 wing planform variables, 14 wing thickness variables, and 21 wing camber variables (i.e., 42 design variables overall). For the TLNS3D.AD code, sensitivities were computed of lift, drag, and pitching moments with respect to the twist, maximum camber, location of maximum camber, and thickness for essentially every wing section on a 97×25×17 grid. The goal was to compute sensitivities for a configuration specified by 15 wing sections (i.e., 60 sensitivities overall). A more detailed specification of the wing would be useless, because it could not be resolved further on the chosen grid.

Both codes were run in single precision on the IBM SP1 parallel computer at Argonne National Laboratory. For the purpose of this experiment, this arrangement can be viewed as a collection of IBM RS/6000 workstations. Each machine has a clock speed of 62.5 MHz, 128–MB main memory, and a 32–K data and 32–K instruction cache. All nodes are connected over a high-speed switch.[18]

Serially, one simulation of WINGDES without the computation of any sensitivities took 11.1 seconds. As shown in Table 2, the serial ADIFOR-generated code took 158.2 seconds to compute all 42 sensitivities. As discussed previously, we used strip-mining for the derivative computations. For each parallel run, the average time taken by each slave process and the standard deviation of these times is shown. In comparison, the time taken by the master process was negligible. By using 8 processors, the time for computing 42 sensitivity derivatives was reduced from 158 to 42 seconds with minimal effort. Since the standard deviation times are small relative to the average process times, it is reasonable to expect that all the parallel processes can be completed in about the average time required for one process.

The TLNS3D.AD code posed a different kind of problem. With a 97×25×17 grid, only 12 sensitivities could be computed serially on a node before memory limitations were encountered. However, by cloning several of these processes, the 60 sensitivities for the 15–wing-section configuration were computed in the times shown in Table 3. The times shown do not include the time for running WTCO.AD, which was executed serially to create the grid sensitivities data and save it on

disk. The TLNS3D.AD code was run for 50 cycles on both the fine and the coarse grids. If only one processor were available, then the memory limitation would require the 60-sensitivity job to be run as 5 jobs of 12 sensitivities each (using different portions of the seed matrix) for an estimated run time of 9.5 hours. By employing 15 processors, the same set of sensitivities can be obtained in about one hour, which represents a dramatic increase in turnaround time. Note that it is not possible to obtain perfect speedup, as each slave process not only has to compute its set of derivatives, but also has to re-execute the simulation run itself. Tables 2 and 3 also show that the standard deviation of processor times is low (i.e., all parallel processes essentially finish in the same amount of time).

Comparison of Cray Y-MP and IBM SP1

For instructional purposes we have compared the behavior of the Cray Y-MP and the IBM SP1 in computing various numbers of sensitivities. Table 4 shows the time per iteration for coarse and fine grids as well as the time required per sensitivity on the fine grid using one node of the SP1. These times were obtained from runs of 50 cycles on the coarse grid and 10 cycles on the fine grid. Only the standard compiler options were used to produce the SP1 code for which the results are shown; (i.e., no hand optimization of the SP1 code was performed). This is significant because the Cray can vectorize loops with long vectors very efficiently. (See the entries for 48 or 60 design variables in Table 1.) Such loops do not execute speedily on the RS/6000 architecture because of frequent cache misses.[19] Hence, the run times obtained should not be taken as an indication of the best performance that can be obtained with a code like TLNS3D on the IBM platform.

