ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

IMPACT OF PARTIAL SEPARABILITY ON LARGE-SCALE OPTIMIZATION

Ali Bouaricha and Jorge J. Moré

Mathematics and Computer Science Division

Preprint MCS-P487-0195

January 1995

(Revised version)

October 1995

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, and by the National Science Foundation, through the Center for Research on Parallel Computation, under Cooperative Agreement No. CCR-9120008.

IMPACT OF PARTIAL SEPARABILITY ON LARGE-SCALE OPTIMIZATION

Ali Bouaricha and Jorge J. Moré

Abstract

ELSO is an environment for the solution of large-scale optimization problems. With ELSO the user is required to provide only code for the evaluation of a partially separable function. ELSO exploits the partial separability structure of the function to compute the gradient efficiently using automatic differentiation. We demonstrate ELSO's efficiency by comparing the various options available in ELSO. Our conclusion is that the hybrid option in ELSO provides performance comparable to the hand-coded option, while having the significant advantage of not requiring a hand-coded gradient or the sparsity pattern of the partially separable function. In our test problems, which have carefully coded gradients, the computing time for the hybrid AD option is within a factor of two of the hand-coded option.

1 Introduction

ELSO is an environment for the solution of large-scale minimization problems

$$\min \{f_0(x) : x \in \mathbb{R}^n\}, \tag{1.1}$$

where $f_0 : \mathbb{R}^n \to \mathbb{R}$ is partially separable, that is, f_0 can be written as

$$f_0(x) = \sum_{i=1}^{m} f_i(x), \tag{1.2}$$

where each element function f_i depends only on a few components of x, and m is the number of element functions. Algorithms and software that take advantage of partial separability have been developed for various problems (for example, [14, 19, 20, 17, 21, 22, 10]), but this software requires that the user provide the gradient of f_0. An important design goal of ELSO is to avoid this requirement.

For small-scale problems we can approximate the gradient by differences of function values, for example,

$$[\nabla f_0(x)]_i \approx \frac{f_0(x + h_i e_i) - f_0(x)}{h_i}, \qquad 1 \le i \le n,$$

where h_i is the difference parameter, and e_i is the i-th unit vector, but this approximation suffers from truncation errors, which can cause premature termination of an optimization

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, and by the National Science Foundation, through the Center for Research on Parallel Computation, under Cooperative Agreement No. CCR-9120008.

algorithm far away from a solution. We also note that, even for moderately sized problems with $n \geq 100$ variables, use of this approximation is prohibitive because it requires n function evaluations for each gradient. For these reasons, the accurate and efficient evaluation of the gradient is essential for the solution of optimization problems.

ELSO is able to solve large-scale unconstrained optimization problems, while requiring only that the user provide the function in partially separable form. This is an important advantage over standard software that requires the specification of the gradient and the sparsity pattern of the partially separable function, that is,

$$\mathcal{S} = \{(i,j) : f_i \text{ depends on } x_j\} = \{(i,j) : \partial_j f_i(x) \not\equiv 0\}. \tag{1.3}$$

ELSO exploits the partial separability structure of the function to compute the gradient efficiently by using automatic differentiation (AD). The current version of ELSO incorporates four different approaches for computing the gradient of a partially separable function in the context of large-scale optimization software. These approaches are hand-coded, compressed AD, sparse AD, and hybrid AD. In our work we have been using the ADIFOR (Automatic Differentiation of Fortran) tool [4, 6], and the SparsLinC (Sparse Linear Combination) library [5, 6], but other differentiation tools can be used.

We demonstrate ELSO's efficiency by comparing the compressed AD, sparse AD, and hybrid AD options with the hand-coded approach. Our conclusion is that the performance of the hybrid AD option is comparable with the compressed AD option and that the performance penalty over the hand-coded option is acceptable for carefully coded gradients. In our test problems, which have carefully coded gradients, the computing time for the hybrid AD option is within a factor of two of the hand-coded option. Thus, the hybrid AD option provides near-optimal performance, while providing the significant advantage of not requiring a hand-coded gradient or the sparsity pattern of the partially separable function.

We describe in Section 2 the different approaches used by ELSO to compute the gradient of a partially separable function. In Section 3 we provide a brief description of the MINPACK-2 large-scale problems and show how to convert these problems into partially separable problems. In Section 4 we compare and analyze the performance of large-scale optimization software using the different options available in ELSO. We present results for both a superscalar architecture (IBM RS6000) and a vector architecture (Cray C90). Our results on the Cray C90 are of special interest because they show that if the hand-coded gradient does not run at vector speeds, the hybrid AD option can outperform the hand-coded option. Finally, we present our conclusions in Section 5.

