Efficient Computation of Gradients and Jacobians by Dynamic Exploitation of Sparsity in Automatic Differentiation*

Christian H. Bischof,† Peyvand M. Khademi,† Ali Bouaricha,† and Alan Carle‡

Argonne Preprint MCS-P519-0595

Abstract. Automatic differentiation (AD) is a technique that augments computer codes with statements for the computation of derivatives. The computational workhorse of AD-generated codes for first-order derivatives is the linear combination of vectors. For many large-scale problems, the vectors involved in this operation are inherently sparse. If the underlying function is a partially separable one (e.g., if its Hessian is sparse), many of the intermediate gradient vectors computed by AD will also be sparse, even though the final gradient is likely to be dense. For large Jacobians computations, every intermediate derivative vector is usually at least as sparse as the least sparse row of the final Jacobian. In this paper, we show that dynamic exploitation of the sparsity inherent in derivative computation can result in dramatic gains in runtime and memory savings. For a set of gradient problems exhibiting implicit sparsity, we report on the runtime and memory requirements of computing the gradients with the ADIFOR (**A**utomatic **DI**fferentiation of **FOR**tran) tool, both with and without employing the SparsLinC (**Spars**e **Lin**ear **C**ombinations) library, and show that SparsLinC can reduce runtime and memory costs by orders of magnitude. We also compute sparse Jacobians using the SparsLinC-based approach—in the process, automatically detecting the sparsity structure of the Jacobian—and show that these Jacobian results compare favorably with those of previous techniques that require a priori knowledge of the sparsity structure of the Jacobian.

Key words. Automatic Differentiation, Sparsity, Partial Separability, Sparse Jacobians, Large-Scale Optimization, MINPACK-2, ADIFOR, SparsLinC.

1 Introduction

The numerical computation of gradients and Jacobians is an important step in the solution of many nonlinear problems, such as numerical optimization, mesh computations, nonlinear least squares, and systems of differential and algebraic equations. For such problems, the derivative computation is often a major contributor to the overall cost, in terms of both runtime and memory requirements.

*This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, by the National Aerospace Agency under Purchase Order L25935D and Cooperative Agreement No. NCCW-0027, and by the National Science Foundation, through the Center for Research on Parallel Computation, under Cooperative Agreement No. CCR-9120008.

†Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439, (bischof,khademi,bouarich)@mcs.anl.gov.

‡Center for Research on Parallel Computation, Rice University, 6100 S. Main St., Houston, TX 77251, carle@cs.rice.edu.

Automatic differentiation (AD) is a technique by which codes for the computation of functions are augmented to produce codes for the computation of desired derivatives of those functions. AD relies on the fact that every function is executed on a computer as a sequence of elementary operations, such as + and −, and intrinsics, such as sin and log. By successive applications of the chain rule to the composition of those elementary operations, derivatives can be computed exactly (up to machine roundoff) and in a completely mechanical fashion. For a detailed description of AD, see [20].

Previous studies have shown that AD is a powerful technique for computing derivatives. Accuracy and runtime efficiency of AD are superior to difference approximations, and the reliability, flexibility and development-time efficacy of AD surpass those of hand-coding approaches. For example, the AD tool ADIFOR (Automatic DIfferentiation of FORtran) [5, 7, 8] has been used to generate derivative codes for many applications in areas such as large-scale optimization [2, 14], computational fluid dynamics [9, 10, 15], weather modeling [26], and groundwater modeling [11]. Other examples of available AD tools are Odyssée [27] and GRESS [23] for Fortran, and ADOL-C [21] and ADIC [13] for C programs.

The forward and reverse modes are the two basic modes of AD, and are distinguished by the manner in which the chain rule is applied for propagating derivatives. This basic difference impacts the computational complexity and flexibility of each mode. The forward mode, despite its flexibility and the predictability of its memory requirements, has thus far been considered impractical for the computation of large-scale gradients, because of its runtime complexity. In this paper, we consider only the forward mode, and show that in some cases this perceived limitation of the forward mode can be overcome by the exploitation of sparsity.

Computationally, the most expensive kernel of derivative codes generated by primarily forward mode tools such as ADIFOR and ADIC is a vector linear combination (VLC):

$$w = \sum_{i=1}^{k} \alpha_i v_i, \tag{1.1}$$

where w and the v_i are gradient vectors, the α_i are scalar multipliers, and k is the arity. If we assume that the gradient vectors are all of same size, say d, then a computation composed primarily of invocations of (1.1), implemented as a dense vector loop, would have computational complexity linear in d, in terms both of runtime and memory requirements (this type of implementation is used in the default or `NONSPARSE` mode of ADIFOR, examples of which we will show in Section 3). On the other hand, if the gradient vectors are sparse, then strategies that attempt to exploit that sparsity, in terms of both the storage of vectors and the implementation of the computation, can be expected to reduce the overall computational complexity. For any given problem, the extent of this reduction in complexity will depend not only on the efficiency of the sparse algorithm, but also on the sparsity characteristics of the underlying function.

For many large-scale problems there is inherent sparsity in the computation of gradients and Jacobians. In gradient computations, the scalar output function, $f : \mathbf{R}^n \rightarrow \mathbf{R}$, is usually dependent on all of the inputs, resulting in a dense gradient vector $g = \nabla f \in \mathbf{R}^n$. However, it has been shown that if the Hessian of f is sparse, then f is partially separable

[22]; that is, f can be expressed as

$$f(x) = \sum_{i=1}^{m} f_i(x), \tag{1.2}$$

where m is the number of element functions $f_i(x)$, and each $f_i(x)$ is typically a function of just a few of the components of x, implying that each $g_i(x) = \frac{df_i}{dx}$ will be sparse, even though $g(x) = \sum_{i=1}^{m} g_i(x)$ is dense.

Sparsity in the derivative computations is likewise a salient issue in many large-scale Jacobian computations where the final Jacobian associated with a function $f : \mathbf{R}^n \rightarrow \mathbf{R}^m$ is sparse. As we will see in Section 2, the sparsity of the final derivative result implies equal or greater potential for sparsity in the intermediate computations.

The various approaches used for exploiting sparsity in Jacobian computations can generally be grouped into two categorizes: *static* and *dynamic*. Static approaches use a priori knowledge of the sparsity structure of the sparse Jacobian to map the Jacobian to an equivalent Jacobian – referred to as the compressed Jacobian – with significantly fewer columns. The success of the static method depends on finding a minimal number of columns, since this number is a good approximation for the computational complexity of these methods, as a multiplicative factor of the cost to compute the function. Coleman and Moré [17] introduced an algorithm for a terse mapping based on graph-coloring heuristics, and successfully applied this algorithm to large-scale optimization problems. Their approach applies equally well to automatic differentiation as it does to finite differencing, and we make use of it in this paper. Using an alternative approach, Newsam and Ramsdell [25] showed that the sparse Jacobian can be estimated with $q + 1$ function evaluations, where q is the maximum of the number of nonzero elements in any row of the Jacobian. However, this approach requires the solution of a system of linear equations with a potentially ill-conditioned matrix. Hence, we decided to employ the numerically safe Coleman/Moré approach.

The dynamic approach, which can also be applied to gradient computations, involves using dynamically allocated memory and sparse data representations for storing and processing only the nonzero information in the derivative objects. An implementation of this approach was introduced by Dixon, Maany, and Mohseninia [19], using the operator overloading capabilities of ADA, and specific to codes written in that language. For first order derivative computations, their implementation of a *sparse doublet* consists of a linked list, with each entry representing a nonzero in terms of its position in the vector, its value, and a pointer to the next entry. A more efficient implementation was introduced by M.C. Bartholomew-Biggs, L. Bartholomew-Biggs, and B. Christianson [3], who used a recurrent recycling scheme for freed up dynamically-allocated memory. These and others works [18] demonstrated the runtime efficiency of the dynamic approach for some small- to medium-sized problems ($n \leq 500$).

The SparsLinC (**Sparse Linear Combinations**) library [6, 8] is a software package for exploiting sparsity in AD, using the dynamic approach. In this paper, we report on the runtime and memory performance measures of differentiating large-scale ($n \leq 160,000$) optimization codes with ADIFOR when interfaced with SparsLinC, and contrast these with `NONSPARSE` ADIFOR results (i.e., without SparsLinC). The codes are taken from the MINPACK-2 test set problems [1] and the molecular distance geometry class of global minimization functions [24]. In Section 2 we look at sparsity in the context of the computation of VLCs in AD and

show how this can be exploited with SparsLinC. Computational results for the gradient and Jacobian experiments are shown in Sections 3 and 4, respectively. Finally, we present our conclusions in Section 5.

2 Sparsity in First-Order Derivative Computation

For large-scale problems, gradient vectors appearing in VLCs in AD computations are often very sparse. In this section, we define sparsity in the context of AD computations, and identify characteristic problem types where sparsity in derivative computations occurs. We also give a brief overview of SparsLinC and show how it exploits sparsity.

2.1 Definition of Sparsity in Automatic Differentiation

To define sparsity, we first revisit some basic AD and ADIFOR definitions. All AD tools require the user to specify, from among the program variables, the *independent and dependent variables*; the independents being the ones with respect to which the partial derivatives of the dependents are to be computed. In addition, ADIFOR performs an activity analysis pass, whereby all variables possibly lying on the dependency path from the independent to the dependent variables are nominated as *active variables* (by definition, the independents and dependents are themselves active).

A *directional derivative*, defined as

$$\lim_{\mathtt{h}\to 0} \frac{f(x+\mathtt{h}\cdot e)-f(x)}{\mathtt{h}}, \tag{2.1}$$

is the partial derivative of an active variable along a direction vector e. In ADIFOR, the user specifies the complete set of desired directions by means of the *seed matrix*. In the simplest case, each unit direction is defined by one of the independent variables, which is equivalent to setting the seed matrix to the identity matrix. A *directional gradient vector* is defined to be the set of directional derivatives of any scalar active variable with respect to all directions specified in the seed matrix (the term *scalar active variable* here refers to active variables declared as scalars in the program and also to the individual elements of active variables that are declared as arrays). In the context of AD, the vector operands in (1.1) are directional gradient vectors.

In the `NONSPARSE` representation, a directional gradient vector V would be declared in the derivative code as an array variable of length d, where d is the number of directions. We denote the number of nonzeros in V at a given point t during the execution by $V_{t.nonz}$. The percentage of zero entries or *sparsity* of V_t is defined as

$$V_{t.sparsity} := (1 - \frac{V_{t.nonz}}{d}) * 100\%. \tag{2.2}$$

A good measure for the *overall sparsity* present in a derivative computation is the median of the sparsities of all directional gradient vectors computed during the execution of the derivative code. We also define the umbrella term *sparsity characteristics* to refer to the sparseness of all directional gradient vectors during the execution of a derivative code.

One implication of (2.2) is that d sets an upper bound on the overall sparsity in a problem. For example, if $d = 10$, overall sparsity is bounded above by 90% (given that,

from the definition of active variables, usually $V_{t.nonz} \geq 1$). This underscores a necessary (but not sufficient) condition in terms of the practicality of a sparse solution: in order for SparsLinC—or any other strategy for exploiting sparsity in AD—to improve the runtime performance of derivative computation, the number of directions with respect to which we wish to compute derivatives should be "large." Otherwise, the directional gradient vectors will be short, and runtime efficiences gained by exploiting sparsity will be defeated by implementation overheads. The determination of what is considered a large sparse problem is to a great extent dependent upon the nature of the problem.

