Optimal Time and Minimum Space-Time Product for Reversing a Certain Class of Programs

José Grimm, Loïc Pottier, Nicole Rostaing-Schmidt

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Optimal time and minimum space-time product for reversing a certain class of programs

José Grimm, Loïc Pottier, Nicole Rostaing-Schmidt

N° 2794

Février 1996

THÈME 2

ISSN 0249-6399

INRIA
SOPHIA ANTIPOLIS

Optimal time and minimum space-time product for reversing a certain class of programs

José Grimm, Loïc Pottier, Nicole Rostaing-Schmidt

Thème 2 — Génie logiciel
et calcul symbolique
Projet SAFIR

Rapport de recherche n ° 2794 — Février 1996 — 19 pages

Abstract: This paper concerns space-time trade-offs for the reverse mode of automatic differentiation on the straight-line programs with nested loops. In the first part we consider the problem of reversing a finite sequence given by $u_{n+1} = f(u_n)$, which can model a certain class of finite loops. We show an optimal time strategy for this problem, the number of available registers being fixed, and a lower bound on the space-time product equal to $\frac{p(\ln p)^2}{(\ln 4)^2}$. We then present an optimal strategy on nested loops with the objective of preserving the program structure. Finally, we consider an application of this storage/recomputation strategy to compute in reverse mode the derivatives of a function represented as a Fortran program.

Key-words: Automatic differentiation, complexity, fortran

(Résumé : tsvp)

La première partie de ce travail a été présentée au colloque de complexité algébrique en mémoire de Jacques Morgenstern, à l'INRIA Sophia Antipolis, en mai 1995

Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 – Télécopie : (33) 93 65 77 65

Temps optimal et espace temps minimal de l'inversion d'une certaine classe de programmes

Résumé : Nous nous intéressons ici aux compromis entre l'espace et le temps possibles dans le mode inverse de la différentiation automatique de programmes d'évaluation avec des boucles imbriquées. Dans la première partie, nous étudions le problème de l'inversion d'une suite finie donnée par récurrence : $u_{n+1} = f(u_n)$. Ce type de suites permet de modéliser une large classe de boucles finies. Pour ce problème, nous donnons une stratégie optimale en temps à nombre de registres fixés, et nous démontrons une borne inférieure atteinte pour le produit espace-temps égale à $\frac{p(\ln p)^2}{(\ln 4)^2}$, où p est la longueur de la suite. Nous en déduisons une stratégie optimale d'inversion pour les boucles imbriquées avec l'objectif de conserver la structure du programme. Finalement, nous étudions une application de ces stratégies de stockage/recalcul au calcul en mode inverse des différentielles d'une fonction donnée par un programme en FORTRAN.

Mots-clé : Différentiation automatique, complexité, fortran

1 Introduction

This article is devoted to the study of storage/recomputation trade-offs for the reverse mode of automatic differentiation. This problem has been studied in this context by A. Griewank [Griewank1992a] and by J. Abbott and A. Galligo [Abbott1991a]. In [Morgenstern1985a], J. Morgenstern gave an elegant proof of a theorem of Baur and Strassen [Baur1983a], stating that a rational function f and its derivatives can be computed in a time at most equal to five times that needed to compute the function itself. This method uses intermediate values of f in reverse order of their computation. Time and space are closely related in programs. If we can store intermediate results in registers, we will save time in subsequent computations. This fact motivates the study of lower bounds for the space-time product (see [Borodin1993a] for an overview on space-time lower bounds).

In the first part of this article, we consider the problem of printing in reverse order a sequence $u_0, \ldots, u_p$ of distinct elements, using a certain amount of storage (r registers). The sequence is given by its first element u_0 and a function f. Each element u_i is computed from the previous one $u_i = f(u_{i-1})$, stored in a register. We assume that one application of f costs one unit of time. Generalizing work of [Abbott1991a] on "divide and conquer" algorithms and [Griewank1992a] on "checkpointing strategy" for reversing a sequence, we prove a sharp asymptotic lower bound on the space-time product needed for all straight-line programs reversing a sequence. In the next section, we extend the treated class of programs. We look at programs containing two nested loops. Our objective in that section is to find an optimal time strategy that preserves the program structure with a fixed total number of registers. We particularly concentrate on the distribution of the available registers among the loops. We then consider the case where the time of one application of f is not constant and show an optimal strategy in this case. We finally present an application of this strategy to compute in reverse mode, the derivatives of a function represented as a Fortran program, instead of just printing values. The derivatives are computed using the Odyssée system [Rostaing1993a]. We show how the experimental computational times obtained on this test case support the theory. We conclude with suggestions for future research.