Figure 2 shows the run time ratio of IBM to Cray time per cycle on the coarse and fine grids for up to twelve design variables. For example, the Cray time for 12 design variables is 45.95 seconds on the fine grid (from Table 1), whereas the IBM time is 128.2 seconds (from Table 4) for twelve design variables; the ratio of IBM to Cray times, shown in Figure 2 is about 2.8 for twelve design variables on the fine grid. In order to interpret Figure 2 correctly, recall that ADIFOR maps an assignment statement from the original TLNS3D code into a sequence of scalar assignments, plus a loop of length NDV. For NDV less than six, the Cray compiler will unroll the short innermost derivative loops and vectorize the second-layer long loops; thus, for one to five design variables the Cray vastly outperforms the IBM. The Cray runs with good vector speed because the innermost derivative loops are unrolled. The superscalar processor on the IBM is kept moderately busy with small loops and otherwise is slowed down through cache misses induced by the large second-layer loops. On the other hand, as the number of design variables grows from 6 to 12 the superscalar chip on the IBM runs more efficiently, whereas the Cray has only relatively short loops to vectorize. The performance of the Cray drops with respect to that of the SP1. Also note that the performance ratios on the fine and the coarse grids are almost identical, as could be expected from a multigrid algorithm.

As the number of design variables continues to grow, the Cray vector performance improves, as evidenced by the entry in Table 1 for 48 and 60 design variables. Several possible scenarios exist for computing 60 sensitivities on the SP1. The single processor memory limits a job execution to a maximum of 12 design variables for the $97 \times 25 \times 17$ grid. Thus, we contrast the execution of 5 different jobs of 12 design variables each run sequentially on a single SP1 processor to the use of multiple SP1 processors to accommodate more design variables in groups of 12 per processor in parallel. In the first case, the Cray performance continually improves relative to the SP1 until all the memory of the Cray is used up. In the second case, the IBM time remains essentially constant as more processors are used in parallel to compute blocks of 12 design variables each, but the Cray time increases with NDV in nonlinear way. It is this second scenario

which is the more interesting of the two from a computational standpoint. From Table 1, it can be seen that the Cray takes 76.92 seconds per fine grid iteration for NDV = 48. The SP1 is assumed to take the same amount of time (128.2 seconds per fine grid iteration, from Table 4) if NDV is 12 or more, since the average standard deviation for multiple processors (Table 3) is low compared to the compute time for the job and only a small portion of the total machine is required. Hence, comparing the derivative strip-mining with four processors on the SP1 with the Cray, the runtime ratio of the SP1 to the Cray would be 128.2/76.92 = 1.67 for NDV=48. Similarly, for NDV = 60, the Cray time is 72.17 seconds, the SP1 time is again assumed to be 128.2 seconds, and the run time ratio is 1.78. These ratios indicate that the SP1 is a viable alternative to the Cray even for moderately large numbers of design variables.

The Cray times do not include that spent in the input queue, which can be a significant portion of the total job time, particularly as time and memory requirements (i.e., the number of design variables and/or the spatial resolution of the problem) increase. At present, no such delays are encountered using the SP1; however, such delays may become a reality as the parallel machine usage increases in the future. At present, the parallel scenario (either SP1, or a network of workstations) provides an attractive alternative to computing sensitivities on the Cray for many cases that can be accommodated by both and provides total machine resources in excess of the Cray Y-MP for larger problems.

Related and Future Work

Work is in progress to improve fundamental Fortran 77 technology such as ADIFOR and to provide automatic differentiation (AD) functionality for C programs, resulting in more efficient derivative tools for black-box differentiation. Techniques are also under development to exploit algorithmic structures (for example, iterative solution procedures common to many programs) in the application of AD tools. Exploitation of such structure is beneficial both to increase the numerical robustness of the derivative code and to speed up derivative computation[20]. This approach is closely related to the incremental iterative form for computing CFD sensitivities[21-23] and applicable to computing the sensitivity of optimal solutions.[24,25]

