Extension of Odyssée to the MPI Library - Reverse Mode -

Christèle Faure, Patrick Dutto

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Extension of Odyssée to the MPI library - Reverse mode -

Christèle Faure — Patrick Dutto

N° 3774

Octobre 1999

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--3774--FR+ENG

Extension of Odyssée to the MPI library - Reverse mode -

Christèle Faure*, Patrick Dutto†

Thème 1 — Réseaux et systèmes
Projet TROPICS

Rapport de recherche n° 3774 — Octobre 1999 — 35 pages

Abstract: Odyssée is an automatic differentiation (AD) package developed at INRIA and UNSA. This tool is able to differentiate a sequential Fortran 77 code with respect to variables chosen by the user. In order to use Odyssée on parallel codes, the class of tractable programs has been extended. We have restrict ourselves to the differentiation of MPI code, but the same methodology can be applied to PVM or any other message passing library.

An information base has been defined in order to make the system follow the dependencies between variables, in this way the code is properly analyzed and the generated code is correct. A library of derivative of MPI commands has been written for the direct and reverse mode. The user must link these libraries to the part automatically generated to executed it. The implementation of both the information base and the library presented in this documents could be modified by the user to fit his specific needs.

In a previous report ([3]), we have presented the extension of the direct mode of Odyssée to MPI. In this report, we present the application of the reverse mode validated on the same two example as the direct mode. The result of the two phases of this study in an extension of Odyssée which is not robust enough to be distributed now, but will be with the next version.

We have also pointed out a need for a theoretical interpretation of the notion of derivative for a parallel code, mainly for the reverse mode when overlapping of the data is used.

Key-words: Odyssée, MPI, Navier-Stokes equations, automatic differentiation, computational differentiation, reverse mode, message passing, computational fluid dynamics, Fortran 77, program transformation.

This work has been partially funded by the Esprit project DECISION (EP 25 058) and the GENIE program (Association of Dassault-Aviation, Aérospatiale and INRIA).

* Email : Christele.Faure@sophia.inria.fr, URL : http://www.inria.fr/tropics/Christele.Faure
† Email : Patrick.Dutto@sophia.inria.fr

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

Extension d'Odyssée à la librairie MPI
- Mode inverse -

Résumé : Odyssée est un outil de différentiation automatique développé à l'INRIA et à l'UNSA, qui a été appliqué à de nombreux codes séquentiels en milieu industriel. Dans ce rapport, nous présentons son extension aux codes parallélisés utilisant MPI. Nous avons choisi de considérer la librairie MPI comme une librairie externe et de ne pas modifier le noyau principal d'Odyssée. Cette extension consiste en un fichier qui lu avant la dérivation permet à Odyssée de pratiquer une analyse de dépendance complète, et plusieurs fichiers Fortran 77 contenant les définition des dérivées des commandes d'Odyssée qui compilées et linkées au code généré permettent d'exécuter la dérivée. Une première phase de ce travail présentée dans [3] a permis de valider cette extension en mode direct. Ce rapport présente un travail équivalent pour le mode inverse. Comme précédemment, nous avons validé cette extension sur un code exemple permettant une description complète du code généré dans ce rapport (voir Section 2.3), mais aussi sur un code pré-industriel (voir Chapter 3). Cette seconde étude a montré la difficulté d'interprétation des dérivées calculées en mode inverse, par rapport à celle calculées en mode direct.

Mots-clés : Odyssée, MPI, équations de Navier-Stokes, différentiation automatique, mode inverse, envoi de message, mécanique des fluides numérique, Fortran 77, transformation de code

Contents

Chapter 1

Introduction

This report presents the second phase of a general study on the application of automatic differentiation to parallelization[1]. In the first part of this work described in [3], we have analyzed the extension of the direct mode of Odyssée to MPI. In this document, we analyze the extension of the reverse mode of Odyssée to MPI. The parallelization strategy adopted in this study combines `data partitioning techniques` and `message passing programming model`. We have restricted ourselves to the differentiation of MPI code using Odyssée, but the same methodology can be applied to PVM or any other message passing library. In order to use Odyssée on parallel codes, the class of tractable programs has been extended.

For sake of simplicity, we have chosen to treat parallelization commands as an external library for the AD tool Odyssée. An other approach should be to treat MPI commands as intrinsic Fortran commands, but then the kernel of the system would have to be modified (and this should be necessary for all external libraries). In this way this study can be generalized to any parallelization package for message passing. In order to extend the results obtained in this study to any other AD-tool, one has to know first if the management of black-box routines is possible or not. If it is not, the user should have to write down fictitious routines equivalent (in terms of dependencies) to the MPI routines.