2 Computing Gradients in ELSO

In addition to hand-coded gradients, ELSO supports three approaches based on automatic differentiation for computing the gradient of a partially separable function. In this section

we describe and compare these approaches.

ELSO relies on the representation (1.2) to compute the gradient of a partially separable function. Given this representation of $f_0 : \mathbb{R}^n \mapsto \mathbb{R}$, we can compute the gradient of f_0 by noting that if the mapping $f : \mathbb{R}^n \mapsto \mathbb{R}^m$ is defined by

$$f(x) = \begin{pmatrix} f_1(x) \\ \vdots \\ f_m(x) \end{pmatrix}, \tag{2.1}$$

then the gradient ∇f_0 can be obtained by

$$\nabla f_0(x) = f'(x)^T e, \tag{2.2}$$

where $e \in \mathbb{R}^m$ is the vector of all ones. The key observation is that the partial separability of f_0 implies that the Jacobian matrix $f'(x)$ is sparse, and thus automatic differentiation techniques can be used to compute the gradient ∇f_0 efficiently. The aim is to compute the gradient so that

$$T\{\nabla f_0(x)\} \leq \Omega_T \, T\{f_0(x)\}, \tag{2.3}$$

$$M\{\nabla f_0(x)\} \leq \Omega_M \, M\{f_0(x)\}, \tag{2.4}$$

where $T\{\cdot\}$ and $M\{\cdot\}$ denote computing time and memory, respectively, and Ω_T and Ω_M are small constants; if the function f_0 is defined by a discretization of a continuous problem, we also wish the constants to be independent of the mesh size. Any automatic differentiation tool can be used to compute $f'(x)$ and thus the gradient of f_0, but efficiency requires that we insist on (2.3) and (2.4).

Automatic differentiation tools can be classified roughly according to their use of the *forward* or the *reverse* mode of automatic differentiation. See, for example, the survey of Juedes [16]. Automatic differentiation tools that use the forward mode generate code for the computation of $f'(x)V$ for any $V \in \mathbb{R}^{n \times p}$. If $L\{f\}$ and $M\{f\}$ are, respectively, the number of floating-point operations and the amount of memory required by the computation of $f(x)$, then an AD-generated code employing the forward mode requires

$$L\{f'(x)V\} \leq (2 + 3p)\, L\{f\}, \qquad M\{f'(x)V\} \leq (1 + p)\, M\{f\},$$

floating-point operations and memory, respectively, to compute $f'(x)V$. For many large-scale problems we can obtain the Jacobian matrix $f'(x)$ by computing $f'(x)V$ for a matrix $V \in \mathbb{R}^{n \times p}$ with p small. Thus, in this case, an automatic differentiation tool based on the forward mode satisfies (2.3) and (2.4). We elaborate on this point when we discuss the compressed AD approach.

Automatic differentiation tools that use the reverse mode generate code for the computation of $W^T f'(x)$ for any $W \in \mathbb{R}^{m \times q}$. We can also use the reverse mode to compute $f'(x)$,

but since the reverse mode reverses the partial order of program execution and remembers (or recomputes) any intermediate result that affects the final result, the complexity of the reverse mode is harder to predict. In general, the reverse mode requires $\mathcal{O}(L\{f\})$ floating-point operations and up to $\mathcal{O}(L\{f\} + M\{f\})$ memory, depending on the code. In particular, there is no guarantee that (2.4) is satisfied. Griewank [11, 12] has discussed how to improve the performance of the reverse mode, but at present the potential memory demands of the reverse mode are a disadvantage. For additional information on automatic differentiation, see the proceedings edited by Griewank and Corlis [13]; the paper of Iri [15] is of special interest because he discusses the complexity of both the forward and the reverse modes of automatic differentiation.

In ELSO we have used the ADIFOR [4, 6] tool and the SparsLinC library [5, 6] because, from a computational viewpoint, they provide all the flexibility and efficiency desired on practical problems. Indeed, Bischof, Bouaricha, Khademi, Moré [3] have shown that the ADIFOR tool can satisfy (2.3) and (2.4) on large-scale variational problems.

We now outline the three approaches used by ELSO to compute the gradient of f_0. As we shall see, all these approaches have advantages and disadvantages in terms of ease of use, applicability, and computing time.