2 Optimal Strategy with a Fixed Number of Registers

In this section, for the reader's convenience, we review results presented at the "Colloque de complexité algébrique" (May 1995) in memory of Jacques Morgenstern.

2.1 Model of Algorithms

We represent algorithms as *straight-line programs* (SLPs). In our case, an SLP is a sequence of instructions $R_i := f(R_j)$, or $Print(R_i)$, where $R_i, 0 \leq i \leq r$ denotes the ith register. We assume that all registers initially contain the value u_0 and that the value of R_0 never changes. An SLP reverses the sequence iff it prints $u_p, \ldots, u_0$ in that order. The quantity p is the length of the sequence, and the quantity r is the number of registers. (The starting value u_0 is special: it is not counted in the length. The register that contains it is also special (read-only): we do not count it in the number of registers.)

A useful register is one that will be used later, before its value has changed. For each SLP and each index j, let (x_{ij}) be the list of indices of the sequence elements contained in useful registers, in increasing order, just before the value u_j is printed.

Let us start with an example: the inversion of a sequence of length five ($p = 5$) using three registers. We represent a register by o with its number above (Register R_0 is not shown). We then have the following sequence and the corresponding SLP. In the left part of the example, each line represents the state of the registers when an element of the sequence is printed, and each column represents one value of the sequence: the first one is u_0, the last one u_5.

```
   1   2 3
. . o . o o        R1 := f(R0)
                   R1 := f(R1)
  3 1   2          R2 := f(R1)
. o o . o          R2 := f(R2)
                   R3 := f(R2)
  3 1 2            Print(R3)
. o o o            R3 := f(R0)
                   Print(R2)
  3 1              R2 := f(R1)
. o o              Print(R2)
                   Print(R1)
  3                Print(R3)
. o                Print(R0)
```

The computation time is equal to 7. In this case, the lists (x_{ij}) are $(2, 4, 5)$, $(1, 2, 4)$, $(1, 2, 3)$, $(1, 2)$, and (1).

Definition 1 *An inversion $\mathcal{I}$ of integers of the sequence $u_0, \ldots, u_p$ with r registers is a family $(x_{i,j})_{0 \leq i \leq r_j, 0 \leq j \leq p}$ such that*

- *for* $i < r_j, x_{i,j} < x_{i+1,j}$*;*
- $x_{0,j} := 0, x_{r_j,j} := j$*; and*
- $r_j \leq r$.

We define the computation time of an SLP to be the number of instructions involving f, and the time of an inversion as the minimum time of any SLP that realizes this inversion. Then the time of an inversion $\mathcal{I}$ is $t_{\mathcal{I}} := \sum_{ij} t_{ij}$, where t_{ij} is the time needed to compute R_i at step j, $t_{ij} = x_{i,j} - y_{ij}$ where $y_{ij} := \sup(x_{i-1,j}, \sup_k\{x_{k,j+1} | x_{k,j+1} \leq x_{i,j}\})$, with the special cases $y_{ip} = x_{i-1,p}$ and $y_{0j} = 0$. The corresponding SLP can be constructed as indicated in Figure 1.

2.2 Optimal Inversions

An inversion is optimal when its time is minimum among all possible inversions.

Lemma 1 *For every inversion $\mathcal{I}$, there exists an inversion $\mathcal{I}' := (x'_{ij})$ such that (x'_{1j}) decreases when j decreases with $t_{\mathcal{I}'} \leq t_{\mathcal{I}}$.*

proof Let $\mathcal{I} := (x_{i,j})$ be an inversion. If $(x_{1,j})$ decreases, we are done. If not, a smallest j and a greatest i exist such that

$$x_{1,j+1} \leq x_{i,j+1} < x_{1,j} \leq x_{i+1,j+1}. \tag{1}$$

Consider first the case where $x_{1,j} = x_{i+1,j+1}$. Let $\mathcal{I}' := \mathcal{I}$, without $x_{i,j+1}$. Computation times t_{ik} are unaffected for $k \geq j$. Other computation times cannot be greater, since there is one less value to keep in a register. Otherwise, let $\mathcal{I}' := \mathcal{I}$ except for $x'_{i,j+1} := x_{1,j}$. Let $\delta = x_{ij} - x_{i,j+1}$. Since $x_{1,j}$ was computed from $x_{i,j+1}$ at a cost δ, we gain δ instructions to compute $x_{i,j}$ for $\mathcal{I}'$. We lose at most δ instructions to shift the register holding $x_{i,j+1}$ by δ to the right.

Hence, we find $\mathcal{I}'$ with $t_{\mathcal{I}'} \leq t_{\mathcal{I}}$. We can repeat this process as long as $i \geq 1$. After that, either the condition of the lemma is satisfied, or (1) is true for a greater j. It suffices to iterate over all j. QED.