Further parallelism can be exposed by exploiting the associativity of the derivative chain rule. Consider the simulation illustrated in Figure 3 consisting of three codes, denoted by f, g, and h, which may represent different disciplines. Code g cannot start before f has finished, and h depends on g's output. To get the desired sensitivities $\frac{du}{dr}$, the serial code for computing this derivative could be strip-mined as we have done here. Spawning processes to separately compute $\frac{df}{dr}, \frac{dg}{ds}, \text{and } \frac{dh}{dt}$ and then multiplying these matrices to obtain the desired sensitivities is an even better technique because the job will execute in less time and utilize the machine resources more effectively. The evaluation of f will be completed before the process that is computing $\frac{df}{dr}$ has finished; thus, another process to compute $\frac{dg}{ds}$ can follow the first completed process. As a result, the individual sensitivity matrices $\frac{df}{dr}, \frac{dg}{ds}, \text{and } \frac{dh}{dt}$ will be computed in a staggered fashion, in parallel, as shown in Figure 4. Note that in order to break the serial dependence in the derivative computation, we have to evaluate f and g twice, once in the code that computes the derivatives $\frac{df}{dr}$ and $\frac{dg}{ds}$, and once in the "function evaluation" chain, to generate the input needed for the next derivative process. We also note that the strip-mining technique can be applied to each of these processes as well.

In this paper, Fortran M was applied after the ADIFOR processing, but tools like ADIFOR must be provided for codes that are written in parallel languages such as Fortran M or High Performance Fortran. This avenue is particularly critical for high-fidelity codes because an AD tool is unlikely to generate sensitivities with less memory or time than the original simulation. No work has yet

been attempted in applying AD to a simulation code that employs explicit message passing calls to achieve fine-grained parallelization because the present ADIFOR tool does not have the capability to imbed the necessary message passing constructs which would allow for a fine-grained parallelization of the derivative calculations. This capability may be incorporated within future versions of the ADIFOR tool.

Conclusions

This work demonstrates the potential of using coarse-grained parallel computing techniques to reduce the execution time required in the computation of many design sensitivities for multidisciplinary problems via automatic differentiation. The blending of the automatic differentiation and the use of a strip-mining parallel computation technique, under the coordination of a Fortran M master process, provides a fast, efficient, and easily implemented means to compute a large number of sensitivities as may be required for typical aircraft design problems. Significant speed ups were achieved for both a vortex lattice code (WINGDES) and a thin-layer Navier-Stokes code (TLNS3D) for transonic flow.

Comparisons were also made between the coarse-grained parallel implementation and a traditional vector supercomputer implementation of the derivative computation for up to 60 design variables. For up to 5 design variables, the vector supercomputer implementation is much faster, but for 6 or more design variables, the coarse-grained parallel implementation is nearly as fast in execution time, and can be much faster in elapsed job turn around time. The memory and queue limitations on the vector computer implementation also may also severely restrict the size or length of the calculation which can be done, relative to the parallel computer implementation.

Acknowledgments

The authors thank Jean-Francois Barthelemy and Eric Unger for making the WINGDES code available to us. We express sincere thanks to Alan Carle for his essential contributions to the ADIFOR project, Bill Gropp for his untiring help in SP1 matters, and Ian Foster, Paul Hovland, Steve Tuecke, and Bob Olson for helping us getting started with Fortran M. We express thanks to Veer Vatsa, Perry Newman, and Tom Zang for their contributions in support of this project. Christian H. Bischof and Timothy L. Knauff, Jr. were supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38, and by the National Aeronautics and Space Administration under Purchase Order L25935D. The authors also gratefully acknowledge the use of the Argonne High-Performance Computing Research Facility. The HPCRF is funded principally by the U.S. Department of Energy, Office of Scientific Computing.

References

1. Bischof, C.; Corliss, G.; Green, L.; Griewank, A.; Haigler, K.; and Newman, P.: “Automatic Differentiation of Advanced CFD Codes for Multidisciplinary Design.” *Computing Systems in Engineering* **3**(6), 1993, pp. 625–637. Also presented at the Symposium on High-Performance Computing for Flight Vehicles, Arlington, VA, Dec. 7–9, 1992.
2. Green, L.; Newman, P.; and Haigler, K.: “Sensitivity Derivatives for Advanced CFD Algorithm and Viscous Modelling Parameters via Automatic Differentiation.” AIAA 93–3321, 1993.
3. Green, L.; Bischof, C.; Carle, A.; Griewank, A.; Haigler, K.; and Newman, P.: “Automatic Differentiation of Advanced CFD Codes with Respect to Wing Geometry Parameters for MDO.” Abstracts from the Second U.S. National Congress on Computational Mechanics, Washington, D.C., August 16–18, 1993, p. 136.
4. Barthelemy, J. F.; and Hall, L.: “Automatic Differentiation as a Tool in Engineering Design.” In *Fourth AIAA/USAF/NASA/OAI Symposium on Multidisciplinary Analysis and Optimization, A Collection of Technical Papers*, Cleveland, OH, AIAA 92–4743–CP, 1992, pp. 424–432.