2.1 Compressed AD Approach

In the compressed AD approach we assume that the sparsity pattern of the Jacobian matrix $f'(x)$ is known for all vectors $x \in \mathcal{D}$, where $\mathcal{D}$ is a region where all the iterates are known to lie. For example, $\mathcal{D}$ could be the set

$$\mathcal{D} = \{x \in \mathbb{R}^n : f_0(x) \leq f_0(x_0)\},$$

where x_0 is the initial starting point. Thus, in the compressed AD approach we assume that the *closure* of the sparsity pattern is known. The sparsity pattern $\mathcal{S}(x)$ for $f'(x)$ at a given $x \in \mathcal{D}$ is just the set of indices

$$\mathcal{S}(x) = \{(i,j) : [f'(x)]_{i,j} \neq 0\};$$

the closure of the sparsity pattern of $f'(x)$ in the region $\mathcal{D}$ is

$$\bigcup \{\mathcal{S}(x) : x \in \mathcal{D}\}.$$

To determine the closure of the sparsity pattern, we are required to know how the function f_0 depends on the variables. When f is given by (2.1), a pair (i,j) is in the closure of the sparsity pattern if and only if f_i depends on x_j. Hence, the closure of the sparsity pattern is the sparsity pattern (1.3) of the partially separable function when x is restricted to lie in $\mathcal{D}$.

```
do j = 1, n
   grad(j) = 0.0
   do k = jpntr(j), jpntr(j+1)-1
      i = indrow(k)
      grad(j) = grad(j) + c_fjac(i,ngrp(j))
   enddo
enddo
```

Figure 2.1: Computing $\nabla f_0(x)$ from the compressed Jacobian array `c_fjac`

Given the sparsity pattern of $f'(x)$, we can determine the Jacobian matrix $f'(x)$ if we partition the columns of the Jacobian matrix into groups of *structurally orthogonal* columns, that is, columns that do not have a nonzero in the same row position. In our work we employ the partitioning software described by Coleman, Garbow, and Moré [8, 7].

Given a partitioning of the columns of $f'(x)$ into p groups of structurally orthogonal columns, we can determine the Jacobian matrix $f'(x)$ by computing the *compressed Jacobian* matrix $f'(x)V$, where $V \in \mathbb{R}^{n \times p}$. There is a column of V for each group, and the k-th column is determined by setting the i-th component of v_k to one if the i-th column is in the k-th group, and to zero otherwise. For many sparsity patterns, the number of groups p is small and independent of n. For example, if a matrix is banded with bandwidth β or if it can be permuted to a matrix with bandwidth β, Coleman and Moré [9] show that $p \leq \beta$.

The compressed Jacobian matrix contains all the information of the Jacobian matrix. Given the compressed Jacobian matrix, we can recover $f'(x)$ in a sparse data structure. We can eliminate the storage and floating-point operations required to determine the sparse representation of the Jacobian matrix $f'(x)$, however, by computing the gradient of f_0 directly from the compressed Jacobian array `c_fjac` and storing the result in the array `grad`. This way of computing the gradient of f_0 is shown in the code segment in Figure 2.1. In this figure, `indrow` is the row index of the sparse representation of $f'(x)$, and `jpntr` specifies the locations of the row indices in `indrow`. The row indices for column `j` are `indrow(k), k = jpntr(j),...,jpntr(j+1)-1`, and `ngrp` specifies the partition of the columns of the sparse representation of $f'(x)$; column `j` belongs to group `ngrp(j)`.

2.2 Sparse AD Approach

For the sparse AD approach we need an automatic differentiation tool that takes advantage of sparsity when V and most of the vectors involved in the computation of $f'(x)V$ are sparse. We also require the sparsity pattern of $f'(x)V$ as a by-product of this computation. At present, the SparsLinC library [5, 6] is the only tool that addresses this situation, but we expect that others will emerge.

The main advantage of the sparse AD approach over the compressed AD approach is that no knowledge of the sparsity pattern is required. A disadvantage, however, is that because of the need to maintain dynamic data structures for sparse vectors, the sparse AD approach usually runs slower than the compressed AD approach.

Numerical results [3] with ADIFOR and SparsLinC show that the compressed AD approach outperforms the sparse AD approach on various architectures. In fact,

$$T\{\nabla f_0(x) : \text{sparse ADIFOR}\} = \kappa \; T\{\nabla f_0(x) : \text{compressed ADIFOR}\},$$

where κ satisfies

SPARC 10	IBM RS6000	Cray C90
$3 \leq \kappa \leq 8$	$6 \leq \kappa \leq 20$	$15 \leq \kappa \leq 45$

.