Define (C_q) to be the condition $x_{1,p} = x_{1,p-1} = \ldots = x_{1,q} = q$. This condition says that the first register remains constant until its value is printed.

Lemma 2 *Let $\mathcal{I}$ be an inversion with $(x_{1,j})$ decreasing, and let $q := x_{1,p}$. Then there exists an inversion $\mathcal{I}'$ satisfying (C_q), with $t_{\mathcal{I}'} \leq t_{\mathcal{I}}$.*

proof Let $\mathcal{I}' := (x'_{ij})$ be such that $x'_{ij} = q$, if $j \geq q$ and $x_{ij} \leq q$, and $x'_{ij} = x_{ij}$ otherwise. (If more than one x_{ij} is $\leq q$, some indices must be renamed.)

We split times t_{ij} into three parts $t_{ij} = t^1_{ij} + t^2_{ij} + t^3_{ij}$, in the following way

```
for j = p downto 0 do
    set E to the empty list
C E is the list of non-empty registers
    for each i such that y_ij = sup_k(x_{k,j+1}) do
        let K such that R_K contains u_{x_{k,j+1}}
        if K ≠ 0
                repeat t_ij times
                    add R_K := f(R_K) to the SLP
                end repeat
            add R_K to E
        else
            let R_s be the register that holds u_{j+1}
            add R_s := f(R_0) to the SLP
            repeat t_ij − 1 times
                add R_s := f(R_s) to the SLP
            end repeat
            add R_s and R_k to E
        end if
    end for
C There remains to compute a certain number n of values.
C E contains at least n elements
    for each i such that y_ij ≠ sup_k(x_{k,j+1}) do
        take R_K not in E, add it to E
        let R_s be the register that holds u_{x_{i−1,j}}
        add R_K := f(R_s) to the SLP
        repeat t_ij − 1 times
            add R_K := f(R_K) to the SLP
        end repeat
    end for
    if u_j is in R_k, add Print(R_K) to the SLP
end for
```

Figure 1: Contruction of an SLP from an Inversion

- If $x_{ij} \leq q$, then $t^1_{ij} := t_{ij}, t^2_{ij} := t^3_{ij} := 0$.
- If $x_{ij} > q, y_{ij} \geq q$, then $t^1_{ij} := t^2_{ij} := 0, t^3_{ij} := t_{ij}$.
- If $x_{ij} > q, y_{ij} < q$, then $t^1_{ij} := 0, t^2_{ij} := q - y_{ij}, t^3_{ij} := x_{ij} - q$.

Define $t_k := \sum t^k_{ij}$, the sum over all $j \geq q-1$. If $t_4 = \sum_{i,j<q-1} t_{ij}$, the computation time is $t_1 + t_2 + t_3 + t_4$.

Since we changed x_{ij} only for $j \geq q$, we have $t'_4 = t_4$. Clearly $0 = t'_2 \leq t_2$. It is not difficult to prove $t'_3 = t_3$. Finally, $t'_1 = 2q - 1 \leq t_1$: whatever strategy is used, the time to compute u_{q-1} if the first register is at location q is at least $q-1$; to this we must add the time to compute u_q. QED.

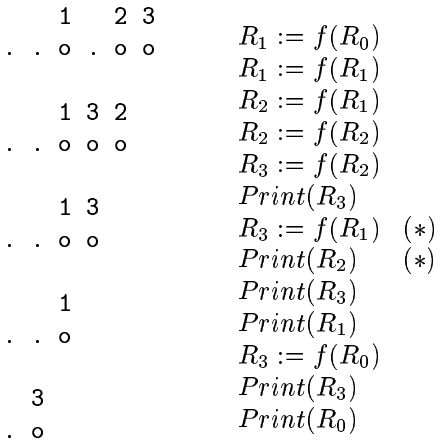
Now let $\mathcal{I}$ be an inversion. Using the previous lemmas, we can replace $\mathcal{I}$ with an inversion satisfying (C_q) and time not greater than $t_{\mathcal{I}}$.

Clearly $\mathcal{I}_1 := (x_{ij})_{j<q}$ is an inversion of the sequence $u_0, \ldots, u_{q-1}$ with r registers. Similarly, $\mathcal{I}_2 := (x_{ij})_{j \geq q}$ can be considered as an inversion of the sequence $u_q, \ldots, u_p$ with $r-1$ registers. $\mathcal{I}$ is called a "divide and conquer" inversion (*DC-inversion*) of $u_0, \ldots, u_p$ with r registers if $\mathcal{I}$ satisfies (C_q), and $\mathcal{I}_1$ and $\mathcal{I}_2$ are either trivial $(q = 0)$ or are DC-inversions. Iterative application of the previous lemmas gives immediately the following theorem.