Extending Odyssée to any external library can be done without writing any fictitious library, but using the ability of the system to cope with black-box routines using an information base. An information base has been defined in order to make the system follow the dependencies between variables, in this way the code is properly analyzed and the generated code is correct. This information base is the same for the direct and reverse mode, we have described it in our previous report [3]. A library of derivative of MPI commands specific to the reverse mode has been written in order to help the user compile and execute the generated code. This library is described in Chapter 2. The implementation of both the information base and the library presented in this documents could be modified by the user to fit some specific needs.

[1]This work has been partially funded by the Esprit project DECISION (EP 25 058).

We have first validated the adapted version of Odyssée on a sample code that uses data partitioning to compute a polynomial. In Chapter 2.3, we present the results obtained on this example. The size of this example allows us to show in the report the whole code, that is the reason why we describe it. The second part of this work has been to test the adapted version of Odyssée on a pre-industrial code. The results of this study are presented in Chapter 3.

Automatic Differentiation (see [6] and [1]) is a set of techniques for computing derivatives at arbitrary points. Automatic Differentiation is based on two main observations: a program execution can be seen as a composition of functions and can thus be differentiated using the chain rule. The derivatives of elementary instructions are computed using standard rules for differentiating expressions such as: "the derivative of a sum is the sum of the derivatives" ...

Two modes of Automatic Differentiation have been studied and implemented by various authors: the direct (or forward) mode that computes the derivatives and the initial values simultaneously, and the reverse (or backward) mode that computes first the initial values and then the derivatives in reverse order. The reverse mode is particularly efficient for computing gradients because its cost is independent of the number of inputs. Two classes of automatic differentiation Tool exist: those that work by code generation, and those that work by operator overloading. Odyssée, Adifor, GRESS, TAMC belong to the first class of automatic differentiation tools based on code generation.

Odyssée (see [4]) is an automatic differentiation tool developed at INRIA that differentiates Fortran-77 units. The two main features of Odyssée are that it is able to differentiate a function even if some source units have not been read by the system, and that the reverse mode is applicable on operational codes. If a function is implemented as a set of units, Odyssée is able to differentiate it as a whole with respect to the inputs specified by the user. From this set of units, Odyssée generates a new set of units that computes the derivatives. In direct mode Odyssée uses the tangent linear algorithm to generate a Fortran-77 code that computes one directional derivative. In reverse mode, the code generated by Odyssée computes the cotangent code (equivalent to hand written adjoint codes) which is the product of a vector with the transposed Jacobian matrix.

Chapter 2

MPI Derivative library

This section is devoted to the description of the library of the derivatives in reverse mode of the basic MPI commands. To do so we use some equivalent mathematical language, and show how to differentiate this language, and then go pback to MPI.

The simplest function a parallel code may use is sending a data from one processor to another. For example, consider a code in which processor 0 (P0) sends value x to processor 1 (P1) which receives it in the variable y. In the code, this is written in a formal way by commands like `send` and `recv` as shown in Figure 2.1.

Send$(x, P1)$ / Executed on P0

(a) x value sent from P0 to P1

Recv$(y, P0)$ / Executed on P1

(b) y value received on P1 from P0

Figure 2.1: Basics of message passing

2.1 Interpretation of the basic commands

The send command can be interpreted as an assignment of the sent value to a fictitious variable `link` (for link send receive), in the same way the receive command could be interpreted as an assignment of this the value stored in this fictitious variable to a variable.

From this interpretation an equivalent code for Send$(x, P1)$ and Recv$(y, P0)$ can be derived as shown in Figure 2.1. This equivalent code can be differentiated as shown in Figure 2.3, and the derivatives for the send and receive commands can then be formalized as shown in Figure 2.4.

Most of the MPI commands can be differentiated using such a sequential interpretation. The non-blocking commands add one more problem because the schedule of send and receive

$\text{link} = x$ $\qquad$ $y = \text{link}$

(a) $\qquad$ (b)

Figure 2.2: Equivalent code

$$\begin{array}{lcl} \bar{x} & = & \bar{x} + \bar{\text{link}} \\ \bar{\text{link}} & = & 0 \end{array} \qquad \begin{array}{lcl} \bar{\text{link}} & = & \bar{\text{link}} + \bar{y} \\ \bar{y} & = & 0 \end{array}$$

(a) $\qquad$ (b)

Figure 2.3: Derivative code

$\text{Recv}(z, P1)$ / Executed on P0
$\bar{x} = \bar{x} + z$

(a)

$\text{Send}(\bar{y}, P0)$ / Executed on P1
$\bar{y} = 0$

(b)

Figure 2.4: Derivative of basic message passing commands

is known at runtime and not at compile time. So it is impossible to reverse at compile time the order of execution of the send and receive commands, the only way to deal with these commands would be to modify the kernel of Odyssée to have a dynamic structure that store the execution of the message passing command into some pile and to reverse this pile instead of the original code. Such an algorithm has been used to deal with `gotos`. In this study we do not consider the non-blocking commands.