These results show, for example, that the solution of an optimization problem with a relatively expensive function evaluation is likely to require at least three times longer if we use sparse ADIFOR instead of compressed ADIFOR. Of course, for the compressed AD option we need to supply the sparsity pattern of the partially separable function.

Also note that the performance penalty of sparse ADIFOR is worst on superscalar (IBM RS6000) and vector (Cray C90) architectures. Thus, for these architectures, there is a stronger need to obtain the advantages of the sparse AD approach without giving up the speed of the compressed AD approach.

2.3 Hybrid AD Approach

As stated in the introduction, an important design goal of ELSO is to avoid asking the user to provide code for the evaluation of the gradient or the sparsity pattern of the partially separable function. We can achieve this goal by using the sparse AD option. However, as noted above, this imposes a heavy performance penalty on the user.

In an optimization algorithm we can avoid this performance penalty by first using the sparse AD option, to obtain the sparsity pattern of the function, and then using the compressed AD option. This strategy must be used with care. We should not use the sparse AD option to obtain the sparsity pattern at the starting point because the starting point is invariably special, and not representative of a general point in the region $\mathcal{D}$ of interest. In particular, there are usually many symmetries in the starting point that are not necessarily present in intermediate iterates.

We can also use the sparse AD option for a number of iterates until we feel that any symmetries present in the starting point have been removed by the optimization algorithm. This strategy is not satisfactory, however, because optimization algorithms tend to retain symmetries for many iterations, possibly for all the iterates.

The current strategy in ELSO is to randomly perturb every component of the user's initial point, and compute the sparsity pattern at the perturbed point. This destroys any

Table 3.1: MINPACK-2 test problems

Name	Description of the Minimization Problems
EPT	Elastic-plastic torsion problem
GL1	Ginzburg-Landau (1-dimensional) superconductivity problem
GL2	Ginzburg-Landau (2-dimensional) superconductivity problem
MSA	Minimal surface area problem
ODC	Optimal design with composite materials problem
PJB	Pressure distribution in a journal bearing problem
SSC	Steady-state combustion problem

symmetries in the original iterates, and the resulting sparsity pattern is likely to be the closure of the sparsity pattern in $\mathcal{D}$.

This strategy may fail if the closure of the sparsity pattern in a neighborhood of the initial iterate is different from the sparsity pattern in a neighborhood of the solution. For most optimization problems, this does not occur. If it occurs, however, failure does not occur unless some entries in the current sparsity pattern are not present in the previous sparsity pattern. The justification of this remark comes about by noting that the compressed AD approach works provided the sparsity pattern of the Jacobian matrix $f'(x)$ is a subset of the sparsity pattern provided by the user. Of course, if the sparsity pattern provided by the user is too large, then the number of groups p is likely to increase, leading to increased memory requirements and some loss in efficiency in the computation of the gradient.

3 Partially Separable Test Problems

We used the test problems in the MINPACK-2 collection to compare the performance of a large-scale optimization software employing the four approaches for computing the gradient of a partially separable function described in Section 2. This collection is representative of large-scale optimization problems arising from applications. Table 3.1 lists each test problem with a short description; see [1] for additional information on these problems.

The optimization problems in the MINPACK-2 collection arise from the need to minimize a function f of the form

$$f(v) = \int_{\mathcal{D}} \Phi(x, v, \nabla v)\, dx, \tag{3.1}$$

where $\mathcal{D}$ is some domain in either $\mathbb{R}$ or $\mathbb{R}^2$, and Φ is defined by the application. In all cases f is well defined if $v : \mathcal{D} \mapsto \mathbb{R}^p$ belongs to $H^1(\mathcal{D})$, the Hilbert space of functions such that v and $\|\nabla v\|$ belong to $L^2(\mathcal{D})$.

Finite element approximations to these problems are obtained by minimizing f over the space of piecewise linear functions v with values $v_{i,j}$ at $z_{i,j}$, $0 \le i \le n_y + 1$, $0 \le j \le n_x + 1$, where $z_{i,j} \in \mathbb{R}^2$ are the vertices of a triangulation of $\mathcal{D}$ with grid spacings h_x and h_y. The

vertices $z_{i,j}$ are chosen to be a regular lattice so that there are n_x and n_y interior grid points in the coordinate directions, respectively. Lower triangular elements T_L are defined by vertices $z_{i,j}, z_{i+1,j}, z_{i,j+1}$, while upper triangular elements T_U are defined by vertices $z_{i,j}, z_{i-1,j}, z_{i,j-1}$. A typical triangulation is shown in Figure 3.1.