Theorem 1 *There exists an optimal inversion reversing $u_0, \ldots, u_p$ with r registers. The optimal inversion is a DC-inversion.*

2.3 Example

Applying Lemma 2 to the previous example, we obtain the program shown below. The computation time is still equal to 7. We can obtain a DC-inversion by applying Lemma 2 again, that is, by exchanging instructions marked with $(*)$.



2.4 Optimal Time

Theorem 1 shows that the initial problem can be reduced to the study of optimal time for DC-inversions. Let us define $M_r^k := \binom{r+k-1}{r}$. The optimal time in Theorem 2 is proved in [Abbott1991a] in the restricted context of DC-inversions.

Theorem 2 *For every p, let k satisfy*

$$M_r^k - 1 \leq p \leq M_r^{k+1} - 1, \text{ and} \tag{2}$$

$$T(r,p) := k(p+1) - M_{r+1}^k. \tag{3}$$

Then $T(r,p)$ is the time of any optimal inversion of the sequence $u_0, \ldots, u_p$ with r registers. Moreover, a DC-inversion is optimal iff its index q satisfies

$$M_r^{k-1} \leq q \leq M_r^k, \text{ and} \tag{4}$$

$$M_{r-1}^k - 1 \leq p - q \leq M_{r-1}^{k+1} - 1. \tag{5}$$

proof Let $f(q) = q + T(r, q-1) + T(r-1, p-q)$, and let E be the set of all q satisfying (4) and (5). It suffices to show that f is minimum iff q is in E and that the minimum value of f is $T(r,p)$. In fact, let $\mathcal{I}$ be an optimal DC-inversion, and define $\mathcal{I}_1$ and $\mathcal{I}_2$ as before. By induction, computation times for $\mathcal{I}_1$ and $\mathcal{I}_2$ are $T(r, q-1)$ and $T(r-1, p-q)$. Hence, the computation time for $\mathcal{I}$ is $f(q)$. Since $\mathcal{I}$ is optimal, q is in E, and the computation time is $T(r,p)$.

The key relation in the proof is $M_r^{k+1} = M_{r-1}^{k+1} + M_r^k$. In particular, it implies that $T(r,p)$ is welldefined, even if k is not uniquely defined by (2). It also shows that E is not empty iff k satisfies (2). In fact, E is an interval $[q_0, q_1]$. Let us define $f(q)$ for every real q. Note that this is a piecewise linear, continuous function. If p is not an integer, a unique $k_{r,p}$ exists such that (2) is satisfied. Define $k_1 = k_{r,q-1}$ and $k_2 = k_{r-1,p-q}$. Then $f(q) = (1 + k_1 - k_2)q + X$, with $X = k_2(p-1) - M_{r+1}^{k_1} - M_r^{k_2}$. On E, we have $k_1 = k_{r,p} - 1$ and $k_2 = k_{r,p}$, so that $f(q) = T(r,p)$. On every interval not containing an integer, f is differentiable, and its derivative is $1 + k_1 - k_2$. It is obvious that $f'(q) > 0$ if $q > q_1$ and $f'(q) < 0$ if $q < q_0$ (since $q > q_1$ is equivalent to $k_1 > k_{r,p} - 1$ or $k_2 < k_{r,p}$). This shows that $f(q) > T(r,q)$ for every noninteger q not in E, and the continuity of f gives us the conclusion. QED.

2.5 A Sharp Lower Bound on the Space-Time Product

Theorem 3 *For a straight-line program reversing $u_0, \ldots, u_p$ with r registers in time T,*

$$(r+1)T = \Omega(p(\ln p)^2).$$

Moreover, $(r+1)T$ is greater than an expression in p which is asymptotically equivalent to $\frac{p(\ln p)^2}{(\ln 4)^2}$. This lower bound is sharp because there exist sequences of length p arbitrarily large for which $(r+1)T$ is asymptotically equivalent to $\frac{p(\ln p)^2}{(\ln 4)^2}$.

proof We give only an outline of the proof using the notation of Theorem 2. We have $(r+1)T \geq (p+1)r(k-1)$ and $M_r^{k+1} \leq \frac{(r+k)^{r+k}}{r^r k^k}$. Let k' be such that

$$f(r,k') := \frac{(r+k')^{r+k'}}{r^r k'^{k'}} = p+1.$$

Then $k' \leq k$. The problem is reduced to minimizing $r(k'-1)$, knowing $f(r,k') = p+1$, when r varies. A careful, but elementary study of this constrained minimization shows that the minimum is obtained when $r = k' + \frac{\ln 2}{1-\ln 2} + \epsilon(k')$ with $\lim_{p\to\infty} \epsilon(k') = 0$ and that the minimum value of $r(k'-1)$ is asymptotically equivalent to $\frac{p(\ln p)^2}{(\ln 4)^2}$. The study of sequences of length $p = M_r^r - 1$ shows that the bound is sharp. QED.