5. Unger, E. R.; Hall, L. E.: "The Use of Automatic Differentiation in an Aircraft Design Problem." AIAA 94–4260, 1994.
6. Griewank, A.; and Corliss, G. F., eds.: *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, SIAM, Philadelphia, PA, 1991.
7. Griewank, A.: "On Automatic Differentiation." *Mathematical Programming: Recent Developments and Applications*, Iri, M.; and Tanabe, K., eds., Kluwer Academic Publishers, Boston, MA, 1989, pp. 83–108.
8. Rall, L. B.: "Automatic Differentiation: Techniques and Applications." *Lecture Notes in Computer Science*,**120**, Springer— Verlag, Berlin, 1981.
9. Bischof, C. H.; Carle, A.; Corliss, G. F.; Griewank, A.; and Hovland, P.: "ADIFOR: Generating Derivative Codes from Fortran Programs." *Scientific Programming*, **1**(1), 1992, pp. 1–29.
10. Bischof, C. H.; and Hovland, P.: "Using ADIFOR to Compute Dense and Sparse Jacobians. ADIFOR Working Note #2." ANL-MCS-TM-158, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.
11. Bischof, C. H.; Corliss, G. F.; and Griewank, A.: "ADIFOR Exception Handling. ADIFOR Working Note #3." ANL-MCS-TM-159, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.
12. Bischof, C. H.; Carle, A.; Corliss, G. F.; Griewank, A.; and Hovland, P.: "Getting Started With ADIFOR. ADIFOR Working Note #9." ANL-MCS-TM-164, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.
13. Averick, B.; More, J.; Bischof, C.; Carle, A.; and Griewank, A.: "Computing Large Sparse Jacobian Matrices using Automatic Differentiation." ANL-MCS-P348–0193, Mathematics and Computer Science Division, Argonne National Laboratory, 1993. Also to appear in *SIAM J. Scientific Computing*.
14. Foster, I. T.; and Mani Chandy, K.: "Fortran M: A Language for Modular Parallel Programming." ANL-MCS-P327-0992, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.
15. Foster, I.; Olson, R.; and Tuecke, S.: "Programming in Fortran M." ANL-93/26, (Rev. 1), Mathematics and Computer Science Division, Argonne National Laboratory, October 1993.
16. Vatsa, V. N.; and Wedan, B. W.: "Development of a Multigrid Code for 3–D Navier-Stokes Equations and Its Applications to a Grid-Refinement Study." *Computers & Fluids,* **18**(4), 1990, pp. 391–403.
17. Carlson, H.; Wackley, K.; and Barger, R.: "Numerical Methods and a Computer Program for Subsonic and Supersonic Aerodynamics Design and Analysis of Wings with Attainable Thrust Considerations." Technical Report NASA CR-3808, Kentron International Inc., 1985.
18. Gropp, W.: "Early Experiences with the IBM SP/1 and the High-Performance Switch." ANL—MCS–93/41, Mathematics and Computer Science Division, Argonne National Laboratory, 1993.
19. Bell, R.: "IBM RISC System/6000 NIC Tuning Guide for Fortran and C," Document GG24-3611-01, 1991.
20. Griewank, A.; Bischof, C.; Corliss, G.; Carle, A.; and Williamson, K.: "Derivative Convergence of Iterative Equation Solvers. ANL-MCS-P333-1192, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.
21. Korivi, V. M.; Taylor, A. C. III; Newman, P. A.; Hou, G. J.-W.; and Jones, H. E.: "An Approximately-Factored Incremental Strategy for Calculating Consistent Discrete Aerodynamic Sensitivity Derivatives." In *Fourth AIAA/USAF/NASA/OAI Symposium on Multidisciplinary Analysis and Optimization, A Collection of Technical Papers,* Cleveland, OH, pp. 465–478. AIAA 92–4746–CP, 1992.
22. Newman, P. A.; Hou, G. J.-W.; Jones, H. E.; Taylor, A. C. III ; and Korivi, V. M.: "Observations on Computational Methodologies for Use in Large-Scale Gradient-Based Multidisciplinary Design." In *Fourth AIAA/USAF/NASA/OAI Symposium on Mul-*