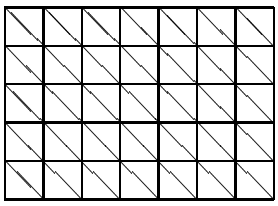

Figure 3.1: Triangulation of domain $\mathcal{D}$

In a finite element formulation of the variational problem defined by (3.1), the unknowns are the values $v_{i,j}$ of the piecewise linear function v at the vertices $z_{i,j}$. The values $v_{i,j}$ are obtained by solving the minimization problem

$$\min \left\{ \sum_{(i,j)} \left(f^L_{i,j}(v) + f^U_{i,j}(v) \right) : v \in \mathbb{R}^n \right\},$$

where $f^L_{i,j}$ and $f^U_{i,j}$ are the finite element approximation to the integrals

$$\int_{T_L} \Phi(x, v, \nabla v)\, dx, \qquad \int_{T_U} \Phi(x, v, \nabla v)\, dx,$$

respectively. Clearly, this is a partially separable problem because the element functions $f^L_{i,j}(v)$ and $f^U_{i,j}(v)$ depend only on the vertices $v_{i,j}, v_{i+1,j}, v_{i,j+1}$ and $v_{i,j}, v_{i-1,j}, v_{i,j-1}$, respectively. We can formulate this problem by setting

$$f(v) = \begin{pmatrix} f^L_{1,1}(v) \\ f^L_{1,2}(v) \\ \vdots \\ f^U_{1,1}(v) \\ f^U_{1,2}(v) \\ \vdots \end{pmatrix}. \tag{3.2}$$

In this case the number of element functions $m \approx 2n$. On the other hand, if we define

$$f(v) = \begin{pmatrix} f^L_{1,1}(v) + f^U_{1,1}(v) \\ f^L_{1,2}(v) + f^U_{1,2}(v) \\ \vdots \end{pmatrix}, \tag{3.3}$$

the number of element functions $m \approx n$. Since the number of element functions differs for (3.2) and (3.3), the number of groups p determined by the partitioning software [8, 7] is likely to be different, and thus the computing times for the compressed Jacobian matrix may depend on p. In our experience the computing time of formulation (3.2) is slightly better than that of (3.3). Therefore, we used formulation (3.2) in the numerical results of Section 4.

The problems in Table 3.1 are representative of a large class of optimization problems. These problems share some common characteristics. The main characteristics are that the computation of f requires order n flops and that the Jacobian matrix of f is sparse. Moreover, the number of groups p determined by the partitioning software leads to an almost dense compressed Jacobian matrix; the only exception is the GL2 problem, where the compressed Jacobian matrix is 50% dense. We expect that our numerical results are representative for any problem with these characteristics.

4 Numerical Results

Our aim in these experiments is to show that the performance of the hybrid AD option of ELSO is comparable to the compressed AD option and that the performance penalty over the hand-coded option is quite reasonable.

We chose a limited-memory variable metric method for these comparisons because codes of this type are commonly used to solve large-scale optimization problems. These methods are of the form

$$x_{k+1} = x_k - \alpha_k H_k \nabla f(x_k),$$

where $\alpha_k > 0$ is the search parameter, and the approximation H_k to the inverse Hessian matrix is stored in a compact representation that requires only the storage of $2n_v$ vectors, where n_v is chosen by the user. The compact representation of H_k permits the efficient computation of $H_k \nabla f(x_k)$ in $(8n_v + 1)n$ flops; all other operations in an iteration of the algorithm require $11n$ flops.

We used the `vmlm` implementation of the limited-memory variable metric algorithm (see Averick and Moré [2]), which is based on the work of Liu and Nocedal [18]. In all of our tests we used $n_v = 5$. Instead of using a termination test, such as

$$\|\nabla f(x)\| \leq \tau \|\nabla f(x_0)\|,$$

we terminate after 100 iterations. This strategy is needed because optimization algorithms that require many iterations for convergence are affected by small perturbations in the function or the gradient, and, as a result, there may be large differences in the number of iterations required for convergence when the different `vmlm` options of ELSO are used.