Another important fact relating to sparsity in derivative computations is that in all likelihood the final sparsity of the least sparse directional gradient vector corresponding to the dependent variables sets the upper bound on the number of nonzeroes of all intermediate directional gradient vectors during the execution. This means that the overall sparsity of the problem may be much higher than that of the final derivative result. This fact follows from the formation of structural merges in VLCs. The notational equation (2.3) is a depiction of a structural merge, that is, the additive propagation of nonzero structures of the right-hand-side vectors to the left-hand-side vector:

$$\begin{pmatrix} \diamond \\ \diamond \\ \\ \diamond \\ \diamond \end{pmatrix} \Leftarrow \begin{pmatrix} \\ \diamond \\ \\ \\ \end{pmatrix} + \begin{pmatrix} \diamond \\ \\ \\ \diamond \\ \end{pmatrix} + \begin{pmatrix} \\ \\ \\ \diamond \\ \diamond \end{pmatrix}. \tag{2.3}$$

Here, the symbol $\diamond$ represents a nonzero entry (zero entries are left blank), $d = 5$, and the nonzero index set of the resulting left-hand-side vector is the union of index sets of the right-hand-side vectors.* Given that a left-hand-side vector in a VLC will usually appear as a right-hand-side vector in a subsequent VLC, it is easy to see that the impact of the statement-level structural merge in (2.3) is that sparsity diminishes as the computation proceeds. It is the converse of this result that motivates our work, namely, if we suspect that the final derivative object we seek is sparse, then we can expect that all directional gradients vectors (i.e., operands of all VLCs in the derivative computation) will be as sparse—and, with great likelihood, more sparse.

Figure 2.1 depicts the sparsity characteristics of directional gradient vectors in the gradient computation of the GL2 problem, which will be introduced in Section 3. The computation involved over 30,000 VLCs. For every 250 VLCs, we gathered statistics on the number of nonzeroes in the resultant left-hand-side vectors and plotted the minimum, maximum, median, and average on the graph. We note that the fact that the maximum drops from about 2,500 nonzeroes to about 500 at around the 26,000-th VLC is not indicative of a decrease in the number of nonzeroes of a particular vector, but rather that among the vectors appearing on the left-hand side of the 250 VLCs previous to the 26,000-th, the one

*This discussion precludes the possibility of the occurrence of (*i*) numerical zeroes resulting from exact cancellation (e.g., $a + (-a)$), and (*ii*) zero multipliers. In our experience, exact cancellation rarely occurs in derivative computation, and currently, SparsLinC does not check for it (i.e., numerically zero vector entries are treated like nonzero entries). SparsLinC does, however, check for zero multipliers, and vectors with zero multipliers are not referenced in VLC computations.

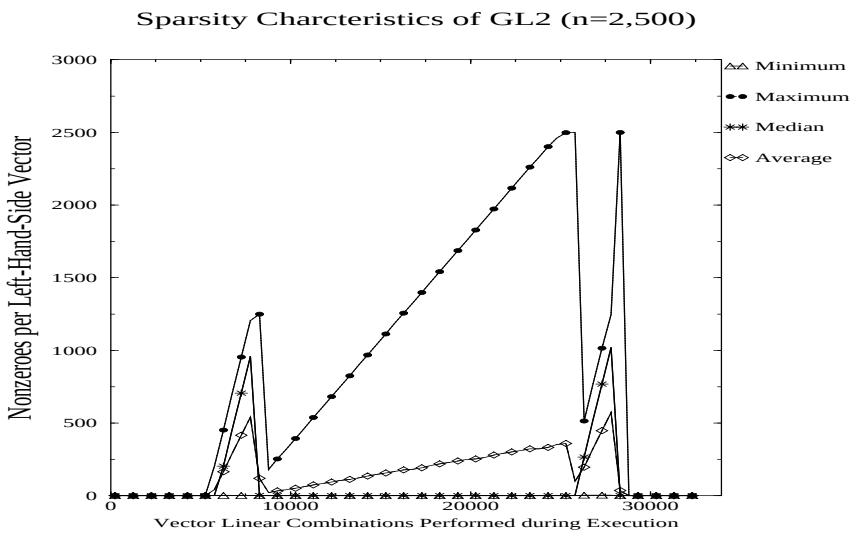

Figure 2.1: Sparsity of vectors on the left-hand side of vector linear combinations performed by SparsLinC during the execution of the gradient code for GL2, n = 2,500.

with the maximum number of nonzeroes had about 500. In fact, the final gradient vector is fully dense (with 2,500 nonzeroes).

The main conclusion to be drawn from Figure 2.1 is that the median and average curves are always far below 2,500, the maximum vector size, thus implying that significant savings could be achieved in the computation of VLCs by exploiting sparsity. We note that the overall sparsity inherent in a derivative computation is largely a function of sparsity present in intermediate computations, and not so much the sparsity of the final result.

2.2 Characteristic Sparse Derivative Problems

Partially separable functions and their gradients arise in many computational contexts, in particular in large-scale optimization problems. In Section 1 we defined these functions (1.2) as a summation of element functions. Since each element function depends only on very few components of the independent variable, each corresponding directional gradient vector is sparse, and typically, the only dense vector present in the computation is the gradient of the partially separable function itself (the example in Figure 2.1 is one such case).

In terms of how to exploit the sparsity in these gradient computations, it is important to note that the mathematical notion of separability defined in (1.2) may not always be reflected in terms of how the code for a partially separable function is written (i.e., the separability may not be reflected in the code structure). Though in some cases the explicit reformulation of the code for such functions is a viable alternative leading to effcient derivative computations of equivalent compressed Jacobians (see Section 4.1 and also [4]), in other cases, an algorithmic approach that transparently (i.e., without the need to rewrite code) exploits this "under-the-rug" sparsity may be preferable. Our results in Section 3 will demonstrate the efficiency of computing gradients of partially separable functions using the latter approach as implemented in SparsLinC.

Large sparse Jacobian problems occur frequently in many contexts, and computing them without effective sparse strategies either has been prohibitively expensive or has depended on a priori knowledge of the closure of the sparsity patterns of the Jacobian, where closure denotes the union of the sparsity patterns for all values of the independent variables. In contrast to partially separable functions where some gradient vectors may be dense, the AD computation of sparse Jacobians, by definition, involves only sparse directional gradient vectors.

In Section 4 we briefly review the compressed Jacobian approach, which has previously been used [17, 2] as an effective strategy for computing sparse Jacobians in cases where the closure sparsity pattern is known. We then present our results of computing sparse Jacobians with known closures using the ADIFOR/SparsLinC approach, and contrast these results with those based on the compressed Jacobian method.

2.3 Exploiting Sparsity with SparsLinC

Our intent is to take advantage of inherent sparsity in directional gradient vectors to reduce both memory requirements and runtime of derivative computation. By devising a scheme that would preclude the storage of zero entries as well as extraneous "zero-sum" computations resulting from the zero entries, we can save on memory and runtime. The dynamic nature of the directional gradient vectors during the execution of AD-generated derivative codes, in terms of the number of nonzero entries in a vector at a given time, as well as the goal of obviating the need for a priori sparsity information requires that the vectors be represented by using dynamic data structures.

SparsLinC is a library of C routines that provide an implementation of VLCs employing dynamic data structures to represent the directional gradient vectors. When invoked in its `SPARSE` mode, ADIFOR generates code in which each VLC is implemented as a call to a SparsLinC routine, as opposed to the `DO`-loop implementation in the default (`NONSPARSE`) mode. Whereas for every scalar active variable, `NONSPARSE` ADIFOR allocates a corresponding directional gradient vector of size d containing the derivative values (many of which may possibly remain zero for the duration of the computation), `SPARSE` ADIFOR allocates an `INTEGER` variable that is interpreted as a pointer to SparsLinC's representation of the vector.

The design of SparsLinC is based on representing the nonzero information in each vector (i.e., the nonzero values and their corresponding vector indices) in one of three data structures—one to represent a vector with zero or one nonzero entries (S1); another for a vector with a few nonzeroes (SS), and a third for a vector with contiguous ranges of nonzeroes (CS). The SparsLinC polyalgorithm heuristically switches between these representations. SparsLinC performs the necessary management of its internal vector representations, including the implementation of a "grow as you go" strategy for the recruitment of dynamically allocated memory for the storage of the sparse vectors, and a recurrent recycling scheme for the reuse of memory. The SS and CS sparse representations make use of a *bucket* storage scheme, where each bucket consists of a (user-configurable) number of elemental data types. Memory is dynamically allocated in units of *stores* of buckets. In contrast to other sparse implementations [3, 19] which allocate memory for each nonzero separately, the bucket scheme allows for greater flexibility and runtime efficiency, at the cost of introducing some storage overhead (not all of the allocated memory is necessarily

Table 3.1: MINPACK-2 Unconstrained Optimization Problems

Name	Description of the Problem
EPT	Elastic-Plastic Torsion
GL2	Ginzburg-Landau (2-d) superconductivity
MSA	Minimal Surface Area
ODC	Optimal Design with Composite materials
PJB	Pressure distribution in a Journal Bearing
SSC	Steady State Combustion

used).

The basic module of SparsLinC performs a VLC (1.1) by using a heapsort to pop the smallest-indexed nonzero (or nonzero range, in the CS case) present among the right-hand side vectors, and constructing the left-hand-side vector in a one-pass traversal. A special "plus-equals" module takes advantage of the existing data structure of the left-hand-side vector in cases in which the same vector also appears on the right-hand side, thus resulting in a more efficient computation.

Two features of SparsLinC are of particular relevance to the present discussion: the transparent exploitation of partial separability, and automatic detection of the sparsity pattern of the Jacobian. Previous attempts at exploiting partial separability in the context of AD have been based on manual modification of the code for the function in such a way as to transform the derivative computation from a dense gradient problem to a sparse Jacobian problem [12]. The transparent exploitation of partial separability in SparsLinC does not require code modification. Instead, it relies on the sparse representation and processing of the VLCs involving the element functions, and on the efficiency of the plus-equals module for the accumulation of the dense gradient vector.

The detection of the sparsity pattern of Jacobians is of interest in a number of computations (e.g., see [14]). As we shall see in Section 4, sparsity detection is a prerequisite for compressing Jacobians using coloring methods. In our Jacobian experiments we use prepackaged MINPACK-2 routines for detecting the sparsity pattern of the Jacobian for each problem. However, in the absence of such special routines (or in cases where the sparsity pattern varies for differing values of the independent variable), we can use SparsLinC for this purpose. The computation of the Jacobian using SparsLinC yields the sparsity pattern of the Jacobian as a natural byproduct of the work it does in computing the Jacobian.

A detailed description of the SparsLinC library will be reported elsewhere, and the full Fortran interface specification can be found in [6].

3 Computing Gradients of Partially Separable Functions

The MINPACK-2 Test Problem Collection [1] contains a number of unconstrained optimization problems from a variety of application areas. Table 3.1 shows the six MINPACK-2 problems we used in our gradient experiments. For each problem, we computed the gradient using both the `NONSPARSE` ADIFOR approach and the `SPARSE` ADIFOR/SparsLinC

Table 3.2: Memory Requirements for Gradient Problems (in Mbytes; $n = 160,000$)

Problem	$M\{F\}$	$M\{G_{AD-16}\}$	$M\{G_{AD-16}\}/M\{F\}$	$M\{G_{Sparse}\}$	$M\{G_{Sparse}\}/M\{F\}$
EPT	1.29	23.06	17.8	14.15	11.0
GL2	2.59	45.03	17.4	19.65	7.6
MSA	2.57	24.34	9.5	12.60	4.9
ODC	1.29	23.06	17.8	13.05	10.1
PJB	1.29	23.06	17.8	14.16	11.0
SSC	1.29	23.06	17.8	14.15	11.0

approach, and compared the memory requirements and runtimes of the two approaches (in what follows we will simply refer to the two approaches as `NONSPARSE` and `SPARSE`, respectively). We performed our experiments on two workstation platforms: Sun SPARCstation IPX, and IBM RS6000-370.

The gradient values obtained for all problems using both the `NONSPARSE` and `SPARSE` approaches agreed to within machine precision of the hand-coded derivatives, available in MINPACK-2.