2.2 Derivative library in reverse mode

The derivatives of the MPI commands are gathered into a derivative library written in Fortran 77 (plus MPI) and named `MPI_cl.f`.

From the scheme of derivative of basic message passing command shown below (in Figure 2.4), the derivatives of the `MPI_send` and `MPI_recv` commands can be easily written.

The form of the calls to the derivative of `MPI_send`, automatically generated by Odyssée from the original calls, is shown in Figure 2.5. From these examples, one can see that the derivative routine has the same name as the original routine suffixed by `cl` and takes as arguments the original arguments plus their derivatives suffixed by `ccl`.

From the general scheme presented below and the remark on the general form of the derivative calls generated by Odyssée, the derivatives of the basic point to point communication commands are easy to write. We give in Appendix A.1 and Appendix A.2 the

```
call MPI_send (buf,lbuf,MPI_real,node,tagid,com,ierror)
```

(a) Original call

```
call MPI_sendcl (buf,lbuf,MPI_real,node,tagid,com,ierror,bufccl)
```

(b) Derivative call

Figure 2.5: Call to `MPI_sendcl` automatically generated by Odyssée

$x_q = \sum_{p \in [1..l]} x_p$

(a) Original

$$\begin{aligned} \bar{x}_1 &= \bar{x}_1 + \bar{x}_q \\ \bar{x}_2 &= \bar{x}_2 + \bar{x}_q \\ &\cdots \\ \bar{x}_l &= \bar{x}_l + \bar{x}_q \\ \bar{x}_q &= 0 \end{aligned}$$

(b) Derivative

Figure 2.6: Equivalent code for the `MPI_reduce` command and its derivative

example for the `MPI_send` and `MPI_recv` commands which are really straightforward from the equivalent representation.

If one looks at the collective communication commands such as `MPI_reduce` the problem is a bit more complicated, but is always deduced from the mathematical equivalent representation as shown in Figure 2.6(a). If one looks at the `MPI_reduce` command for the summation, the mathematical equivalent representation could be $x_q = \sum_p x_p$ where p is the label of all the processors where the inputs are stored and q is the label of all processor where to compute the summation. The derivative of the equivalent representation is shown in Figure 2.6(b). As we have already said in [3], the reduce command is associated to many reduction operations and for each of them, the derivative must be written. In Appendix A.3, we show only the the Fortran 77 code written when the reduction operation is the sum.

2.3 Test of the derivative library on a sample example

We have chosen a source code that implement the evaluation of a polynomial in parallel by data partitioning. The root processor sends the data to all the processors, each processor computes one term and the terms are summed using a reduce command.

```
subroutine poly (x,alpha,resu,tag,rank,numtasks)

implicit none
include 'mpif.h'

integer numtasks,rank,ierr,dest,source,stat,tag(3)
real alpha(3),x(3),y,resu

if (rank .eq. 0) then
   do dest=0,2
      call MPI_SEND(x(dest),1,MPI_REAL,dest,tag(dest),MPI_COMM_WORLD,ierr)
   end do
end if

source = 0
call MPI_RECV(x(rank),1,MPI_REAL,source,tag(rank),MPI_COMM_WORLD,stat,ierr)

y = alpha(rank)*x(rank)

call MPI_REDUCE(y,resu,1,MPI_REAL,MPI_SUM,0,MPI_COMM_WORLD,ierr)
return
end
```

Figure 2.7: Source code of the original program

```
# Sets include path
setvar include_path ("~/usr/local/mpi/include/")

# Reads the information basis from MPI.bi
loadibasis MPI

# Reads all the input files
load test.f

# Differentiates poly with respect to x and prints it in the file
#  polycl
diff -cl -vars x -h poly -o polycl
```

Figure 2.8: Source code of batch file

To generate the derivative of such a code, we have used a batch file (presented in Figure 2.8) that can be executed by Odyssée from a makefile.

The derivative is then automatically generated and we show this derivative in Figure 2.9. On can see in the generated code that a lot of storage instructions are meaningless. This is being studied and will be solved within the next version of Odyssée.

The main program that calls the derivative code must be written by hand. It initializes the original variables but also the input direction and prints the output gradient as shown in Figure 2.10.