tidisciplinary Analysis and Optimization, A Collection of Technical Papers, Cleveland, OH, pp. 531–542. AIAA 92–4753–CP, 1992.

23. Korivi, V. M.; Taylor, A. C. III; Hou, G. J.-W.; Newman, P. A.; and Jones, H. E.: "Sensitivitity Derivatives for Three-Dimensional Supersonic Euler Code Using Incremental Iterative Strategy." In *11th AIAA Computational Fluid Dynamics Conference, A Collection of Technical Papers, Part 2,* Orlando, FL, 1993, pp. 1053–1054. Expanded version accepted for publication in *AIAA J.*
24. Sobieszczanski-Sobieski, J.; Barthelemy, J.-F.; and Riley, K. M.: "Sensitivity of Optimum Solutions to Problem Parameters." *AIAA J.*, **20**(9), 1982, pp. 1291–1299.
25. Barthelemy, J.-F.; and Sobieszczanski-Sobieski, J.: "Optimum Sensitivity Derivatives of Objective Functions in Nonlinear Programming." *AIAA J.*, **21**(6), 1983, pp. 913–915.

Table 1. Cray Y-MP Single Processor Timing (seconds) for TLNS3D.AD Code with Two–Level Multigrid on 97×25×17 Viscous Grid

NDV	Time/Iter. (coarse grid)	Time/Iter. (fine grid)	Time/NDV (fine grid)
01	0.51	05.08	5.08
02	0.70	07.19	3.59
04	1.09	11.85	2.96
06	3.63	40.68	6.78
08	3.74	44.59	5.57
10	3.89	43.76	4.38
12	4.02	45.95	3.83
48	6.62	76.92	1.60
60	6.19	72.17	1.20

Table 2. IBM SP1 Multiprocessor Timing (seconds) for WINGDES.AD Code

Number of processors	Average time	Standard deviation
1	158.2	0.00
2	92.0	2.19
4	58.4	1.83
8	41.8	1.09

Table 3. IBM SP1 Multiprocessor Timing (seconds) for TLNS3D.AD Code with Two–Level Multigrid on 97×25×17 Viscous Grid (NDV=60)

Number of processors	Average time	Standard deviation
01	34,160 (estimated)	0.00
05	6,832	53.1
15	3,757	11.5

Table 4. IBM SP1 Single-Processor Timing (seconds) for TLNS3D.AD Code with Two–Level Multigrid on 97×25×17 Viscous Grid

NDV	Time/iter. (coarse grid)	Time/iter. (fine grid)	Time/NDV (fine grid)
01	4.34	51.5	51.5
02	5.00	58.7	29.4
04	6.08	66.9	16.7
06	7.54	82.1	13.7
08	8.74	97.4	12.2
10	9.76	108.5	10.8
12	10.76	128.2	10.7