3.1 Memory Requirements

In terms of feasibility, memory is a critical issue in the computation of large gradients. The memory required for a straightforward AD implementation of the gradient computation would be roughly equivalent to augmenting the memory required for the function computation by a factor of n, where n is number of components of the independent variable, alternately referred to as the problem size. For large n, this linear expansion can lead to excessive paging, or simply be prohibitive. For example, a `NONSPARSE` ADIFOR implementation of the gradient computation for the GL2 problem of size 160,000 would require more than 400 gigabytes of virtual memory.

One approach to breaking the linear memory dependence is stripmining of derivative computation. This approach involves dividing the gradient computation into strips, each strip consisting of the differentiation with respect to a few components of the independent variable, and each strip having reasonable memory requirements (as we will see in Section 4.3, stripmining can also be used in Jacobian computations). In the case of ADIFOR-generated code, stripmining can be done conveniently through the seed matrix mechanism. The runtime penalty for this approach is an extraneous function recomputation per strip, which can be effectively amortized for large strip sizes. Since each strip is computed independently, stripmining can also be used for easy parallelization of the gradient computation [10].

*The Unix command '`size` *executable-file*' reports the total amount of statically allocated memory (i.e., the memory requirements that can be assessed at link-time such as array sizes, etc.) needed to load and run the executable. In the case of SparsLinC, where memory is also allocated dynamically, we call a SparsLinC routine that reports the total amount of dynamically allocated memory, and add this to the statically allocated memory, in order to arrive at the total memory requirements.

Table 3.2 contains a summary of the memory requirements* of our gradient experiments for the case of $n = 160,000$. We use the notation $M\{C\}$ to denote the memory requirements of a computation C, and we report only one set of numbers, since memory requirements on the two test-bed platforms are identical. The first column in Table 3.2 shows the memory required for computing the original function, F. The next double column shows first the memory requirements, $M\{G_{AD-16}\}$, of a stripmined `NONSPARSE` computation with a strip size $p_s = 16$, and then the ratio of this gradient memory requirement to that of the corresponding original function. We note that in each case except one, $M\{G_{AD-16}\}/M\{F\}$ is slightly greater than p_s, matching our expectation of linear augmentation in memory requirements. The exception is the MSA problem, where the $M\{G_{AD-16}\}/M\{F\}$ ratio is smaller than p_s. Here the computations involving a large work array in the original function are unrelated to derivative computation; hence ADIFOR does not generate a corresponding augmented derivative work array (for a full discussion of this issue, see "Variable Nomination" in [8]). We also note that the equivalence of the $M\{F\}$'s and $M\{G_{AD-16}\}$'s of the EPT, ODC, PJB and SSC problems is due to the similarity of the code structures of these problems.

The second double column in Table 3.2 shows the memory requirements of the `SPARSE` computation, $M\{G_{Sparse}\}$, along with the ratio of this gradient memory requirement to that of the corresponding original function. Here we see the terseness of SparsLinC's memory allocation scheme in that $M\{G_{Sparse}\}$ is always within a factor 4.9–11.0 of $M\{F\}$. The variation among these ratios is indicative of the variation in the sparsity characteristics of their corresponding gradient computations. Note that in all cases, $M\{G_{Sparse}\} < M\{G_{AD-16}\}$, despite the fact that G_{Sparse} computes the whole gradient at once, whereas G_{AD-16} computes the gradient a strip at a time.

3.2 Runtime

Figure 3.1 is a summary of all our gradient runtime results. For each of the optimization problems in Table 3.1, we timed the `SPARSE` and the stripmined `NONSPARSE` gradient computations for several problem sizes ranging from $n = 2,500$ to $n = 160,000$. Similar to the case of memory requirements, here we were interested in the augmentation factor of the runtime of the gradient computation with respect to the runtime of the function computation. Subsequently, the plots shown in the six panels of Figure 3.1 are all gradient-to-function runtime ratio plots.

Each panel shows the gradient-to-function ratios for two of the MINPACK-2 problems, for both `SPARSE` and `NONSPARSE` gradient computations on a given machine. Note also that for each panel both axes are logarithmic in scale ($\log_{10}$).

The most salient result evident in Figure 3.1 is the disparity between the `SPARSE` and `NONSPARSE` runtime ratios for each problem. In all cases the `NONSPARSE` gradient computation displays a purely linear behavior with respect to the function computation over the range of problem sizes. Note that in all cases the linear coefficient is very close to 1 (by virtue of the fact that all the `NONSPARSE` plots are close to the $y = x$ diagonal line). As we mentioned in Section 1, this is due to the `DO`-loop implementation of VLCs in the derivative code generated by ADIFOR in the default or `NONSPARSE` mode. On the other hand, the `SPARSE` runtime ratios, though not independent of n, show a markedly reduced dependency

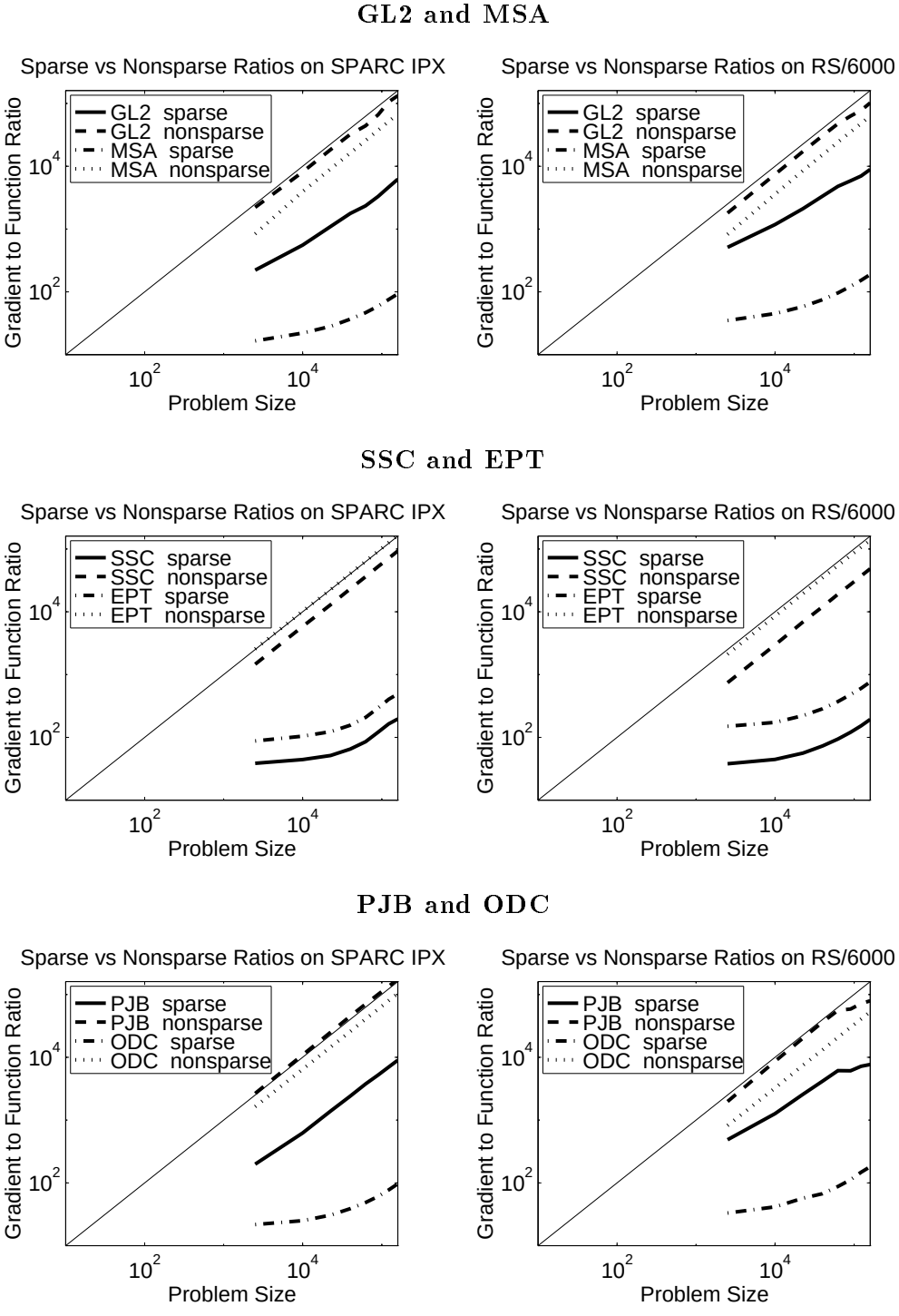

Figure 3.1: Log-Log plots for each platform of gradient-to-function runtime ratios of the `SPARSE` and `NONSPARSE` runs for all gradient problems.

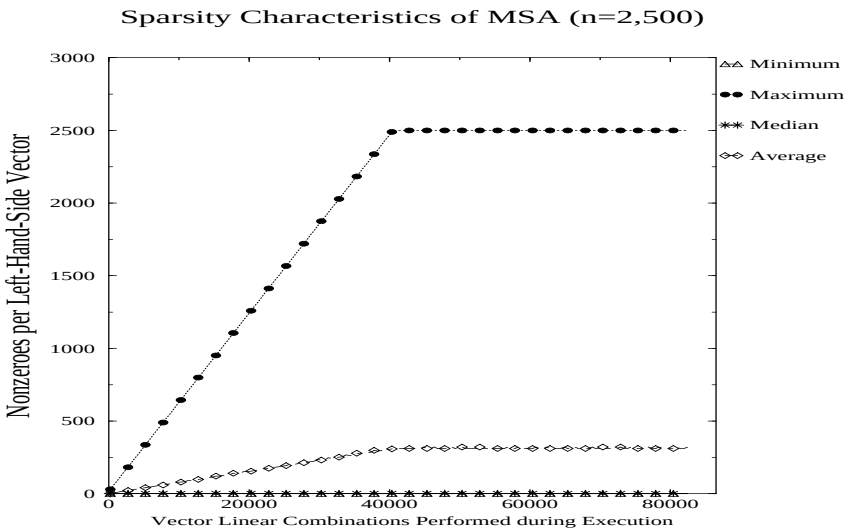

Figure 3.2: Sparsity of vectors on the left-hand side of vector linear combinations performed by SparsLinC during the execution of the gradient code for MSA, n = 2,500.

on n.

Note that in the case of four of the problems (EPT, MSA, ODC, and SSC), gradient computation is clearly feasible with SparsLinC, whereas the `NONSPARSE` computation is so costly that it would be considered impractical. For example, the runtime of computing the MSA and ODC gradients of size 160,000 on the SPARC IPX is less than 100 times the function runtime, which is a reasonable cost for some applications, given that SparsLinC requires no rewriting of code. We also note that this cost can be further reduced by efficient hand-coding of the gradient or, in the case of partially separable functions, by reformulating the gradient computation as a sparse Jacobian computation [4].

The variance between the degree of disparity between `NONSPARSE` and `SPARSE` results is due to the differing sparsity characteristics of the MINPACK-2 problems. Figure 3.2 depicts the sparsity characteristics of the MSA problem, for the case of $n = 2,500$. In contrast to Figure 2.1, where for parts of the GL2 computation we observe medians of approximately 1000 nonzeroes per vector, in the case of the MSA computation the median is always approximately 1. Correspondingly, we note in Figure 3.1 the markedly smaller gradient-to-function ratios for the `SPARSE` MSA computation as compared with the `SPARSE` GL2 computation.

One aspect of the results in Figure 3.1 is that despite the qualitative similarity of the corresponding curves on the SPARC IPX and the RS6000, the quantitative results differ in certain characteristic ways. We note that in general, the disparity between the `SPARSE` and `NONSPARSE` results are somewhat larger on the SPARC IPX. As is clear from the plots, this is caused both by the `SPARSE` ratios being smaller on the SPARC IPX (compare for example the `SPARSE` MSA results on the two platforms) and by the `NONSPARSE` ratios being larger.