To compile the source code, the command is:

```
mpif77 -I/usr/local/mpi/include polycl.f MPI_cl.f -o test_poly
```

To execute the code, we use the following script:

```
mpirun -machinefile machines -np 3 test_poly
```

The result of the execution is:

```
 Number of tasks=  3My rank=  0
 Number of tasks=  3My rank=  1
 Number of tasks=  3My rank=  2
 the sum is     26.0000
 partial derivatives are    1.00000    2.00000    3.00000
```

The numerical results of this execution are correct with respect to the input given in the main program (see Figure 2.10). On this baby example, we have shown that the derivation procedure that uses the information base `MPI.bi`, as well as the execution that uses the hand written derivative library are correct.

```
SUBROUTINE POLYCL (X,ALPHA,RESU,TAG,RANK,NUMTASKS,XCCL,RESUCCL)
IMPLICIT NONE
INCLUDE 'mpif.h'

C Initializations of local variables
      YCCL = 0.

C Trajectory
      TEST2 = RANK.EQ.0
      IF (TEST2) THEN
        SAVE5 = DEST
        DO DEST = 0,2
           SAVE3(DEST) = IERR
           CALL MPI_SEND(x(dest),1,MPI_REAL,DEST,TAG(DEST),MPI_COMM_WORLD,IERR)
        END DO
      END IF
      SAVE6 = SOURCE
      SOURCE = 0
      DO NNN1 = 1,3
         SAVE7(NNN1) = X(NNN1)
      END DO
      SAVE9 = IERR
      CALL MPI_RECV(X(RANK),1,MPI_REAL,SOURCE,TAG(RANK),MPI_COMM_WORLD,STAT,IERR)
      SAVE10 = Y
      Y = ALPHA(RANK)*X(RANK)
      SAVE11 = RESU
      CALL MPI_REDUCE(Y,RESU,1,MPI_REAL,MPI_SUM,0,MPI_COMM_WORLD,IERR)

C Transposed linear forms

      RESU = SAVE11
      CALL MPI_REDUCECL(Y,RESU,1,MPI_REAL,MPI_SUM,0,MPI_COMM_WORLD,IERR,YCCL,RESUCCL)
      Y = SAVE10
      XCCL(RANK) = XCCL(RANK)+YCCL*ALPHA(RANK)
      YCCL = 0.
      IERR = SAVE9
      DO NNN1 = 3,1,-1
         X(NNN1) = SAVE7(NNN1)
      END DO
      CALL MPI_RECVCL(X(RANK),1,MPI_REAL,SOURCE,TAG(RANK),MPI_COMM_WORLD,STAT,IERR,XCCL(RANK))
      SOURCE = SAVE6
      IF (TEST2) THEN
        DO DEST = 2,0,-1
           IERR = SAVE3(DEST)
           CALL MPI_SENDCL(X(DEST),1,MPI_REAL,DEST,TAG(DEST),MPI_COMM_WORLD,IERR,XCCL(DEST))
        END DO
        DEST = SAVE5
      END IF
      END
```

Figure 2.9: Automatically generated derivative

```
        program eval_polycl

        implicit none
        include 'mpif.h'

        integer numtasks, rank, ierr, dest, source,i
        integer stat(MPI_STATUS_SIZE), tag(0:2)

        real alpha(0:2)
        real x(0:2), xccl(0:2)
        real resu, resuccl

        x(0) = 10.
        x(1) = 5.
        x(2) = 2.

        alpha(0)=1.
        alpha(1)=2.
        alpha(2)=3.

        tag(0)=0
        tag(1)=1
        tag(2)=2

c initialization of the direction

        resuccl  = 1.
        xccl(0) = 0.
        xccl(1) = 0.
        xccl(2) = 0.


        call MPI_INIT(ierr)
        call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
        call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
        print *, 'Number of tasks=', numtasks, 'My rank=',rank

        call polycl (x,alpha,resu,tag,rank,numtasks,xccl,resuccl)

        if (rank .eq. 0) then
           write (6,*)'the sum is',resu
           write(6,*)'partial derivatives are',xccl
        endif

        call MPI_FINALIZE(ierr)

        end
```

Figure 2.10: Driver for `polycl`

Chapter 3

Derivation of a 3D compressible Navier-Stokes solver (NS3D)

We have tested the information base and the derivative MPI library described previously on a parallel three-dimensional compressible Navier-Stokes solver developed at INRIA. For more details on this code, the reader can refer to [7]

The solver under consideration is a representative subset of an existing industrial code, N3S-MUSCL, which is a three dimensional compressible Navier- Stokes solver.

The conservative form of the Navier-Stokes equations is discretized by using a mixed finite element/finite volume method on fully unstructured tetrahedral meshes. The convective fluxes are computed by means of an upwind scheme which is chosen to be Roe's scheme. Second order spatial accuracy is achieved by using an extension to unstructured meshes of the "Monotonic Upwind Scheme for Conservative Law" (MUSCL) method introduced by Van Leer. A standard Galerkin approximation is used to evaluate the viscous fluxes. The strategy considered for advancing the solution in time makes use of a linearized implicit formulation.