3 Programs with Nested Loops

Let us consider the sequence computed by the following program of two nested loops

```
for i=1 to n do
   for j=1 to m do
      x:=f(i,j,x)
```

We assume again that one application of f costs one unit of time. The problem here is to find a strategy for reversing this sequence of length nm with the constraint that the structure of nested loops be preserved. We seek to minimize the computational time with a fixed number of registers available to reverse each loop.

We denote by r_1 the number of registers reversing the i-loop and by r_2 the number of registers to reverse the j-loop. The minimal time $T(r_1, n, r_2, m)$ will be obtained by applying the DC-strategy first to the i-loop, the j-loop being considered as a single statement, and second to the j-loop. Using the notation of Theorem 2, we calculate the minimal time for reversing the two loops in this way as

$$T(r_1, n, r_2, m) = mT(r_1, n) + nT(r_2, m-1).$$

We assume now that the sum of the available registers $r = r_1 + r_2$ is fixed. We would like to compute the value of r_1 that gives minimal time $T(r_1, n, r_2, m)$. However, the function $r_1 \mapsto T(r_1, n, r - r_1, m)$ is difficult to study. We can see in Figure 2 an example of the theoretical time as a function of the number of registers. To test our approach, we performed some numerical experiments. We can remark from Figures 3 and 4 that the optimal r_1 is equal to $r/2$ when n and m are of the same order of magnitude. It is still an open problem, however, to give even an asymptotic expression of the optimal r_1 as a function of r and n/m.

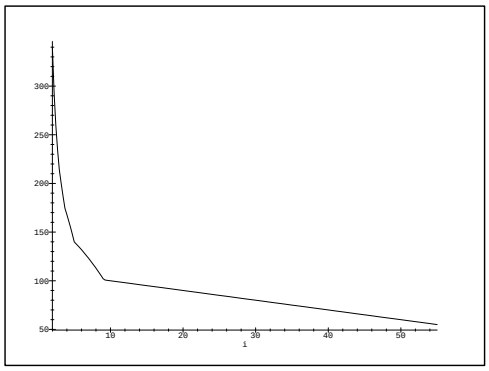


Figure 2: Time $T(r, 55)$ as a function of r registers.

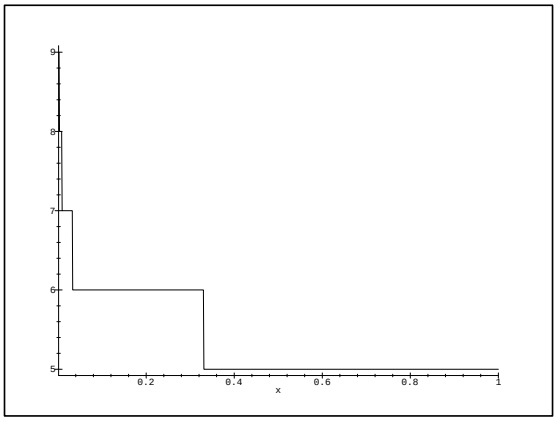


Figure 3: Optimal value r_1 when $x = m/n$ varies from 0 to 1, with $n = 1000$ and $r = 10$.

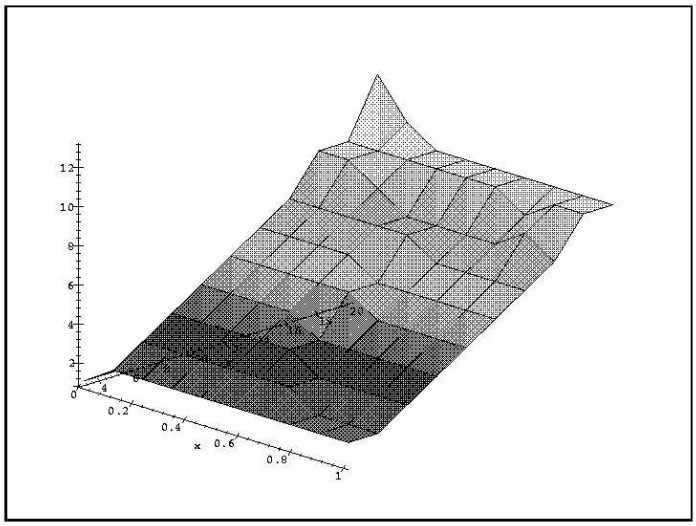


Figure 4: Optimal r_1 when $x = m/n$ varies from 0 to 1, r varies from 3 to 20, with $n = 1000$.