These performance differences are a reflection of architectural features of the two platforms. The SPARC IPX essentially has a scalar processor and a flat memory hierarchy.

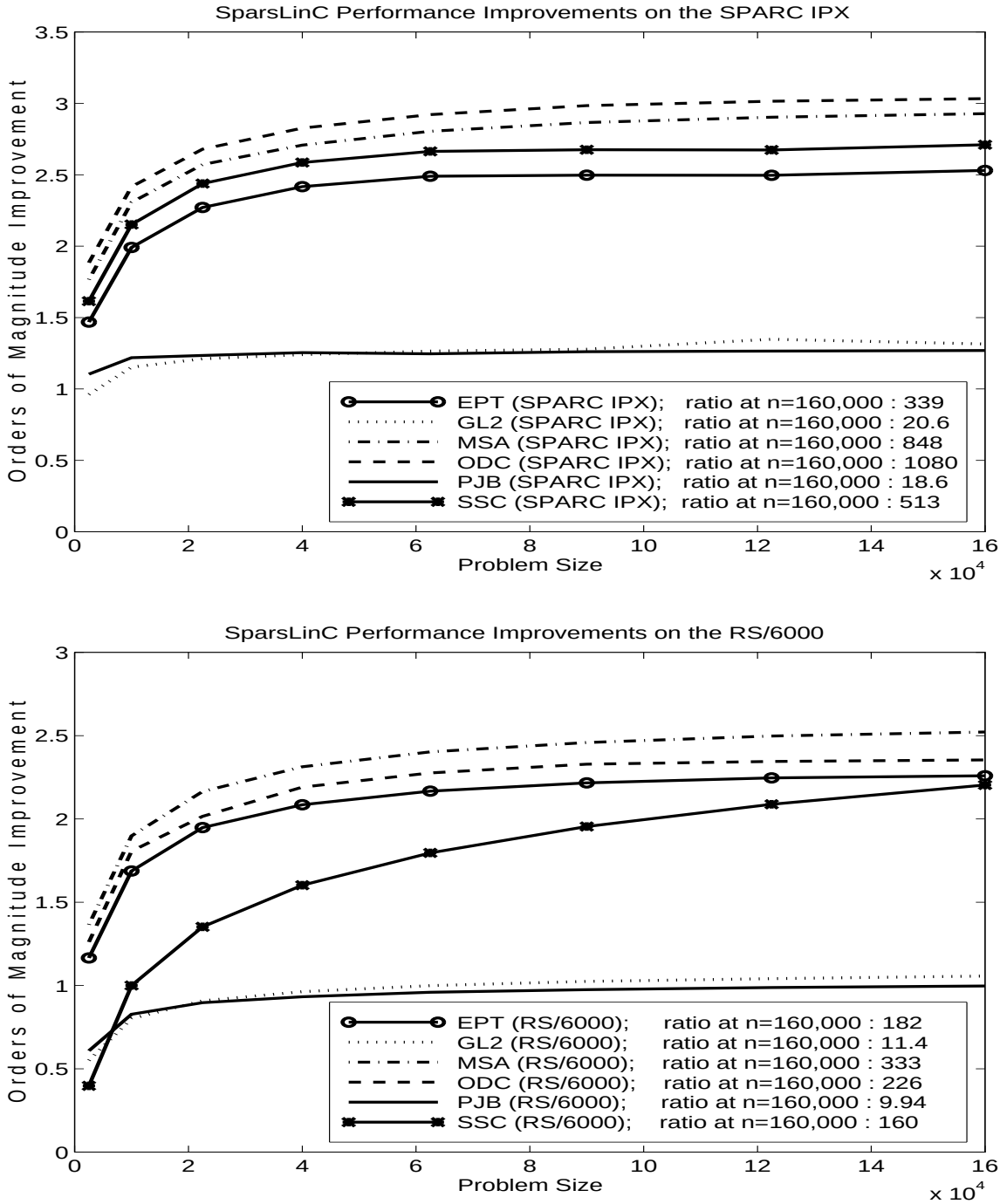

Figure 3.3: Logarithmic plot for each platform of `SPARSE` divided by `NONSPARSE` runtimes for all gradient problems.

Hence, vector operations execute only marginally faster, and memory locality (i.e., the reuse of data and the accessing of adjacent memory locations) is not much of an issue. In contrast, the RS6000 architecture employs a superscalar chip and a cache-based memory architecture. Hence, this machine performs better if executing short vector operations, since these operations can fill the short pipes and take advantage of memory locality. This is advantageous in the `NONSPARSE` computation where the VLCs are implemented via `DO`-loops. On the other hand, indirect addressing, used extensively in the SparsLinC algorithm, while fairly inconsequential on the SPARC, may lead to a performance degradation on the RS6000, as memory locality suffers.

Figure 3.3 is an alternative way of presenting the results in Figure 3.1. Here we have plotted for each machine and each problem the $\log_{10}$ of the ratio of `NONSPARSE` to `SPARSE` runtimes in order to demonstrate the orders of magnitude runtime improvements achieved by SparsLinC. Note that the line definition box also includes for each problem the true ratio of `NONSPARSE` to `SPARSE` runtimes for the case of $n = 160,000$.

For the largest problem size we note that the improvements in the gradient computation due to SparsLinC range from a factor of nearly 10 in the case of the PJB problem running on the RS6000, to a factor of greater than 1,000 for the ODC problem on the SPARC IPX. We also note that for large problem sizes the runtime efficiencies achieved by SparsLinC near a plateau. We suspect that this behavior is caused by the dense gradient computation increasingly dominating the overall computational task.

4 Computing Sparse Jacobians

For our sparse Jacobian experiments we selected five problems from the MINPACK-2 problem set of systems of nonlinear equations and one problem from the molecular distance geometry (MDG) class of global minimization functions. The MINPACK-2 problems are identified in Table 4.1. These are a standard set of test problems; the reader is referred to [1] for detailed descriptions.

The MDG problem is described in Moré and Wu [24] as follows: Given bond lengths $\delta_{i,j}$ between a subset $\mathcal{S}$ of the atom pairs, determine whether there is a molecule that satisfies these bond length constraints. Moré and Wu formulated this problem in terms of finding the global minimum of the function

$$f(x) = \sum_{i,j \in \mathcal{S}} w_{i,j} \left(\|x_i - x_j\|^2 - \delta_{i,j}^2 \right)^2, \tag{4.1}$$

where $w_{i,j}$ are positive weights, and x_i, $x_j \in \mathbb{R}^3$ are the positions of the i-th and j-th atoms in the molecule. Here, we reformulate the MDG problem in terms of finding the global minimum of the nonlinear least squares problem

$$\underset{x \in \mathbf{R}^n}{\text{minimize}} \, \|F(x)\|_2, \tag{4.2}$$

where $F(x) \in \mathbb{R}^m$, $n = 3 \times$ the number of atoms (3 coordinates per atom), $m =$ the number of $\delta_{i,j}$ terms that are specified in order for the problem to be deterministic ($m \geq n$), and for each $\delta_{i,j}$, the corresponding (k-th) component of $F(x)$ is defined by

$$F_k(x) = w_{i,j} \left(\|x_i - x_j\|^2 - \delta_{i,j}^2 \right)^2, \quad k \equiv (i,j), \quad k \in \mathcal{S}.$$

Table 4.1: MINPACK-2 Nonlinear Equations

Name	Description of the Problem
FDC	Flow in a Driven Cavity
FIC	Flow in a Channel
IER	Incompressible Elastic Rod
SFD	Swirling Flow between Disks
SFI	Solid Fuel Ignition

This transformation, while not altering the nature of the underlying computation or solution, poses the derivative problem as a sparse Jacobian computation.

We compared the ADIFOR/SparsLinC approach with an approach that uses a graph-coloring algorithm to compute the compressed Jacobian. In this section, we describe the compressed Jacobian approach. Then, we present the memory requirements and runtime results of both approaches for the MINPACK-2 and the MDG test problems.

4.1 The Compressed Jacobian Approach

The compressed Jacobian approach has been used as an effective strategy for computing sparse Jacobians in conjunction with both finite differencing [17] and automatic differentiation methods [2]. The prerequisite for applying the compressed Jacobian approach to a given sparse Jacobian computation is a priori knowledge of the sparsity pattern of the Jacobian.

The basic idea underlying this approach is that all the relevant (i.e., nonzero) information in the full Jacobian $f'(x)$ of size $m \times n$ can be represented in a compressed Jacobian $C(x)$ of size $m \times p$, where $C(x) = f'(x)S$, for some seed matrix S, and usually $p \ll n$. Essentially, S maps each group of *structurally orthogonal* columns of $f'(x)$ (i.e., columns that do not have a nonzero in the same row position) to a column of $C(x)$. Because of the structural orthogonality property we can uniquely extract all entries of the original Jacobian matrix from the compressed Jacobian [2].

If the sparsity pattern of the Jacobian can be determined, graph-coloring techniques can be used to arrive at S and p. These algorithms produce a partitioning of the columns of the Jacobian into p structurally orthogonal groups by coloring the column-intersection graph associated with the Jacobian. In our experiments we employ the graph-coloring software described in [16] to obtain S, and then compute $C(x)$ by initializing the ADIFOR-generated derivative code of $f(x)$ with the seed matrix set to S.

From the point of view of computational complexity, the consequence of the compressed Jacobian approach is clear. Specifically, the memory and runtime requirements of the Jacobian computation will approximately augment function requirements by a factor p rather than n. For many sparsity patterns, in particular regular grid problems, p is small and independent of n. Hence, the compressed Jacobian can be computed in a constant times the function computation time, regardless of n.

Naturally, the compressed Jacobian approach does introduce additional computational requirements, namely, (i) computing the sparsity pattern; (ii) performing the graph col-

oring; and (*iii*) subsequent to the computation of the compressed Jacobian, unraveling to arrive at the full Jacobian. However, often Jacobian computation is a subproblem within an optimization problem, and in an optimization algorithm we invariably need to compute a sequence $\{f'(x_k)\}$ of Jacobians for some sequence $\{x_k\}$ of iterates. In most cases we need to perform (*i*) and (*ii*) only once, since we can specify the *closure* of the sparsity patterns (i.e., a sparsity pattern that, for every iterate x_k, contains the sparsity pattern of $\{f'(x_k)\}$). If we are not able to specify the closure, the compressed Jacobian approach requires a call to the graph-coloring software at each iteration. Finally, the computational requirements of (*iii*) are relatively small.

The interesting difference between the MINPACK-2 and the MDG Jacobians is that for each MINPACK-2 problem the chromatic number is independent of n, whereas the chromatic number for the MDG Jacobians grows as a function of n. This difference is due to the "regularity" of the sparsity structures of the MINPACK-2 Jacobians as we change n, and the lack of the same in the case of the MDG Jacobians.

Figure 4.1 shows the sparsity patterns for some SFD and MDG Jacobians (here each dot represents the occurrence of a nonzero in the final Jacobian). In the case of each problem, we have chosen two small problem sizes in order to visually compare their respective sparsity structures. For the MDG problem of size $n = 81$, we have additionally plotted the upper left-hand corner of the Jacobian, since it is difficult to distinguish the location of the nonzeroes in the full Jacobian.

In comparing the sparsity patterns of the two SFD problems, we note that an identically shaped nonzero block structure appears repeatedly along the main diagonal as n is increased. The implication of this regularity of structure for the graph-coloring algorithm is that a fixed number of colors are sufficient for the representation of the compressed Jacobian, that is, p remains constant for all n. Though the algorithms used in graph coloring are based on partitioning methods, one can visualize the results as each block in the full Jacobian sliding left to occupy the corresponding set of rows in the p columns of the compressed Jacobian. Though distinct in shape, the sparsity patterns of the remaining MINPACK-2 Jacobians also have this regularly repeating feature, since they are all regular grid problems.