The parallelization strategy combines mesh partitioning techniques and a message-passing programming model. The underlying mesh is assumed to be partitioned into several sub-meshes, each defining a sub-domain. Basically the same code is going to be executed within every sub-domain. The partitioning approach makes use of overlapping mesh partitions (a one tetrahedra wide overlapping mesh partition).

3.1 Structure of the original code

The source code can be described in a really simple manner by its main loop shown in Figure 3.1. You can see from Figure 3.2 that it is much more complicated than that.

We are interested in the derivative of the flux `ce` computed by the routine `computeflux` with respect to the physical state `ua` at each time step.

```
Repeat step = step+1
       flux = ComputeFlux (sol(step-1))
       sol = ComputeSol(flux)
Until step = step_{max}
```

Figure 3.1: Structure of the source code

3.2 Automatic generation of the derivative code

As we have already shown in the previous report for the direct mode [3] and in Section 2.3 for the reverse mode on a sample example, the differentiation of a code using Odyssée is really easy. On `ComputeFlux` the batch file to be executed by Odyssée is:

```
# Set include path
setvar include_path (/usr/local/MPI/include)

# Reads all the input files
loadbatch aero_load

# Reads the information base from MPI.bi
loadibasis MPI

# Differentiate ComputeFlux with respect to ua and print it
# in the file ComputeFluxcl
diff -cl -goto -vars ua -h ComputeFlux -o ComputeFluxcl
```

Using this batch file, the derivative code will be generated in a file called `ComputeFluxcl.f`. We have to use the `-goto` option of differentiation to handle some `goto` instructions which are used within the original code. The differentiation algorithm used by Odyssée is then different from the standard one in the sense that the computation of adjoint variables as well as the storage are dynamic.

The cotangent code of `ComputeFlux` generated using `Odyssée` computes a gradient of the five components of the flux `ce` with respect to the five components $(\rho, \rho u, \rho v, \rho w, E)$ of the state vector `ua` such that:

$$\forall i \in [1..5] \quad \forall j \in [1..n] \quad uaccl(i,j) = uaccl(i,j) + \sum_{k,l} \frac{\partial ce(k,l)}{\partial ua(i,j)} ceccl(k,l)$$

where n is the total number of nodes.

```
para3dbis +- ...
          +- nsc3dm +- submsh +- glisum +- (MPI_allreduce)
                               +- subnme
                               +- tilt +- endcom +- (MPI_finalize)
                                       +- (flush)
                               +- verif +- (flush)
                    +- viscdt +- elapse +- (dclock)
                               +- glblow +- (MPI_allreduce)
                               +- gradfb
                               +- nsyncg +- (MPI_barrier)
                    +- excgrd +- arecwn +- (MPI_recv)
                               +- asedwn +- (MPI_send)
                               +- recwat +- (MPI_wait)
                    +- movmsh +- elapse +- (dclock)
                               +- exccor +- arecwn +- (MPI_recv)
                                         +- asedwn +- (MPI_send)
                                         +- recwat +- (MPI_wait)
                               +- excdsp +- arecwn +- (MPI_recv)
                                         +- asedwn +- (MPI_send)
                                         +- recwat +- (MPI_wait)
                               +- glbsum +- (MPI_allreduce)
                               +- nsyncg +- (MPI_barrier)
                    +- computeflux +- elapse +- (dclock)
                                   +- excflux +- arecwn +- (MPI_irecv)
                                              +- asedwn +- (MPI_send)
                                              +- recwat +- (MPI_wait)
                                   +- fluroe
                                   +- nsyncg +- (MPI_barrier)
                                   +- vcurvm
                    +- improe
                    +- jacobi +- elapse +- (dclock)
                               +- excrhs +- arecwn +- (MPI_recv)
                                         +- asedwn +- (MPI_send)
                                         +- recwat +- (MPI_wait)
                               +- glbsum +- (MPI_allreduce)
                               +- nsyncg +- (MPI_barrier)
                    +- excsol +- arecwn +- (MPI_recv)
                               +- asedwn +- (MPI_send)
                               +- recwat +- (MPI_wait)
                    +- resu3d +- getcor +- nrecwn +- (MPI_get_count)
                                                   +- (MPI_recv)
                                         +- nsedwn +- (MPI_send)
                               +- getsol +- nrecwn +- (MPI_get_count)
                                                   +- (MPI_recv)
                                         +- nsedwn +- (MPI_send)
                               +- glbhig +- (MPI_allreduce)
                               +- glblow +- (MPI_allreduce)
RR n° 3774                     +- glisum +- (MPI_allreduce)
```

Figure 3.2: Call graph of the source code

3.3 Hand writing of the driver

The reverse mode of automatic differentiation is appropriate to compute gradients. For sake of simplicity, let say one wants only one partial derivative, for example the derivative of `ce(2,34)` with respect of `ua(1,299)`. We recall that if the original function computes `ce` with respect to `ua`, the cotangent code will compute `uaccl` with respect to `uaccl, ceccl`. The driver to be written must :

1. set the input direction `ceccl` to 1 for the component $(2, 34)$ and 0 for all the other components,
2. call the cotangent code of `ComputeFlux`,
3. print the output direction `uaccl`.