All computations were performed on two platforms: an IBM RS6000 (model 370) using double-precision arithmetic, and a Cray C90 using single-precision arithmetic. The IBM

RS6000 architecture has a superscalar chip and a cache-based memory architecture. Hence, this machine performs better when executing short vector operations, since these operations fill the short pipes and take advantage of memory locality. The Cray C90 is a vector processor without a cache that achieves full potential when the code has long vector operations. Without optimization of the source Fortran code, short vector loops and indirect addressing schemes perform poorly.

Table 4.1 has the computing time ratios of the compressed AD and sparse AD function-gradient evaluation to the hand-coded function-gradient evaluation on the IBM RS6000. These results show that the use of the sparse AD gradient can lead to a significant degradation in performance.

Table 4.1: Computing time ratios of the compressed AD and sparse AD function-gradient evaluation to the hand-coded function-gradient evaluation on the IBM RS6000 with $n = 10,000$

Prob	Compressed AD	Sparse AD
EPT	3.6	44.5
GL1	8.5	164.3
GL2	5.7	34.9
MSA	1.8	14.5
ODC	3.2	22.8
PJB	4.5	54.7
SSC	2.6	19.8

Tables 4.2 and 4.3 compare the computing time for the compressed AD, sparse AD, and hybrid AD options of `vmlm` to the computing time of the hand-coded option. The most important observation that can be made from these tables is that the computing times for the hybrid AD option are approximately the same as those for the compressed AD option. The performance similarity between the hybrid AD option and the compressed AD option is expected because the difference in cost between the two options is only one sparse AD gradient evaluation and the partitioning of the columns of the Jacobian matrix into groups of structurally orthogonal columns. Tables 4.2 and 4.3 show that the hybrid AD option is clearly the method of choice because of its significant advantage of not requiring a hand-coded gradient or the sparsity pattern of the partially separable function.

The ratios in Tables 4.2 and 4.3 are below the corresponding ratios in Table 4.1. This result can be explained by noting that the ratios in Tables 4.2 and 4.3 can be expressed as

$$\frac{T_{ad} + T_{alg}}{T_{hc} + T_{alg}}, \tag{4.1}$$

where T_{ad}, T_{alg}, and T_{hc} are the computing times for the function and AD-generated gradient evaluation, the `vmlm` algorithm, and the function and hand-coded gradient evaluation,

Table 4.2: Computing time ratios of the compressed AD, sparse AD, and hybrid AD options of `vmlm` to the hand-coded option on the IBM RS6000 with $n = 10,000$

Prob	Compressed AD	Sparse AD	Hybrid AD
EPT	2.6	19.1	2.8
GL1	2.3	17.6	2.5
GL2	2.6	12.6	2.8
MSA	1.6	10.0	1.7
ODC	2.0	10.0	2.1
PJB	3.2	15.9	3.4
SSC	2.2	13.1	2.3

Table 4.3: Computing time ratios of the compressed AD, sparse AD, and hybrid AD options of `vmlm` to the hand-coded option on the IBM RS6000 with $n = 40,000$

Prob	Compressed AD	Sparse AD	Hybrid AD
EPT	2.8	19.2	3.0
GL1	2.3	17.6	2.5
GL2	2.8	12.2	2.9
MSA	1.7	10.1	1.9
ODC	2.1	10.0	2.2
PJB	3.3	14.8	3.4
SSC	2.3	13.3	2.4

respectively. Since $T_{ad} > T_{hc}$, we have

$$\frac{T_{ad} + T_{alg}}{T_{hc} + T_{alg}} \leq \frac{T_{ad}}{T_{hc}},$$

which is the desired result. If T_{ad} and T_{hc} are the dominant costs, the ratio (4.1) should be close to T_{ad}/T_{hc}. This can be seen in the results for the MSA and SSC problem, since these are the two most expensive functions in the set.

Tables 4.2 and 4.3 also show that when we increase the problem dimension from $n = 10,000$ to $n = 40,000$, the corresponding compressed AD, sparse AD, and hybrid AD ratios remain about the same. This observation can be explained by noting that the ratio (4.1) can also be expressed as

$$\frac{rT_{hc} + T_{alg}}{T_{hc} + T_{alg}}, \tag{4.2}$$

where r is the ratio in Table 4.1. Since T_{hc} and T_{alg} grow by approximately a factor of 4 when n changes from 10,000 to 40,000, the ratio (4.2) remains constant.

We present results only for the SSC and GL2 problems on the Cray C90. We selected these problems because they have different characteristics. In particular, the number of groups in the compressed AD approach is $p = 3$ for the SSC problem, while $p = 9$ for the GL2 problem.