In contrast to the MINPACK-2 problems, for the two MDG Jacobians we note that as n grows, the shape of the structures in the Jacobian sparsity pattern changes. This is clearly apparent when comparing the sparsity pattern of the MDG Jacobian for $n = 24$ with the corresponding slice of the MDG Jacobian for $n = 81$. The reason for this variation has to do with the problem specification itself. The formulation of the MDG problem is such that commensurate with increasing n, the number of atoms in the molecule is increased. In order for the problem to remain deterministic, corresponding to the increase in n, m (i.e., the number of distances between the atoms ($\delta_{i,j}$'s) which must be specified) also increases. The net effect is the structural change evident in the Jacobian sparsity, which in turn leads to a different chromatic number being computed by the coloring algorithm for each n.

In Sections 4.2 and 4.3 we will compare the performance of the ADIFOR/SparsLinC approach with that of the ADIFOR/compressed Jacobian approach when the chromatic number, p, is independent of n and when it is not, respectively. We refer to the ADIFOR/SparsLinC approach as the `SPARSE` approach in the sense that the complete (sparse) Jacobian is computed, and we refer to the `NONSPARSE` ADIFOR/compressed Jacobian approach simply as the `COMPRESSED` approach.

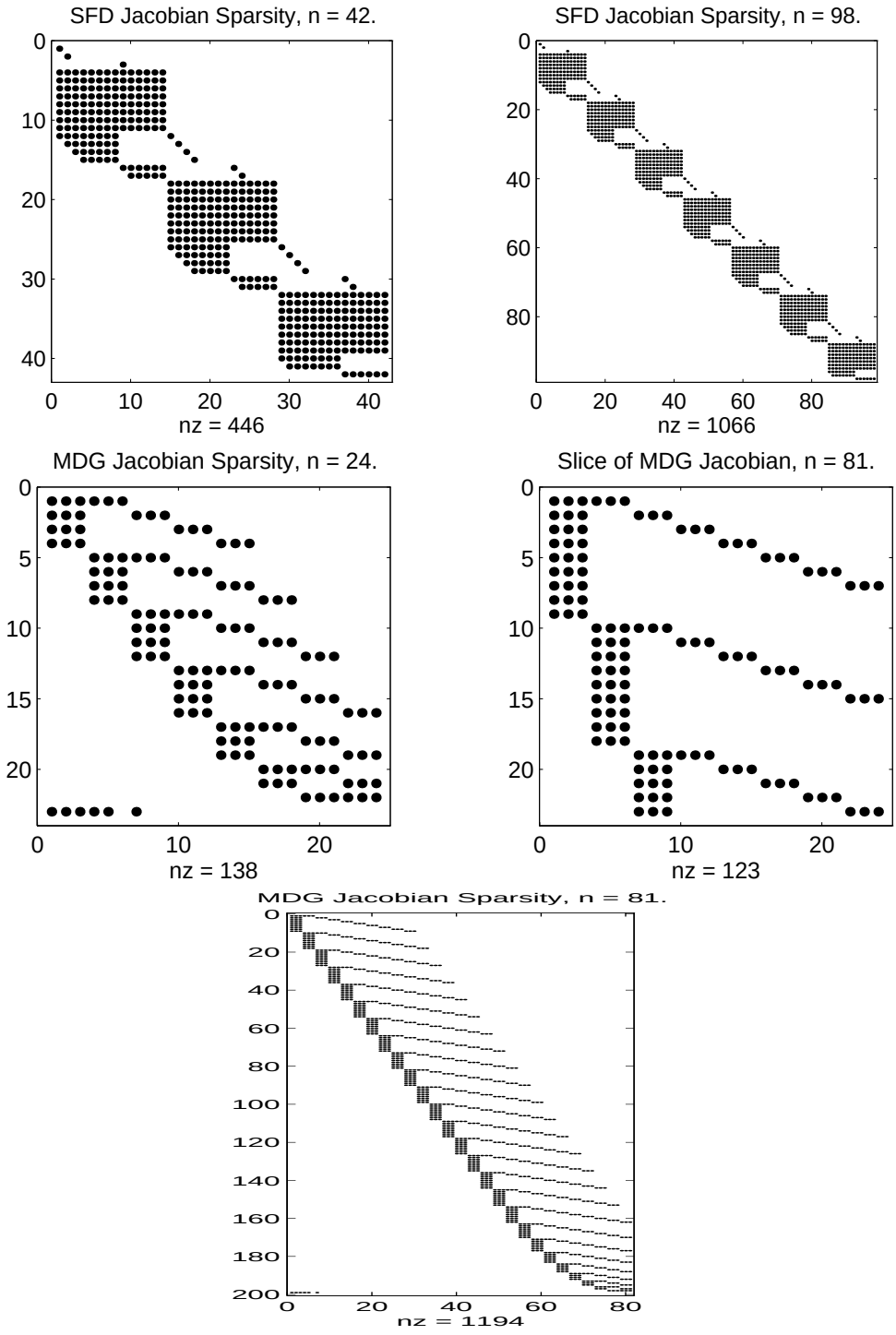

Figure 4.1: SFD Jacobian sparsity patterns for $n = 42$, and $n = 98$, and MDG Jacobian sparsity patterns for $n = 24$, and $n = 81$.

Table 4.2: Structural Information about the MINPACK-2 Jacobians

Problem	nnz($f'(x)$)	p	density of $C(x)$
FDC	13 m	19	68%
FIC	8 m	9	89%
IER	11 m	17	65%
SFD	12 m	14	86%
SFI	5 m	7	71%

For our purposes of comparing the results of these two functionally equivalent approaches, we find it useful to think of the `COMPRESSED` approach as the *static* approach to exploiting sparsity in AD, since it statically allocates arrays of length p for the storage of the directional gradient vectors, and since it requires a priori knowledge of the sparsity pattern of the Jacobian. Conversely, the `SPARSE` approach can be thought of as the *dynamic* approach to exploiting sparsity in AD, since it uses dynamic memory allocation for the storage of the directional gradient vectors, and since the sparsity pattern is computed as a by-product during the computation of the Jacobian.

When p is independent of n and is relatively small, as in the case of the MINPACK-2 problems, two issues influence whether the static or dynamic approach is preferable. The first is the issue of convenience. The dynamic approach does not require knowledge of the closure of the sparsity patterns; hence, from the view point of code development, it is the more convenient approach. Note that if it is not possible to compute the sparsity closure, it may be the only approach.

The second issue concerns the sparsity of the compressed Jacobian, which turns out to be highly correlated with the relative performance of the `SPARSE` computation as compared with the `COMPRESSED` one. Consider the case where the compressed Jacobian is fully dense. Then the static approach is ideal, since it contains no extraneous zero-sum computations nor does it allocate extraneous storage. Though by design the dynamic approach also avoids extraneous computations and storage, it is burdened by overheads associated with dynamic memory allocation and the maintenance of dynamic data structures. On the other hand, if the compressed Jacobian is sparse, the dynamic approach becomes preferable, since its overheads are offset by the many extraneous computations and unused memory in the static approach.

Tables 4.2 and 4.3 contain information relating to the structure of the MINPACK-2 and the MDG Jacobian problems, respectively. In the case of the MINPACK-2 problems, $m = n$ and p remains constant. Hence, in Table 4.2 we have shown for each problem the number of nonzeroes in the Jacobian, nnz($f'(x)$), as well as the chromatic number, p, computed by the graph-coloring software. As previously discussed, p is independent of n for every MINPACK-2 problem because the sparsity structure of the corresponding Jacobian is also independent of n. On the other hand, for the MDG problem the Jacobian structure is distinct for each problem size n. Hence, in Table 4.3 we have shown the corresponding n, m, and p values. For all n, the number of nonzeroes in the MDG Jacobian is given by

Table 4.3: Structural Information about the Molecular Distance Geometry Jacobians

n	m	p	density of $C(x)$
375	2801	78	8%
525	4051	78	8%
648	7111	111	5%
1029	15583	150	4%
1536	30689	195	3%
2187	55729	246	2%
3000	94951	303	2%
3993	153671	366	2%
5184	238393	435	1%
6591	356929	510	1%
8232	518519	591	1%
10125	733951	678	1%

$\text{nnz}(f'(x))= 6m$.

The last column in both Tables 4.2 and 4.3 show the density of $C(x)$, computed as $\text{nnz}(f'(x))/mp$. In Sections 4.2 and 4.3, we will show the correlation of these densities with the memory requirements and runtime performances of the two approaches.

4.2 Computing Sparse Jacobians When p Is Independent of n

4.2.1 Memory Requirements

Table 4.4 presents the total memory requirements of the two approaches for computing the Jacobian of the MINPACK-2 problems for the case of $n = 160{,}000$. In addition to the notation introduced in Table 3.2, we use $M\{J\}$ to denote the memory requirements of the `COMPRESSED` approach, which includes the memory needed for the graph-coloring computation, and $M\{J_{Sparse}\}$ to denote the memory requirements of the `SPARSE` approach. For each problem, the ratio $M\{J\}/M\{F\}$ remains constant for all n. Hence, Table 4.4 is a sufficient summary of all the memory results.

As mentioned in Section 3.1, the `NONSPARSE` mode of ADIFOR tends to augment memory requirements of the function computation linearly. In the case of the compressed Jacobian computations the augmentation factor is the chromatic number, p. However, since the graph-coloring algorithm also introduces additional memory requirements proportional to $\text{nnz}(f'(x))$, the $M\{J\}/M\{F\}$ ratios in Table 4.4 are larger than the corresponding p values in Table 4.2, and we have

$$M\{J\} \leq \kappa \; p \; M\{F\} \tag{4.3}$$

with $1.5 \leq \kappa \leq 1.9$. In general, κ is largest for the problems with denser compressed Jacobians. For example, for the FIC problem, the compressed Jacobian is 89% dense and $\kappa = 1.9$, whereas for the IER problem, the compressed Jacobian is 65% dense and $\kappa = 1.5$.

Table 4.4: Memory Requirements for MINPACK-2 Jacobians (in Mbytes; $n = 160,000$)

Problem	$M\{F\}$	$M\{J\}$	$M\{J\}/M\{F\}$	$M\{J_{Sparse}\}$	$M\{J_{Sparse}\}/M\{F\}$
FDC	2.57	74.90	29.1	54.35	21.1
FIC	2.58	43.05	16.7	32.17	12.5
IER	2.57	67.22	26.1	50.65	19.7
SDF	2.58	60.84	23.6	41.77	16.2
SFI	2.57	33.94	13.2	28.35	11.1

Given that the memory requirements of the `SPARSE` approach are based on the need to represent the nonzero information in the derivative computation, and that this information is close in volume to the size of the compressed Jacobian (for the MINPACK-2 problems, within $65\% - 89\%$), we expect the values of $M\{J_{Sparse}\}/M\{F\}$ to be somewhat correlated with the corresponding p for each problem. A comparison of the last column of Table 4.4 and the corresponding values of p shows this to be the case, where we have

$$M\{J_{Sparse}\} \leq \sigma \; p \; M\{F\} \tag{4.4}$$

with $1.1 \leq \sigma \leq 1.6$.

We note that the memory requirements of the `COMPRESSED` approach are $19\% - 46\%$ greater than the memory requirements of the `SPARSE` approach. This is somewhat surprising, since in all cases the compressed Jacobians are fairly dense, and one might suspect that a static approach would be more parsimonious in terms of memory usage. The disparity is mainly attributable to the need for the additional memory required by the graph-coloring algorithm in the `COMPRESSED` approach, but also to the terseness of SparsLinC's memory allocation scheme, and the fact that SparsLinC exploits intermediate sparsity in derivative computations.

Inequality (4.4) is a useful metric in terms of comparing the memory requirements of the `SPARSE` with the `COMPRESSED` approach. However, SparsLinC has no knowledge of p, which can be thought of as a global measure of sparsity. SparsLinC's representation of directional gradient vectors is based on very localized information, namely, the number of nonzeroes in each gradient vector. The most striking example demonstrating the difference between these global and local approaches toward storage would be the computation of a sparse Jacobian with one fully dense row and a dense diagonal. In this case $p = n$; hence, the compressed Jacobian approach would fail entirely. On the other hand, SparsLinC would need to maintain only one vector of length n, and $n - 1$ vectors of length 1, resulting in much less memory-intensive (as well as faster) code.