As in this example, we want to use the original loop to generate several inputs for the driver, we must recall the original function at the end to go from one step of the loop to the next one.

The program is then able to compute the local value of one partial derivative which is:

$$\frac{\partial ce_p(2,34)}{\partial ua_p(1,299)}$$

where ce_p, ua_p are the variables computed by the processor p.

```
subroutine driver (tnul,in_file,out_file)

include 'common_diff.h'
integer in_file(0:3), out_file(0:3)
real*8 tnul
integer i, j

do i=1, 5
   do j=1, nsmax
      ceccl(i,j) = 0.
      dxccl(i,j) = 0.
      dyccl(i,j) = 0.
      dzccl(i,j) = 0.
      uaccl(i,j) = 0.
  end do
end do

do i=1, 5
   do j=1, nsmaxg
      unccl(i,j) = 0.
  end do
end do

CALL setdirection (ceccl,5,nsmax,2,34)
CALL computefluxcl(tnul)

CALL computeflux(tnul)
end
```

Figure 3.3: Driver for the cotangent code of `ComputeFlux`

Chapter 4

Validation of the derivatives of NS3D

To validate the results we have used the `scalar product test` to have a global validation. This test checks the coherency of the cotangent code with the tangent code by using the associativity of the product. In our previous report, we have shown that the tangent code was correct by comparison to an approximation computed by finite differences. In those tests, our hypothesis is that the tangent code is correct.

4.1 Parallel scalar product test

Let consider a scalar function that computes y from x. On this example, the `scalar product test` (see [2]) consists of comparing on one side $yttl * yttl$ and on the other side $xttl * xccl$. We call misfit the difference between those two quantities, which must be zero

$$\text{misfit} = yttl * yttl - xttl * xccl.$$

From this scalar test, it is easy to extrapolate the vectorial test on a set of components (denoted $\mathcal{C}$):

$$\sum_{i \in \mathcal{C}} yttl_i * yttl_i = \sum_{i \in \mathcal{C}} xttl_i * xccl_i.$$

On a parallel code where `data partitioning` is implemented, each global data is divided into local data distributed on a set of processors (denoted $\mathcal{P}$), one must then write a new scalar product like this :

$$\sum_{p \in \mathcal{P}} \sum_{i \in \mathcal{C}} yttl_i^p * yttl_i^p = \sum_{p \in \mathcal{P}} \sum_{i \in \mathcal{C}} xttl_i^p * xccl_i^p.$$

From this, one knows if the tangent and cotangent codes are coherent by looking at the difference:

$$\text{misfit}^p = \sum_{p\in\mathcal{P}} \sum_{i\in\mathcal{C}} yttl_i^p * yttl_i^p - xttl_i^p * xccl_i^p.$$

This computation can be implemented in several manner depending on the way associativity of the sum is treated. In order to delay cancelation, the latest the difference is computed, the best the numerical result will be. We have chosen to compute on each processor p:

$$\begin{aligned} \text{left}^p &= \sum_{i\in\mathcal{C}} yttl_i^p * yttl_i^p \\ \text{right}^p &= \sum_{i\in\mathcal{C}} xttl_i^p * xccl_i^p \end{aligned}$$

then to compute on processor 0 the sum over all the processors of left and right separately:

$$\begin{aligned} \text{total}^0_{\text{left}} &= \sum_{p\in\mathcal{P}} \text{left}^p \\ \text{total}^0_{\text{right}} &= \sum_{p\in\mathcal{P}} \text{right}^p \end{aligned}$$

and finally to compute the misfit

$$\text{misfit}^0 = \text{total}_{\text{left}} - \text{total}_{\text{right}}.$$

In order to get results that can be compared, we compute the misfit relatively to the maximum scalar product which is:

$$\text{misfit} = \frac{\text{total}_{\text{left}} - \text{total}_{\text{right}}}{Max(\text{total}_{\text{left}}, \text{total}_{\text{right}})}.$$

The code that implements this algorithm using `MPI` is shown in Figure 4.1.

4.2 Validation of the derivatives

The function `ComputeFlux` computes from a local (partitioned) value of the solution ua_p on each processor p, the global (non partitioned) value of the flux ce. On must notice that the original function `ComputeFlux` is the composition of two sub-functions `computeflux` and `gather` as shown in Figure 4.2 for three processors. The `gather` function does no computation but uses message passing whereas `computeflux` executes the computation local to each processor.