Table 4.4: Computing time ratios of the compressed AD and sparse AD function-gradient evaluation to the hand-coded function-gradient evaluation on the Cray C90

Prob	n	Compressed AD	Sparse AD
GL2	10000	25.1	624.6
GL2	40000	28.3	694.4
SSC	10000	1.9	49.2
SSC	40000	1.9	49.4

Table 4.5: Computing time ratios of the compressed AD and hybrid AD options of `vmlm` to the hand-coded option on the Cray C90

Prob	n	Compressed AD	Hybrid AD
GL2	10000	14.2	18.0
GL2	40000	16.9	20.3
SSC	10000	1.9	2.4
SSC	40000	1.9	2.4

Table 4.4 presents the computing time ratios of the compressed AD and sparse AD function-gradient evaluation to the hand-coded function-gradient evaluation. The sparse AD approach uses the indirect addressing and dynamic memory allocation of the SparsLinC library [5, 6] and thus performs poorly on vector architectures [3]. As a result, the performance of the sparse AD approach is far from being practical on the Cray. In the rest of this section we present results only for the compressed and hybrid AD options.

Table 4.5 presents the computing time ratios of the compressed AD and hybrid AD options of `vmlm` to the hand-coded option. These results show that the performance of the hybrid AD option is comparable to that of the compressed AD option. On the other hand, the performance of the compressed AD option relative to the hand-coded option is poor for the GL2 problem. The reason for this poor performance is that the GL2 hand-coded gradient fully vectorizes, while the compressed AD gradient does not vectorize. Hence, the hand-coded gradient executes at vector speeds, while the compressed AD gradient executes at scalar speeds. The situation is different for the SSC function. In this case, neither the hand-coded gradient nor the compressed AD gradient vectorizes, so they both execute at scalar speeds.

The poor performance of the compressed AD and hybrid AD options is due to the short innermost loops of length p, where p is the number of groups in the compressed AD approach. These loops are vectorizable, but when the compiler vectorizes only innermost loops, as is the case of the Cray C90, the performance degrades. We can vectorize the compressed AD gradient by *strip-mining* the computation of the gradient; that is, the gradient computation is divided into strips and each strip computes the gradient with respect to a few components of the independent variables. In the case of the compressed AD gradient, strip-mining can

Table 4.6: Computing time ratios of the compressed AD function-gradient evaluation (with loop unrolling) to the hand-coded function-gradient evaluation on the Cray C90

Prob	n	Compressed AD
GL2	10000	13.3
GL2	40000	13.7
SSC	10000	0.4
SSC	40000	0.4

Table 4.7: Computing time ratios of the compressed AD and hybrid AD options of `vmlm` (with loop unrolling) to the hand-coded option on the Cray C90

Prob	n	Compressed AD	Hybrid AD
GL2	10000	8.9	12.7
GL2	40000	10.0	13.4
SSC	10000	0.5	1.0
SSC	40000	0.5	1.0

be done conveniently via the seed matrix mechanism. A disadvantage of the strip-mining approach is that the function is evaluated in every strip, resulting in a runtime overhead of `nstrips` $-$ `1` extra function evaluations, where `nstrips` is the number of strips. Using strips of size 5 is appropriate for the Cray C90 because the compiler unrolls innermost loops of length five or less, and, as a result, the loops that run over the grid points in the second coordinate direction are vectorized.

There is one additional complication. Since the value of p is not known at compile time, the Cray compiler cannot unroll a loop of length p even if the computed value of p at runtime is less than or equal to five. We fix this problem by setting the upper bound of the innermost loops to a fixed number at least equal to p but at most equal to 5. The generation of the compressed AD gradients with a fixed upper bound of the innermost loops can be done automatically by setting the appropriate ADIFOR flags [6].

The computing time ratios for the strip-mining approach (with loop unrolling) are shown in Tables 4.6 and 4.7. The improvement is dramatic for both the compressed AD and hybrid AD options. If we compare the results in Table 4.5 with those in Table 4.7, we find that the computing time ratios are reduced by a factor of 1.6 for the GL2 problem and a factor of 2 for the SSC problem.

Also note that the results in Tables 4.6 and 4.7 show that the compressed AD approach performs better on the SSC problem than the hand-coded approach. The reason for this is that the strip-mining in the compressed AD approach improves the performance of this approach, while the hand-coded approach is still running at scalar speeds. These results illustrate the important point that the compressed and hybrid AD approaches can run faster than the hand-coded approach if the user does not provide a carefully coded gradient.