Hence, it would be more descriptive of the localized vector representations in SparsLinC to express the augmentation factor of the memory requirements of the `SPARSE` approach with respect to those of the function, as proportional to the average number of nonzeroes per row of the Jacobian,

$$M\{J_{Sparse}\} \leq \sigma' \; \frac{\text{nnz}(f'(x))}{m} \; M\{F\}, \tag{4.5}$$

Table 4.5: Runtime Ratios on the SPARC IPX ($n = 160,000$)

Problem	Coloring	COMPRESSED	SPARSE	SPARSE/COMPRESSED
FDC	19.4	15.9	52.7	3.3
FIC	6.7	6.9	32.3	4.7
IER	20.4	10.7	24.5	2.3
SFD	9.0	8.3	25.2	3.0
SFI	29.2	16.4	41.5	2.5

Table 4.6: Runtime Ratios on the RS6000 ($n = 160,000$)

Problem	Coloring	COMPRESSED	SPARSE	SPARSE/COMPRESSED
FDC	31.5	11.5	109.0	9.5
FIC	30.2	7.8	161.0	20.7
IER	69.9	8.8	86.2	9.8
SFD	54.7	10.9	156.0	14.3
SFI	47.1	7.4	83.2	11.2

where for the MINPACK-2 problems, $1.4 \leq \sigma' \leq 2.2$. As we shall see in Section 4.3.1, inequality (4.5) is a more reliable generalization of the expected memory requirements of the SPARSE approach.

4.2.2 Runtime

Figures 4.2 and 4.3 summarize the runtime results of the Jacobian problems on the SPARC IPX and RS6000, respectively. As in the case of the gradient experiments, the problem sizes range from $n = 2,500$ to $n = 160,000$. Each panel in these figures contains a plot of the SPARSE Jacobian-to-function runtime ratios. Additionally, the results from the COMPRESSED computation are shown with two plots in each panel, one showing the coloring-to-function runtime ratios, and the other showing the compressed Jacobian-to-function runtime ratios. As we explained in Section 4.1, for most optimization algorithms, the coloring computation is performed only once, whereas iterates of the compressed Jacobian are computed repeatedly, hence the separation of the two computations in our plots. In addition to plotting these runtime ratios, we report their values for the case of $n = 160,000$ in Tables 4.5 and 4.6.

A main feature of these results is that for both approaches the runtime of the Jacobian computation is independent of n, as shown by the constant ratio plots. In the case of the COMPRESSED approach, this behavior is expected, since we know the Jacobian runtime will be approximately equal to the linear augmentation of the function runtime by a factor p, and as we have stated, p is independent of n. Based on the COMPRESSED ratios in Tables 4.5 and 4.6, we can state that

$$T\{J_{Compressed}\} \leq \lambda \; p \; T\{F\}, \tag{4.6}$$

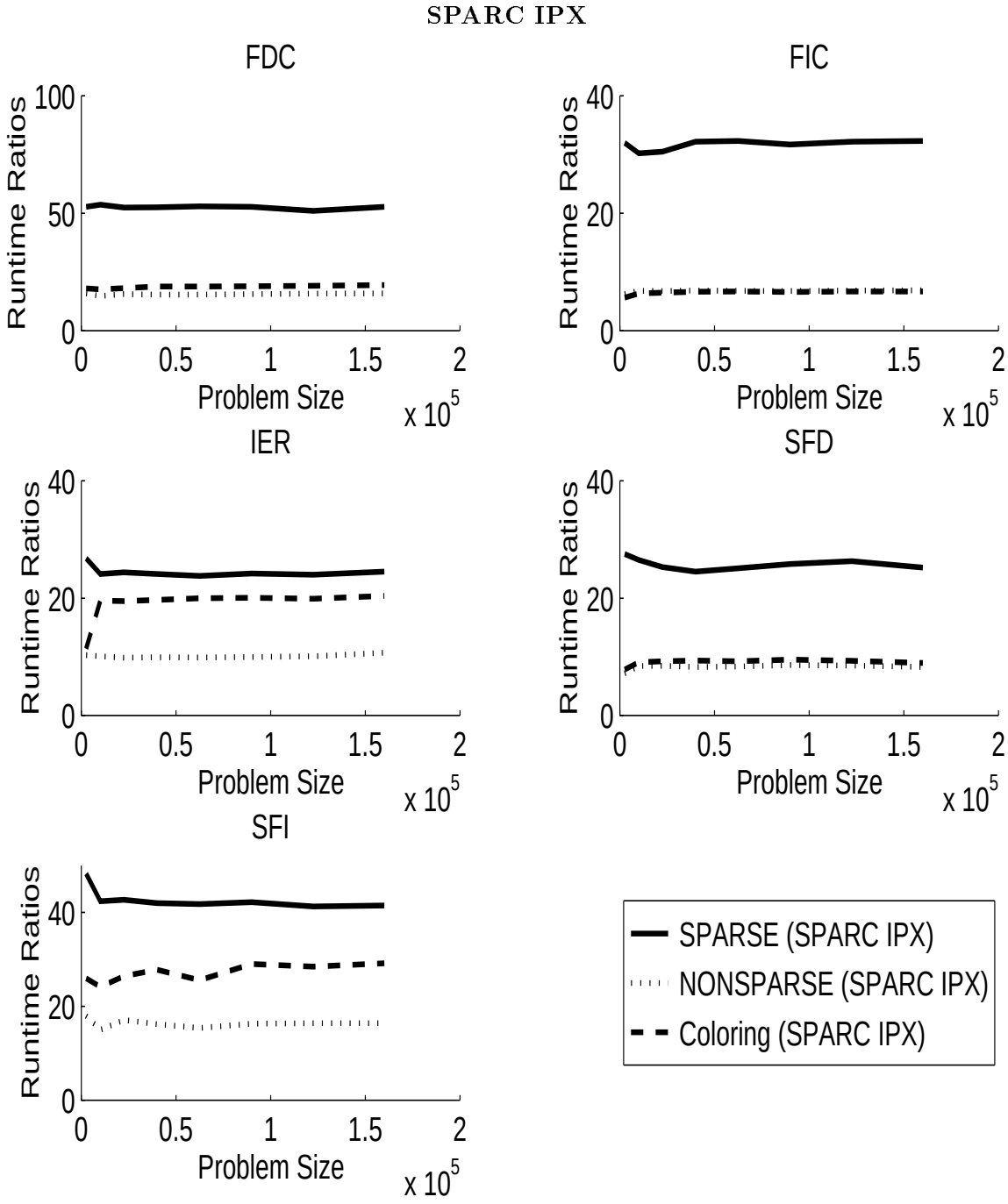

Figure 4.2: Runtime ratios of MINPACK-2 Jacobian problems on the SPARC IPX. For each problem, `COMPRESSED` computation-to-function, `SPARSE` computation-to-function and coloring-to-function runtime ratios are shown for problem sizes $n = 2,500$ to $n = 160,000$.

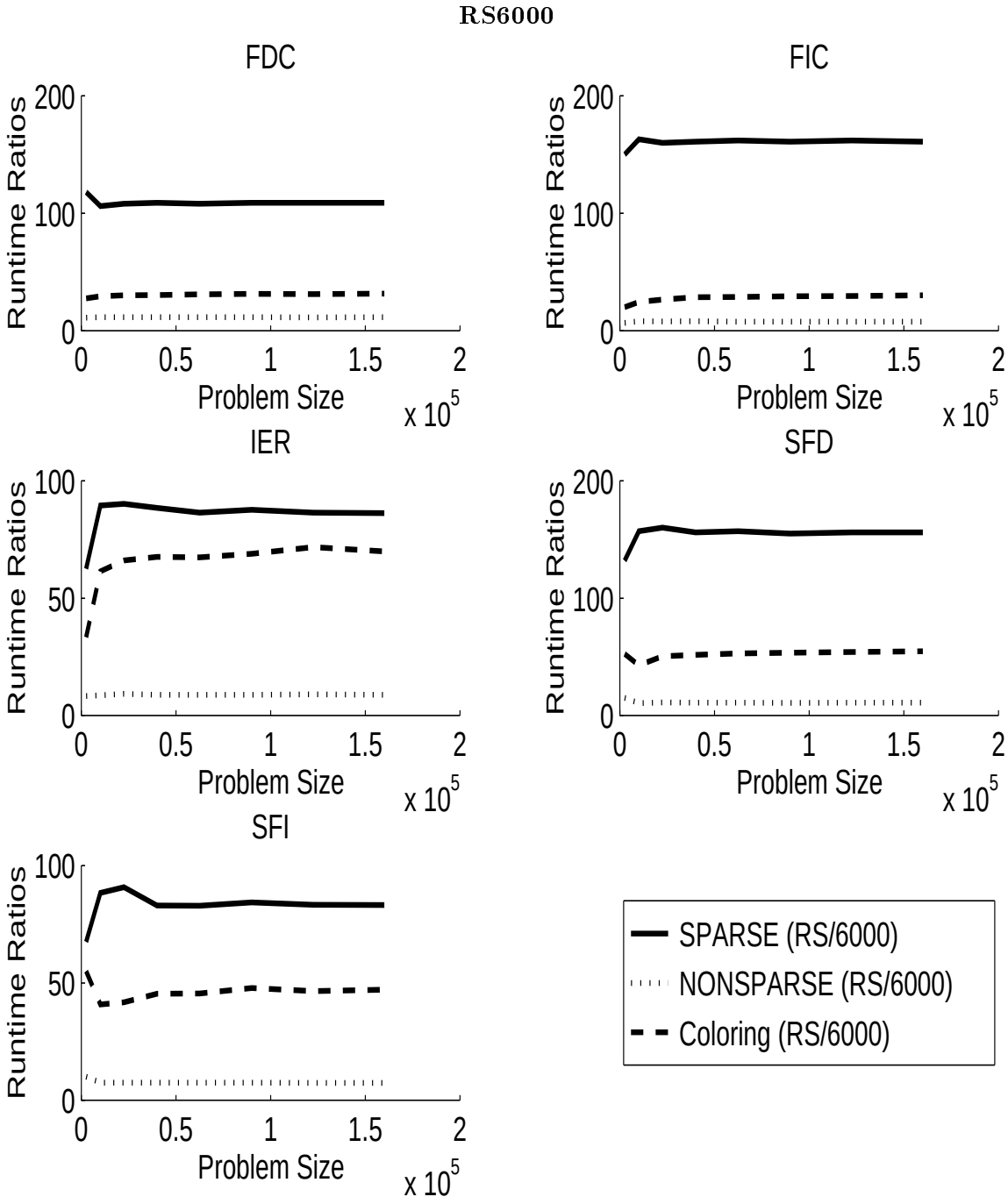

Figure 4.3: Runtime ratios of MINPACK-2 Jacobian problems on the RS6000. For each problem, `COMPRESSED` computation-to-funtion, `SPARSE` computation-to-function and coloring-to-function runtime ratios are shown for problem sizes $n = 2,500$ to $n = 160,000$.

where $T\{.\}$ denotes the runtime of each computation and where $0.6 \leq \lambda \leq 2.4$ on the SPARC IPX and $0.5 \leq \lambda \leq 1.1$ on the RS6000. The variation in λ values for the different problems is due to the peculiarities of each code, in particular the extent to which ADIFOR can minimize the computations performed in the derivative code through activity analysis. The variation in λ values between the two platforms is due to architectural differences reviewed in Section 3.2, though it is surprising to see that the RS6000 ratios are slightly greater for the FIC and SFD problems. We note that in all cases λ is a small constant, which matches our expectation.