In order to validate the total derivative of `ComputeFlux`, we have validated `computeflux`, `gather` and `ComputeFlux`. The application of the scalar product test to the total function is

```
left = 0.
right = 0.
C     summation of ps1 and ps2 on the local components
do i=1,n
   left = left + yttl(i)*yttl(i)
   right = right + xttl(i)*xccl(i)
end do
C     summation of left on all the processors
call MPI_REDUCE (left,left_total,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,MPI_COMM_WORLD,IERR)

C     summation of left on all the processors
call MPI_REDUCE (right,right_total,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,MPI_COMM_WORLD,IERR)

C     difference of both scalar products relative
misfit = (left_total-right_total)/max(left_total,right_total)
```

Figure 4.1: Parallel misfit function

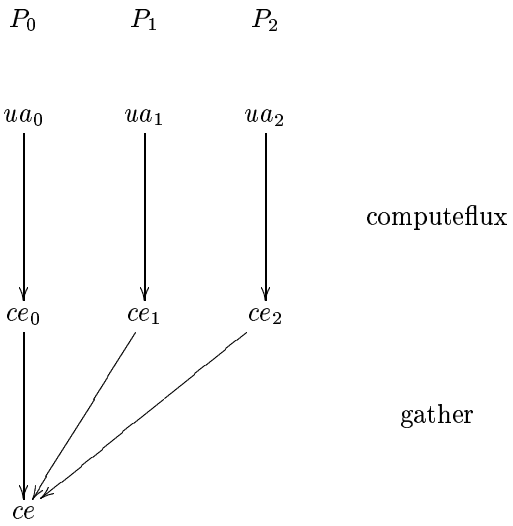


Figure 4.2: Description of `ComputeFlux`

iteration	gather	computeflux	ComputeFlux
1	0.00E+00	0.12E-14	0.00E+00
2	0.00E+00	0.59E-15	0.35E-15
10	0.35E-15	0.14E-14	0.35E-15
20	0.17E-15	0.20E-15	0.88E-15
30	0.70E-15	0.19E-14	0.35E-15
40	0.53E-15	0.79E-15	0.35E-15
47	0.00E+00	0.39E-15	0.00E+00

Table 4.1: Misfit table (double precision)

presented in Figure 4.3. The other codes are really the same and are not shown in this report. The evaluation of the misfit function requires the execution of the tangent and the cotangent codes on the same inputs for the function. In the driver they are stored and restored using `store-state` and `restore-state`. The derivative inputs for the tangent code are chosen arbitrarily, and the cotangent derivative inputs are the output of the tangent code.

For some iterations of the main loop arbitrarily chosen, Table 4.1 shows that the error in the relative misfit between the tangent and cotangent codes is at most E-14. The computations are performed in double precision, and one knows that in double precision zero is E-15. In the table, one can see that the misfit values are at all iterations nearly zero, we even get in some cases the 0.00E+00 value.

4.3 Analysis of the tests

The application of the scalar product test that has been applied on this code gives some really nice results. The tangent and cotangent codes coincide at the precision machine level, which is really difficult to get on a large code. We can conclude from these tests that the derivative code automatically generated as well as the derivative library hand written have been validated on a general code based on the SPMD method.

To interprete the derivative computed by the tangent and cotangent codes, we try to compute using both codes the same partial derivative:

$$\frac{\partial ce(2,34)}{\partial ua(1,299)}$$

where $(1,299),(2,34)$ are the global indexes of the components.

As we have said before, the function `ComputeFlux` computes from a local (partitioned) value of the solution ua_p on each processor p, the global (non partitioned) value of the flux ce. But one must notice that the code by itself does not `split` the global value ua (known by the master processor P_0) into its local values ua_0, ua_1, ua_2 (on all the processors P_0, P_1, P_2).

```
      subroutine driver (tnul)

      include 'common_diff.h'
      include 'mpif.h'
      integer i, j
      real*8 tnul

C Initialization of all the derivatives to zero

C Stores the input of the function
      CALL store_state()

C Computes the tangent code
      do i=1, 5
         do j=1, nsmax
            uattl(i,j) = 1.
         end do
      end do
      CALL computefluxtl(tnul)

C Restores the input of the function
      CALL restore_state()

C Computes the cotangent code
      do j=1, nsmax
         do i=1, 5
            ceccl(i,j) = cettl(i,j)
         end do
      end do
      CALL computefluxcl(tnul)

C Computes and prints the misfit value
      misfit = misfit (5,nsmax,cettl,uattl,uaccl)
      write (6,115) misfit

C Computes the original function
      CALL restore_state()
      CALL computeflux(tnul)

115   FORMAT((e30.20,1x))
      end
```