4 Sequences where the Execution Time Is Not Constant

If we suppress the hypothesis that *one application of f costs one unit of time*, we can consider sequences of statements where the computational cost of one application of f

is not constant. In that case, let t_i be the time needed to execute the ith statement or the ith iteration of the loop, and $a_j = \sum_{i \leq j} t_i$. The optimal algorithm is a DC-inversion, defined y by its index q, and its time is:

$$T(r,p) = \min_q a_q + T(r, q-1) + T(r-1, p-q)$$

where $T(r, q-1)$ is the optimal time needed to reverse the sequence $\{i_1, \ldots, i_{q-1}\}$, and $T(r-1, p-q)$ the time needed to reverse the remaining of the sequence with one fewer register.

This is not a practical algorithm, because there are no explicit formulas for $T(r, q-1)$ and $T(r-1, p-q)$, even if t_i is explicitely known (common examples are $t_i = i$ or $t_i = p - i$). If t_i is not explicitely known, (known only after computing the ith statement) computing the exact value of q is impossible without storing all t_i.

The algorithm we propose consists of saying that, if all t_i are nearly the same, then $T(r,p)$ is equivalent to $T_0(r, a_p)$, where T_0 is the quantity defined by Equations 2 and 3. Even though this might be wrong, we hope that q defined by (4) and (5) is a good guess.

To be precise, let E be the set of all q such that (4) and (5) are satisfied, where p is replaced by a_p. Then the index q is such that a_q is in E. Note that the set E may be empty, case where q is either the smallest q such that a_q is greater than $\sup E$, or the greatest less than $\inf E$. In practice, we compute $a = \inf E$, and compute q such that $a_{q-1} < a \leq a_q$.

In the case where the time t_i is not known a priori, we can execute the program once, in order to get a_p, then compute $\inf E$, and compute $x_1, \ldots, x_q$ as long as the accumulated time a_q is less than $\inf E$.

5 Application to Automatic Differentiation

We have applied this strategy to compute in reverse mode the derivatives of a test problem written in Fortran. This test case is representative of a program solving an evolution equation. In this section, our objective is to compare the performances (time and memory space requirements) of three strategies for computing in reverse mode the derivatives of the solution with respect to the control variable. The first strategy is the strategy implemented in Odyssée [Rostaing1993a]. The second strategy consists of applying the DC-strategy previously described, but only to the main loop. In the third strategy, the DC-algorithm is applied to reverse the main loop and the nested loop. We conclude this section with some experimental results.

5.1 Brief Description of the Problem

Considered a Fortran program for solving an evolution equation on the open set $I =]0, 1[$:

$$\frac{\partial y}{\partial t} - Ly = u,$$

where L is the Laplacian, y is the state of the system, and u is the control variable. The initial condition is denoted y_0. On the boundary of I, $y = 0$ (Dirichlet's condition).

To solve this equation, we proceed as follows

- We take $u = 1$ and $y_0 = 0$ constant on I.
- Time resolution scheme is implicit. We take 250 steps in time.
- In space, the classical variational formulation is used. The space is discretized with $np + 2$ points. The length of the intervals is constant.

5.2 Program Organization

The call graph of the original program is presented in Figure 5. The `state` subroutine is composed of two parts:

- Compute the matrix (`matrix` subroutine).
- Run a loop 250 times that computes the second member (`rhs` subroutine) of the linear system and solve it with a Jacobi method (`solve` subroutine).

The `solve` subroutine is also composed of a loop that is executed either until a residual is lower than a value ϵ or 100 times.

```
main--+-state------+-matrix
                   +-rhs
                   +-solve
```

Figure 5: Call graph of the initial Fortran program.

5.3 Automatic Differentiation

5.3.1 Using Odyssée's Standard Strategy

Our program was first differentiated by using the standard reverse mode of the Odyssée [Rostaing1993a] system. Odyssée performs automatic differentiation by program

transformations. Its input is a Fortran program and a list of variables. It returns a new Fortran program for computing the derivatives. In the reverse mode, each generated subroutine is composed of two parts. The first part computes and saves in local variables the values that are modified when the original program is executed. The second part computes the derivatives in reverse mode using the saved values where they are needed. Consequently, when a variable is modified at each step of a loop, all of its values are saved in an array whose size is equal to the number of iterations of the loop.

```
mainad-------statead----+-matrix
                        +-rhs
                        +-solve
                        +-solvead
                        +-rhsad
```

Figure 6: Odyssée strategy.