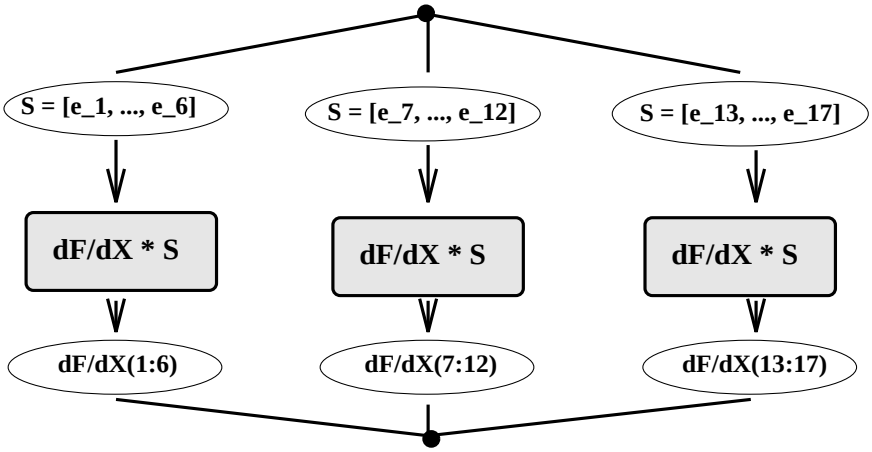

Figure 1: An Example of Parallel Sensitivity Computation through Derivative Stripmining

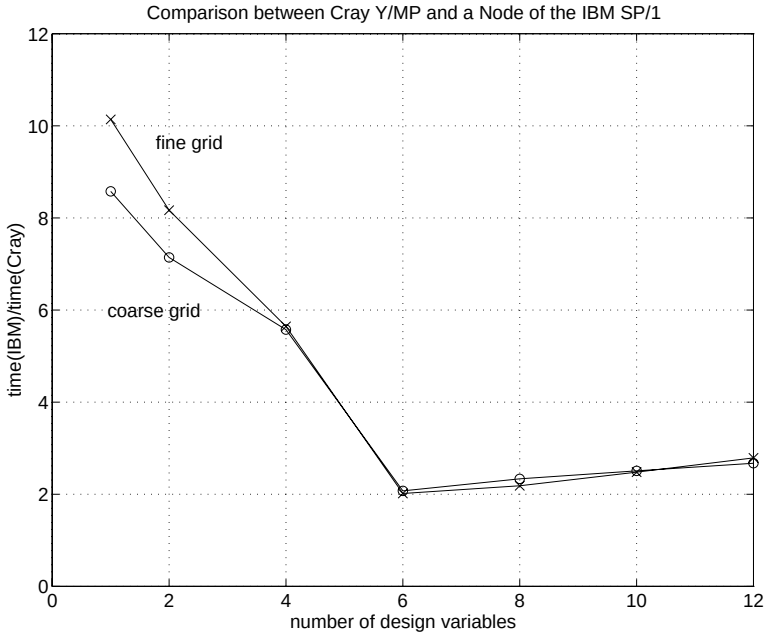

Figure 2: Cray vs. IBM Performance.

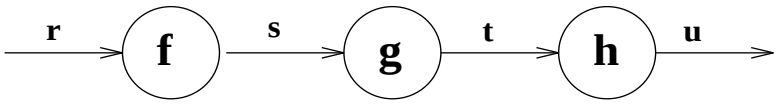

Figure 3: A Sequence of Simulation Modules

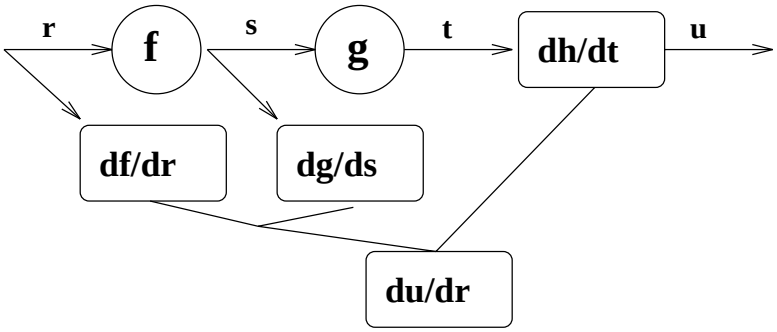

Figure 4: Exploiting Chain Rule Associativity to Break Dependencies