5 Conclusions

We have developed an environment for the solution of large-scale optimization problems, ELSO, in which the user is required to provide only code for the evaluation of a partially separable function. ELSO exploits the partial separability structure of the function to compute the gradient efficiently using automatic differentiation.

Our test results show that the hybrid option in ELSO provides performance that is often not more than two times slower than a well-coded hand-derived gradient on superscalar architectures, while having the significant advantage of not requiring a hand-coded gradient or the sparsity pattern of the partially separable function.

Acknowledgments

We wish to thank Christian Bischof and Peyvand Khademi for their assistance with the ADIFOR tool and the SparsLinC library.

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, Revised, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[2] B. M. Averick and J. J. Moré, *Evaluation of large-scale optimization problems on vector and parallel architectures*, SIAM J. Optimization, 4 (1994), pp. 708–721.

[3] C. Bischof, A. Bouaricha, P. Khademi, and J. J. Moré, *Computing gradients in large-scale optimization using automatic differentiation*, Preprint MCS-P488-0195, Argonne National Laboratory, Argonne, Illinois, 1995.

[4] C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland, *ADIFOR: Generating derivative codes from Fortran programs*, Scientific Programming, 1 (1992), pp. 1–29.

[5] C. Bischof, A. Carle, and P. Khademi, *Fortran 77 interface specification to the SparsLinC library*, Technical Report ANL/MCS-TM-196, Argonne National Laboratory, Argonne, Illinois, 1994.

[6] C. Bischof, A. Carle, P. Khademi, and A. Mauer, *The ADIFOR 2.0 system for the automatic differentiation of Fortran 77 programs*, Preprint MCS-P381-1194, Argonne National Laboratory, Argonne, Illinois, 1994. Also available as CRPC-TR94491, Center for Research on Parallel Computation, Rice University.

[7] T. F. Coleman, B. S. Garbow, and J. J. Moré, *Fortran subroutines for estimating sparse Jacobian matrices*, ACM Trans. Math. Software, 10 (1984), pp. 346–347.

[8] ——, *Software for estimating sparse Jacobian matrices*, ACM Trans. Math. Software, 10 (1984), pp. 329–345.

[9] T. F. Coleman and J. J. Moré, *Estimation of sparse Jacobian matrices and graph coloring problems*, SIAM J. Numer. Anal., 20 (1983), pp. 187–209.

[10] A. R. Conn, N. I. M. Gould, and P. L. Toint, LANCELOT, Springer Series in Computational Mathematics, Springer-Verlag, 1992.

[11] A. Griewank, *Achieving logarithmic growth of temporal and spatial complexity in reverse communication*, Optim. Methods Software, 1 (1992), pp. 35–54.

[12] ——, *Some bounds on the complexity of gradients, Jacobians, and Hessians*, in Complexity in Nonlinear Optimization, P. Pardalos, ed., World Scientific Publishers, 1993, pp. 128–161.

[13] A. Griewank and G. F. Corliss, eds., *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, Society for Industrial and Applied Mathematics, 1991.

[14] A. Griewank and P. L. Toint, *Numerical experiments with partially separable optimization problems*, in Numerical Analysis: Proceedings Dundee 1983, D. F. Griffiths, ed., Lecture Notes in Mathematics 1066, Springer-Verlag, 1984.

[15] M. Iri, *History of automatic differentiation and rounding error estimation*, in Automatic Differentiation of Algorithms, A. Griewank and G. F. Corliss, eds., SIAM, 1992, pp. 3–16.

[16] D. Juedes, *A taxonomy of automatic differentiation tools*, in Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. Corliss, eds., SIAM, 1991, pp. 315–329.

[17] M. Lescrenier, *Partially separable optimization and parallel computing*, Ann. Oper. Res., 14 (1988), pp. 213–224.

[18] D. C. Liu and J. Nocedal, *On the limited memory BFGS method for large scale optimization*, Math. Programming, 45 (1989), pp. 503–528.

[19] P. L. Toint, *Numerical solution of large sets of algebraic nonlinear equations*, Math. Comp., 46 (1986), pp. 175–189.

[20] ——, *On large scale nonlinear least squares calculations*, SIAM J. Sci. Statist. Comput., 8 (1987), pp. 416–435.

[21] P. L. Toint and D. Tuyttens, *On large-scale nonlinear network optimization*, Math. Programming, 48 (1990), pp. 125–159.

[22] ——, *LSNNO: A Fortran subroutine for solving large-scale nonlinear network optimization problems*, ACM Trans. Math. Software, 18 (1992), pp. 308–328.