On our example, using Odyssée, we have obtained the program whose call graph is presented in Figure 6. The `mainad` program calls the `statead` subroutine, where the values of y are computed 250 times and saved in an array of size 250. The subroutine `statead` calls the `solvead` subroutine where y is computed 100 times, and all of its intermediate values are saved in an array of size 100. The `matrix` subroutine is not differentiated because it is constant. The `statead` Fortran code is structured as shown in Figure 7.

5.3.2 Optimal Strategies

We have applied the DC-strategy to the main loop by changing by hand the previous automatically generated program. We replaced the `statead` subroutine by a subroutine that computes the `state` derivatives in reverse mode using the optimal DC-strategy. In the `opt1` program, the `state2` subroutine is the same as the original `state` without the 250 step loop. `statead2` contains only the derivative part of `statead`. The needed recomputation and storage of the y intermediate values are done by the `store_recomp_state` subroutine which implements the DC-strategy.

We then applied the optimal strategy to both loops modifying the y vector: the main loop in the `state` subroutine and the inner loop in `solve`. In the `solve` subroutine, the test on the residual was replaced by a loop of 100 steps, to know *a priori* the number of executed iterations as in the considered model of programs.

```
C Recomputation part
     call matrix
     do i=1, 250
      save rhs_outvars
      call rhs
      save solve_outvars
      call solve
     end do

C Differentiation part
   do i=250, 1, -1
      restore  solve_outvars
      call solvead
      restore rhs_outvars
      call rhsad
     end do
```

Figure 7: Structure of the `statead` subroutine.

```
opt1-----store_recomp_state--+-statead2---+-rhsad
                             |            +-solvead
                             +-compute_state--state2--+-matrix
                                                      +-rhs
                                                      +-solve
```

Figure 8: Optimal strategy on the main loop.

This number of steps is sufficient to find the correct solution to the linear system. The DC-strategy was called to compute the `solve` derivatives, using a handwritten modification of the `opt1` program. The `opt2` call graph is presented in Figure 9. The `opt2` program comes from a modification of the `opt1` program. The `state3` subroutine is the same as `state2` except that it does not call `solve` and save the corresponding intermediate results (as shown in Figure 7). The computation of `solve` is performed by the `store_recomp_solve` subroutine, called by `statead3`. Then `solvead2` contains only the derivative part of `solvead`.

From an implementation point of view, this manipulation of inner loops is more difficult to automate than for the main loop, because it is necessary not only to

```
opt2--store_recomp_state
          |
          +-statead3---+-rhsad
          |            +-store_recomp_solve--+-solvead2
          |                                  +-compute_solve--solve2
          |
          +-compute_state--state3--+-matrix
                                   +-rhs
```

Figure 9: Optimal strategy on the main loop and the solve loop.

modify the subroutine itself but also to replace all of the calls to its derivative subroutine in the whole program by the one implementing the DC-strategy.

5.4 Experimental Results

In the following table, we present some comparisons of the time and the memory requirements for four different Fortran programs. In this table the first column represents:

- **main**: the original program to be differentiated.
- **odyssee**: the standard Odyssée strategy.
- **opt1(10,250)**: the optimal strategy applied to the main loop with ten registers.
- **opt2(5,250) (5, 100)**: the optimal strategy applied to the main loop with five registers and to the nested loop also with five registers. This distribution among the ten registers was chosen because it leads to the minimal time.

The experiments were performed on a SPARC station 10, using Fortran 77 unbundled compiler and the `-O2` option.

Strategy	Time (seconds)	Memory Space (bytes)
main	0.16	1208
Odyssée	1.16	76768
opt1 (10, 250)	1.45	42744
opt2 (5, 250) (5, 100)	3.25	6368

We verified that the time is a function of the number of available registers, as expected from the theoretical results. The evolution of the time with respect to the

number of registers used to reverse the sequence is shown in Figure 10. In that test, the number of registers available for the main loop was kept constant (equal to 10), and we ran the program 100 times, changing at each time the number of registers available for the inner loop. Then $r2$ varied from 1 to 100. The curve obtained corresponds to the result expected from the theory.

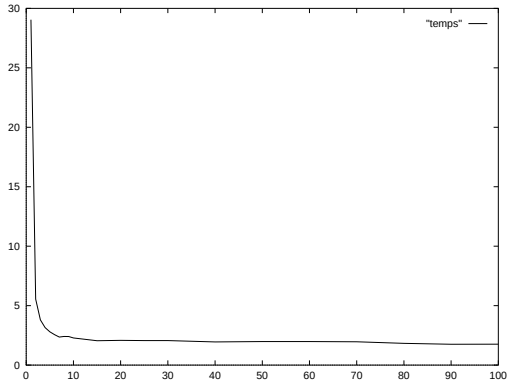


Figure 10: Times in seconds as a function of the number of registers.