For the `SPARSE` Jacobian computation, the constant behavior with respect to function runtime is due to the sparsity present in all the intermediate directional gradient vectors. Unlike the `SPARSE` gradient computation, where at least one directional gradient vector (i.e., the final gradient vector) grows to be fully dense, in the Jacobian computation all intermediate vectors will be as sparse or more sparse than the least sparse row of the Jacobian. From the results of the coloring algorithm, we know that this row can have no more than p nonzeroes (otherwise the compressed Jacobian would not "fit" into p columns). In terms of an augmentation over function time, we can state that

$$T\{J_{Sparse}\} \leq \omega \; p \; T\{F\}, \tag{4.7}$$

where $1.4 \leq \omega \leq 5.9$ on the SPARC IPX and $5.0 \leq \omega \leq 17.9$ on the RS6000. Similar to the case of the memory requirements of `SPARSE` approach, we can express the augmentation of the `SPARSE` runtime with respect to the function runtime as a factor of the average number of nonzeroes per row of the Jacobian:

$$T\{J_{Sparse}\} \leq \omega' \; \frac{\text{nnz}(f'(x))}{m} \; T\{F\}, \tag{4.8}$$

where $2.1 \leq \omega' \leq 8.3$ on the SPARC IPX and $7.8 \leq \omega' \leq 20.1$ on the RS6000. We will revisit inequality 4.8 in Section 4.3.2.

The last column in Tables 4.5 and 4.6 show the ratios of `SPARSE` runtime to `COMPRESSED` runtime for each problem on each platform. These ratios—all of which are above 1—represent the penalty paid for the convenience of the dynamic approach. As expected, there is a strong correlation between these ratios and the sparsity of the corresponding compressed Jacobian. For example, on both platforms the highest ratios occur for the FIC problem, which has the least sparse compressed Jacobian. Conversely, the ratios for the IER problem, which has the most sparse compressed Jacobian, are among the smallest. In general, the sparser the compressed Jacobian, the better the runtime performance of the `SPARSE` approach relative to that of the `COMPRESSED` approach.

The coloring algorithm also demonstrates constant runtime behavior with respect to function runtime, with the constant factor usually being somewhere between the corresponding factors of the `COMPRESSED` and `SPARSE` computations. The closure of the sparsity patterns of each MINPACK-2 Jacobian problem is known; hence, the impact of the one-time graph-coloring computation on the overall optimization computation would be minimal. Had the closure been unknown, the coloring computation would need to be repeated at each iteration, with the possible consequence of the `SPARSE` approach having a faster runtime. For example, in Table 4.5 note that for the IER and SFI problems on the SPARC IPX, the sum of the runtime ratios of the coloring and `COMPRESSED` compressed Jacobian computations exceeds the corresponding ratio from the `SPARSE` computations.

4.3 Computing Sparse Jacobians When p Grows with n

4.3.1 Memory Requirements

Table 4.7 presents the total memory requirements of the two approaches for computing the Jacobians of the MDG problem for various problem sizes (we use the same notation introduced in Table 4.4). In contrast to Table 4.4 where the memory requirements were shown only for the largest-sized problem ($n = 160,000$), Table 4.7 shows the memory requirements for all problem sizes, ranging from $n = 375$ to $n = 10,125$. This is done for two reasons. First, since now p grows with n, so too does the ratio $M\{J\}/M\{F\}$, in contrast to the constant $M\{J\}/M\{F\}$ in the case of the MINPACK-2 Jacobians where p was constant.

Second, largely because of the growing value of p, the executable for the largest-sized problems became too large to fit in the virtual memory available on either platform. Hence, we resorted to the stripmining approach (see Section 3.1) for the `COMPRESSED` computation, which dramatically reduced memory requirements, while slightly increasing the runtime. We used strip size $p_s = 16$ for the problems of dimension $n = 3,000$ to $n = 8,232$. However, this choice of p_s for $n = 10,125$ led to performance degradation as a result of paging. Hence, we used $p_s = 8$ for the largest problem size (we discuss our choice of strip sizes further in Section 4.3.2). In Table 4.7, the memory requirements of the stripmined `COMPRESSED` computations are marked with $*$ for $p_s = 16$ and $**$ for $p_s = 8$.

The results in Table 4.7 show that for the non-stripmined `COMPRESSED` computation of the MDG Jacobians, the inequality (4.3) holds as it did in the case of the MINPACK-2 Jacobians, with the difference that here p is not constant. Nonetheless, we can compute κ in (4.3) for the MDG problem and ascertain that, as expected, its value varies only slightly ($0.35 \leq \kappa \leq 0.41$) for the problem sizes $n = 375$ to $n = 2,187$. We note that here κ is a factor 3–5 smaller than in the MINPACK-2 cases. This is mainly due to the presence of several large work arrays of dimension m in the MDG function code, which are unrelated to any derivative computation; hence, ADIFOR does not generate corresponding augmented derivative work arrays.

For the stripmined `COMPRESSED` computation of the MDG Jacobians, we need a new inequality, since now the linear augmentation is dependent on the strip size and not the chromatic number:

$$M\{J_{Stripmined}\} \leq \kappa' \; p_s \; M\{F\} \tag{4.9}$$

where $\kappa' = 0.54$ for all problem sizes for which $p_s = 16$, and $\kappa' = 0.74$ when $p_s = 16$. This strict dependance of κ' on p_s implies that the memory requirement penalty of the stripmining method decreases with respect to the function memory requirements as the strip size increases. By virtue of the same reasoning, we can explain why $\kappa < \kappa'$, since essentially the non-stripmined computation can be thought of as a stripmined computation with one strip of size $p_s = p$.

We also observe from Table 4.7 that the $M\{J_{Sparse}\}/M\{F\}$ ratio for the `SPARSE` approach varies only slightly for different problem sizes. As noted in Section 4.1, $\text{nnz}(f'(x))= 6m$ for the MDG problem; hence the average number of nonzeroes per row of the Jacobian is always 6, for all n. Consequently, inequality (4.5) holds, and the corresponding range of σ' values for the MDG problem is given by, $1.5 \leq \sigma' \leq 1.7$.

Table 4.7: Memory Requirements for the Distance Geometry Jacobians (in Mbytes)

n	$M\{F\}$	$M\{J\}$	$M\{J\}/M\{F\}$	$M\{J_{Sparse}\}$	$M\{J_{Sparse}\}/M\{F\}$
No Stripmining					
375	0.07	2.22	31.7	0.70	10.0
525	0.10	3.19	31.9	0.98	9.8
648	0.18	7.46	41.4	1.65	9.2
1029	0.38	21.16	55.7	3.49	9.2
1536	0.75	52.66	70.2	6.77	9.0
2187	1.36	118.29	87.0	12.22	9.0
Stripmined ($p_s = 16$)					
3000	2.30	19.88*	8.6	20.71	9.0
3993	3.72	32.00*	8.6	33.47	9.0
5184	5.76	49.47*	8.6	51.85	9.0
6591	8.62	73.88*	8.6	77.48	9.0
8232	12.51	107.11*	8.6	112.16	9.0
Stripmined ($p_s = 8$)					
10125	17.70	103.76**	5.9	158.47	9.0

In summary, the memory requirements of the `COMPRESSED` approach are greater than those of the `SPARSE` approach by a large factor, ranging from 3.2 to 9.7 for the non-stripmined problems. This result is due to the compressed Jacobians of the MDG problems being very sparse, and more and more so as n grows. Stripmining can be a very effective method of reducing the memory requirements of the `COMPRESSED` approach—and, as we shall see, at the cost of a modest runtime penalty. Finally, we note that though we did not need to stripmine the `SPARSE` computation, this method is equally possible and straightforward.

4.3.2 Runtime

Figures 4.4 and 4.5 show the Jacobian-to-function runtime ratios for the MDG problems on the SPARC IPX and RS6000, respectively, with sizes $n = 375$ to $n = 10125$. Each figure includes three plots showing the ratios of the `COMPRESSED` (dotted), `SPARSE` (solid), and graph-coloring (dashed) runtimes to the function runtimes. We also present the detailed Jacobian-to-function and coloring-to-function runtime ratios on the SPARC IPX and the RS6000 in Tables 4.8 and 4.9, respectively. Note that, as before, the runtime ratios of a stripmined `COMPRESSED` computation are marked with $*$ for $p_s = 16$ and $**$ for $p_s = 8$.

The main conclusion that can be drawn from the plots in Figures 4.4 and 4.5 is that the the Jacobian-to-function runtime ratios appear to grow linearly with n for the `COMPRESSED` approach, whereas these ratios are independent of n for the `SPARSE` approach. Hence, the `SPARSE` approach is clearly the method of choice for problems where the chromatic number grows as a function of the problem size and yet the sparsity of the corresponding Jacobians remains constant (i.e., $\mathrm{nnz}(f'(x))$ is proportional to m). The efficient runtime performance

Table 4.8: Runtime Ratios of the MDG Problem on the SPARC IPX

n	Coloring	COMPRESSED	SPARSE	SPARSE/COMPRESSED
No Stripmining				
375	31.6	113	124	1.1
525	31.3	121	128	1.1
648	31.7	187	124	0.7
1029	32.0	222	126	0.6
1536	38.9	332	127	0.4
2187	39.2	420	123	0.3
Stripmined ($p_s = 16$)				
3000	38.9	431*	121	0.3
3993	44.71	557*	117	0.2
5184	46.70	645*	132	0.2
6591	50.91	782*	108	0.1
8232	49.60	857*	122	0.1
Stripmined ($p_s = 8$)				
10125	54.5	1233**	123	0.1

Table 4.9: Runtime Ratios of the MDG Problem on the RS6000

n	Coloring	COMPRESSED	SPARSE	SPARSE/COMPRESSED
No Stripmining				
375	66.7	108	257	2.4
525	66.7	111	289	2.6
648	73.3	160	287	1.8
1029	73.5	213	285	1.3
1536	75.0	281	288	1.0
2187	78.2	378	293	0.8
Stripmined ($p_s = 16$)				
3000	84.7	396*	289	0.7
3993	92.3	472*	289	0.6
5184	100.0	579*	285	0.5
6591	112.1	662*	291	0.4
8232	122.6	777*	294	0.4
Stripmined ($p_s = 8$)				
10125	123.5	1022**	286	0.3

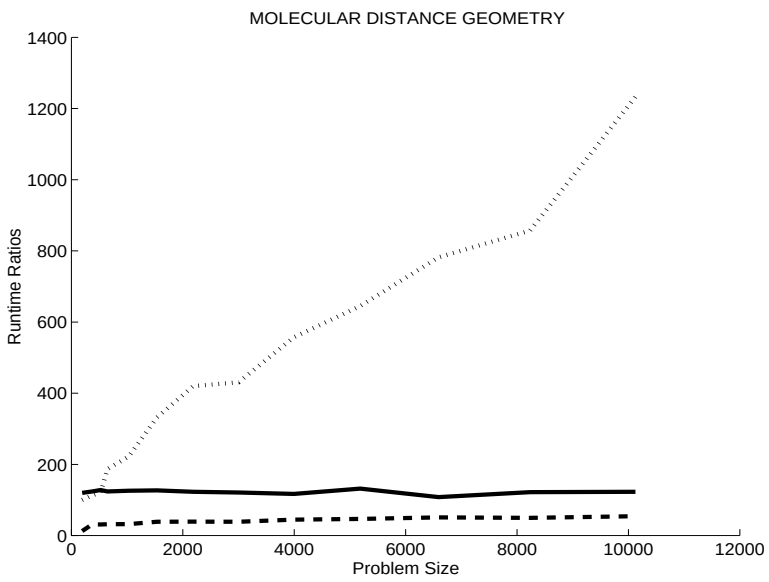

Figure 4.4: Jacobian-to-function runtime ratios on the SPARC IPX. `COMPRESSED` (dotted), `SPARSE` (solid), and graph-coloring (dashed).

of the `SPARSE` approach relative to the `COMPRESSED` approach is attributed to the fact that the already very sparse compressed Jacobians get even sparser as we increase the value of n (see the last column of Table 4.3).

In contrasting the numbers in Tables 4.8 and 4.9, we note that the previously discussed architectural effects (see Section 4.2.2) once again favor the `SPARSE` approach on the SPARC IPX and the `COMPRESSED` approach on the RS6000. In particular, the crossover point at which the `SPARSE` computation runs faster than the `COMPRESSED` computation occurs at $n = 525$ on the SPARC IPX and at $n = 2,187$ on the RS6000. Also, for the largest problem, the speedup of the `SPARSE` versus the `COMPRESSED` computation is 10 on the SPARC IPX and 3.5 on the RS6000.

We observe that for both the straightforward and the stripmined `COMPRESSED` computations, the inequality (4.6) holds; however, in contrast to the MINPACK-2 problems, p is not a constant. For the MDG problem, we have $1.4 \leq \lambda \leq 1.8$ on the SPARC IPX and $1.3 \leq \lambda \leq 1.5$ on the RS6000. Likewise, for the `SPARSE` computation inequality (4.8) holds with $18 \leq \omega' \leq 22$ on the SPARC IPX and $43 \leq \omega' \leq 49$ on the RS6000.

The strip sizes were chosen such that they would result in reasonable memory values. As was previously explained, stripmining is needed in order for the `COMPRESSED` computation to fit in memory, and comes at the cost of an additional (extraneous) function evaluation per strip. In terms of the effect of stripmining on the runtime of the `COMPRESSED` computation, we note that the `COMPRESSED` plots in Figures 4.4 and 4.5 appear to have three distinct slopes corresponding to the three computational flavors: (i) straightforward or non-stripmined, (ii) stripmined with $p_s = 16$, and (iii) stripmined with $p_s = 8$. As expected, the slope of the

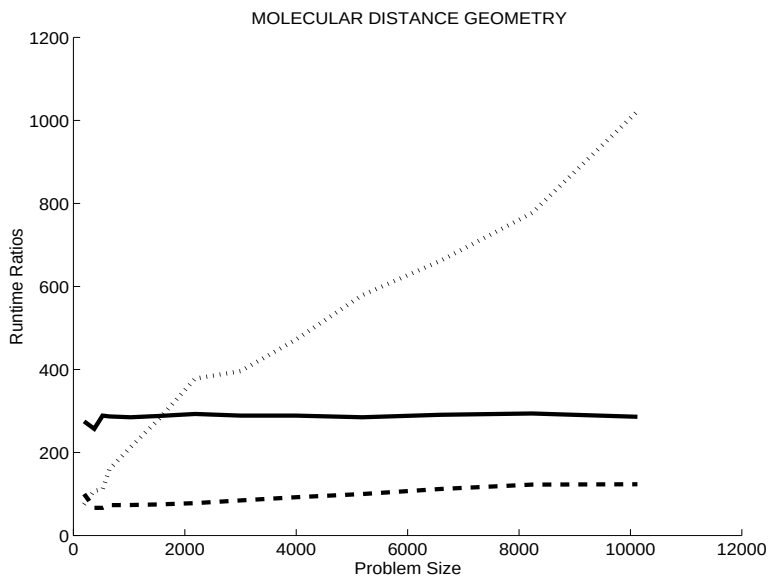

Figure 4.5: Jacobian-to-function runtime ratios on the RS6000. `COMPRESSED` (dotted), `SPARSE` (solid), and graph-coloring (dashed).

line segment corresponding to (iii) is greater than that of (ii), because halving the strip size, doubles the overhead function computations. By virtue of the same overhead effects, one might expect the slope of the line segment corresponding to (ii) to be greater than that of (i). However, the contrary is true. We suspect that this fact is due to the degradation in runtime performance of the larger non-stripmined problems brought about by excessive paging.

Finally, we note in Figures 4.4 and 4.5 that the coloring-to-function ratio results—in like manner to their the MINPACK-2 counterparts—are constant, with the constant factor being smaller than the corresponding factor for the `SPARSE` computation.

5 Conclusions

We have shown the effectiveness of the SparsLinC library in conjunction with the ADIFOR tool for exploiting sparsity in automatic differentiation. The fact that SparsLinC does not require a priori knowledge about the structure of sparsity in a particular problem, coupled with the fact that sparsity is exploited transparently, without the need for rewriting code, makes the ADIFOR/SparsLinC combination a valuable and convenient tool set. It can be used either for quick prototyping of the sparsity characteristics of a problem or as a means of getting improved efficiency in sparse derivative computations at very little cost in terms of program development time.

The ADIFOR/SparsLinC approach reduces the runtime cost of computing gradients of partially separable functions by as many as three orders of magnitude over the `NONSPARSE`

ADIFOR approach, while imposing modest memory requirements. SparsLinC does not require a priori knowledge about where and how the partial separability is coded; if it exists, SparsLinC will exploit the consequent sparsity.

The relative efficiency of SparsLinC's runtime results in computing sparse Jacobians as compared with the performance of the compressed Jacobians method is dependent on whether or not the chromatic number, p, derived from graph-coloring of the Jacobian is dependent on the problem size, n. (This comparison presupposes the feasibility of a priori computation of the closure sparsity pattern of the Jacobian needed for the compressed Jacobian approach.) If p is independent of n, both the ADIFOR/SparsLinC approach and the ADIFOR/compressed Jacobians approach will run at a constant factor times the function runtime, with the compressed Jacobian approach typically being the faster of the two.

If p grows with n, the compressed Jacobians approach will have a linear dependence on n, which implies much higher runtimes and memory requirements, making ADIFOR/SparsLinC the clear method of choice in such cases.

Acknowledgments

We thank Jorge Moré for his assistance with the MINPACK-2 problem set, Paul Hovland and Aaron Ross for their contributions to building early prototypes of SparsLinC, Andrew Mauer and Tim Knauff for their assistance in debugging SparsLinC, Vitaly Shmatikov for his assistance with the sparsity characteristics plots, and Zhijun Wu for his assistance with the MDG problem.

References

[1] B. M. Averick, R. G. Carter, J. J. Moré, and G. L. Xue. The MINPACK-2 test problem collection. Technical Report ANL/MCS-TM-150, Rev. 1, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[2] Brett Averick, Jorge Moré, Christian Bischof, Alan Carle, and Andreas Griewank. Computing large sparse Jacobian matrices using automatic differentiation. *SIAM Journal on Scientific Computing*, 15(2):285–294, 1994.

[3] M.C. Bartholomew-Biggs, L. Bartholomew-Biggs, and B. Christianson. Optimization & automatic differentiation in ADA: Some practical experiences. *Optimization Methods & Software*, 4(1):47–73, 1994.

[4] Christian Bischof, Ali Bouaricha, Peyvand Khademi, and Jorge Moré. Computing gradients in large-scale optimization using automatic differentiation. Preprint MCS-P488-0195, Mathematics and Computer Science Division, Argonne National Laboratory, 1995.

[5] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1(1):11–29, 1992.

[6] Christian Bischof, Alan Carle, and Peyvand Khademi. Fortran 77 interface specification to the SparsLinC library. Technical Report ANL/MCS-TM-196, Mathematics and Computer Science Division, Argonne National Laboratory, 1994.

[7] Christian Bischof, Alan Carle, Peyvand Khademi, and Andrew Mauer. The ADIFOR 2.0 system for the automatic differentiation of Fortran 77 programs, 1994. Preprint MCS-P481-1194, Mathematics and Computer Science Division, Argonne National Laboratory, and CRPC-TR94491, Center for Research on Parallel Computation, Rice University. To appear in IEEE Computational Science & Engineering.

[8] Christian Bischof, Alan Carle, Peyvand Khademi, Andrew Mauer, and Paul Hovland. ADIFOR 2.0 user's guide. Technical Memorandum ANL/MCS-TM-192, Mathematics and Computer Science Division, Argonne National Laboratory, 1994. CRPC Technical Report CRPC-95516-S.

[9] Christian Bischof, George Corliss, Larry Green, Andreas Griewank, Kara Haigler, and Perry Newman. Automatic differentiation of advanced CFD codes for multidisciplinary design. *Journal on Computing Systems in Engineering*, 3(6):625–638, 1992.

[10] Christian Bischof, Larry Green, Kitty Haigler, and Tim Knauff. Parallel calculation of sensitivity derivatives for aircraft design using automatic differentiation. In *Proceedings of the 5th AIAA/NASA/USAF/ISSMO Symposium on Multidisciplinary Analysis and Optimization,* AIAA 94-4261, pages 73–84. American Institute of Aeronautics and Astronautics, 1994.

[11] Christian Bischof, Greg Whiffen, Christine Shoemaker, Alan Carle, and Aaron Ross. Application of automatic differentiation to groundwater transport models. In Alexander Peters, editor, *Computational Methods in Water Resources X*, pages 173–182, Dordrecht, 1994. Kluwer Academic Publishers.

[12] Christian H. Bischof and Moe El-Khadiri. Extending compile-time reverse mode and exploiting partial separability in ADIFOR. Technical Report ANL/MCS-TM-163, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[13] Christian H. Bischof, William T. Jones, Andrew Mauer, and Jamshid Samareh. Experiences with the application of the ADIC automatic differentiation tool to the CSCMDO 3-D volume grid generation code. In *Proceedings of the 34th AIAA Aerospace Sciences Meeting*, pages AIAA 96–0716, 1996. To appear.

[14] Ali Bouaricha and Jorge Moré. Impact of partial separability on large-scale optimization. Preprint MCS-P487-0195, Mathematics and Computer Science Division, Argonne National Laboratory, 1995.

[15] Alan Carle, Lawrence Green, Christian Bischof, and Perry Newman. Applications of automatic differentiation in CFD. In *Proceedings of the 25th AIAA Fluid Dynamics Conference, AIAA Paper 94-2197.* American Institute of Aeronautics and Astronautics, 1994.

[16] 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.

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

[18] Laurence C. W. Dixon. Use of automatic differentiation for calculating Hessians and Newton steps. In Andreas Griewank and George Corliss, editors, *Proceedings of the Workshop on Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 114–125, Philadelphia, 1991. SIAM.

[19] Laurence C. W. Dixon, Z. A. Maany, and M. Mohseninia. Automatic differentiation of large sparse systems. *Journal of Economic Dynamics and Control*, 14(2), 1990.

[20] Andreas Griewank. The chain rule revisited in scientific computing. *SIAM News*, 24, No. 3, p. 20 & No. 4, p. 8, 1991.

[21] Andreas Griewank, David Juedes, and Jean Utke. ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software*, June 1995. To appear.

[22] Andreas Griewank and Philippe L. Toint. On the unconstrained optimization of partially separable objective functions. In M. J. D. Powell, editor, *Nonlinear Optimization 1981*, pages 301–312, London, 1981. Academic Press.

[23] Jim E. Horwedel. GRESS: A preprocessor for sensitivity studies on Fortran programs. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 243–250. SIAM, Philadelphia, 1991.

[24] J. J. Moré and Z. Wu. Global continuation for distance geometry problems. Technical Report MCS-P505-0395, Argonne National Laboratory, 1995.

[25] Garry N. Newsam and John D. Ramsdell. Estimation of sparse Jacobian matrices. *SIAM Journal on Algebraic and Discrete Methods*, 4(3):404–418, 1983.

[26] Seon Ki Park, Kelvin Droegemeier, Christian Bischof, and Tim Knauff. Sensitivity analysis of numerically-simulated convective storms using direct and adjoint methods. In *Preprints, 10th Conference on Numerical Weather Prediction, Portland, Oregon*, pages 457–459. American Meterological Society, 1994.

[27] Nicole Rostaing, Stephane Dalmas, and Andre Galligo. Automatic differentiation in Odyssee. *Tellus*, 45a(5):558–568, October 1993.