We next allowed $r1$ (number of registers available to reverse the main loop) and $r2$ (number of registers for the inner loop) to vary, with the sum being constant $=$ 10. The time in Figure 11 can be read on the vertical axis. A logarithmic scale was chosen because the curve is quite flat between $r1 = 3$ and $r1 = 7$. The minimum time is equal to 3.33 seconds with $r1 = r2 = 5$. The longest time is equal to 33.86 seconds for $r1 = 1$.

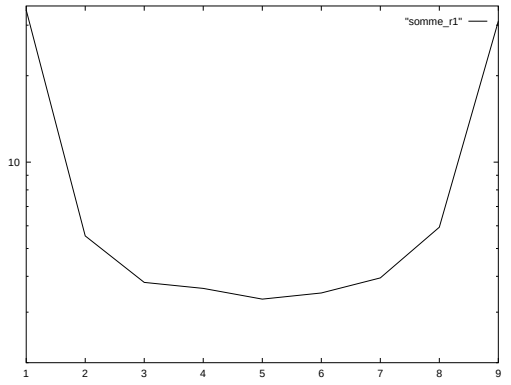


Figure 11: Times in seconds as a function of $r1$.

We can see in Figure 11 that when the lengths of both sequences are of the same order of magnitude, the minimum is reached for $r1 = r2$. Therefore, we performed tests increasing by a factor of 10 the number of iterations in the main loop. The results are presented in Figure 12. As previously, the vertical scale is logarithmic. The minimum is reached for $r1 = 6$, but we notice that from $r1 = 4$ to $r1 = 7$ (i.e., around $r1 = r2$), the curve is quite flat.

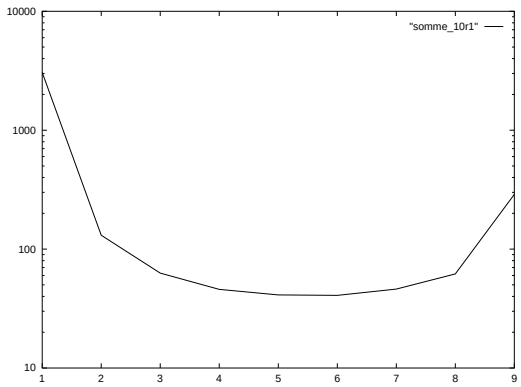


Figure 12: Times in seconds as a function of $r1$.

6 Future Work

In the current version of the Odyssée system the reverse mode is only available with the Standard Strategy, which consists in saving all of the overwritten variables.

The complete automatization of the optimal strategy that we have presented here is under development in the Odyssée system. The user will be able to select a loop in a subroutine and differentiate this loop in reverse mode according to the optimal strategy. He will specify the number of available registers. The selection of the optimal strategy on a loop will imply the automatic generation of four new subroutines:

1. the subroutine which computes and saves the outputs of the initial initial loop.
2. the subroutine computing in reverse mode the derivatives of the body loop.
3. the subroutine which computes all the sequence in reverse mode with the optimal strategy: it calls the two previous subroutines according to the optimal algorithm.

4. the subroutine computing in reverse mode the derivatives of the main subroutine (from which the loop has been extracted), and calling the optimal strategy (the subroutine described in item 3) to compute the loop derivatives.

In a near future, we plan to implement the case where all of the iterations of the loop do not take the same execution time but depend on the step number. The main difficulty comes from estimating the time needed for each element of the sequence and to compute the indices where to cut the sequence for the DC-strategy.

Missing from this study is the case where the number of iterations is not fixed but depends on criteria varying with the input values. This problem can be solved by executing the program once to know how many times the loop is executed and using this information in the DC-strategy.

References

[Abbott1991a] J. Abbott and A. Galligo, *Reversing a finite sequence*, preprint, 1991.

[Baur1983a] W. Baur and V. Strassen, *The complexity of partial derivatives*, Theoretical Computer Science, 22 (1983), pp. 317–330.

[Borodin1993a] A. Borodin, *Time-space tradeoffs (Getting closer to the barrier?)*, in Algorithms and Computation, ISAAC '93, Ng, Raghavan, Balasubramanian, and Chin, eds., Springer-Verlag, Berlin, 1993, pp. 209–220.

[Griewank1992a] A. Griewank, *Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation*, Optimization Methods and Software, 1 (1992), pp. 35–54.

[Morgenstern1985a] J. Morgenstern, *How to compute fast a function and all its derivatives, a variation on the theorem of Baur-Strassen*, SIGACT News, 16 (1985), pp. 60–62.

[Rostaing1993a] N. Rostaing, S. Dalmas, and A. Galligo, *Automatic differentiation in Odyssée*, Tellus, 45A (1993), pp. 558–568.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399