Figure 4.3: Misfit of the tangent/cotangent code of `computeflux`

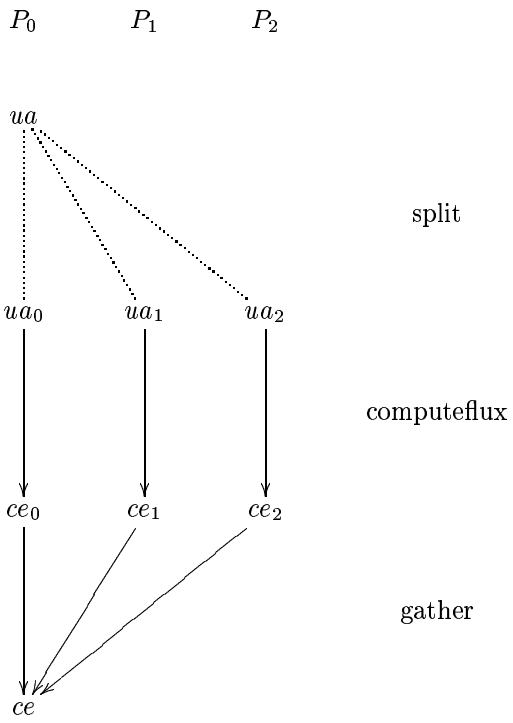


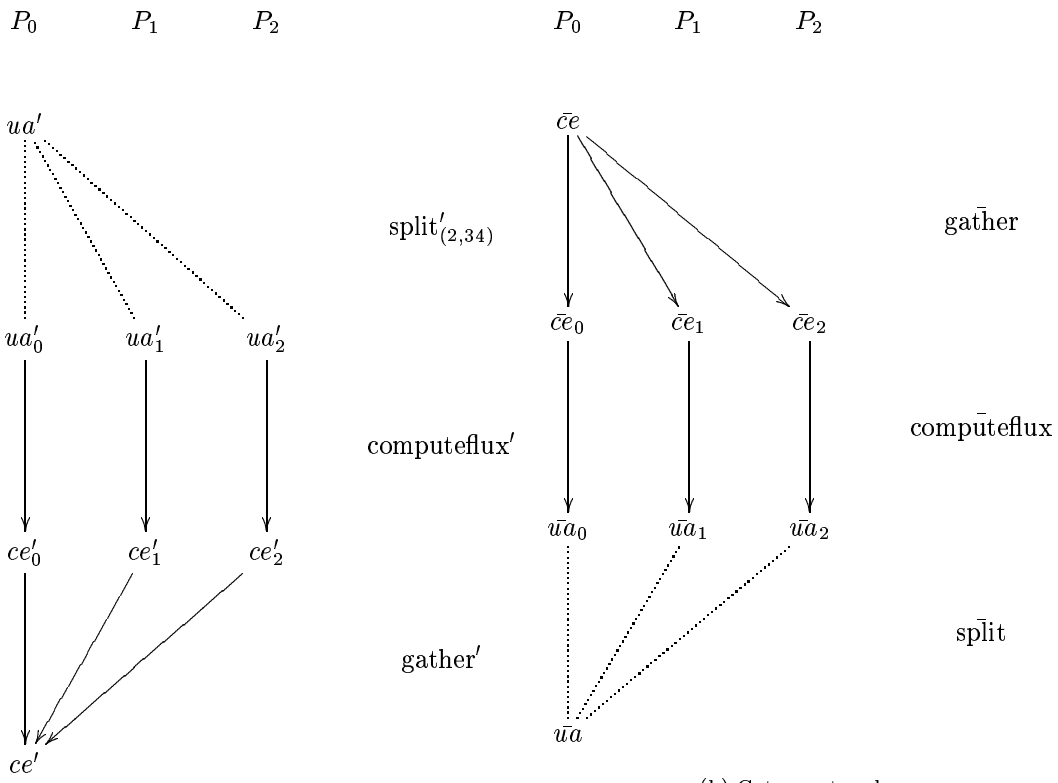
Figure 4.4: Description of `ComputeFluxTotal`

The code `ComputeFluxTotal` that computes the global value of ce from the global value ua is the composition of `computeflux`, `gather` and `split` as shown in Figure 4.4. The only interpretation one can give of a parallel code is the global computation obtained on the master processor.

Wa have studied the basic information necessary to write the `split` function: the function that maps the global indexes to the local ones (see Table 4.2). In this figure, the value *void* means that the global indexes has no corresponding value on the processor.

Global index	Local indexes		
	P_0	P_1	P_2
$(2, 34)$	$(2, void)$	$(2, 8)$	$(2, 805)$
$(1, 299)$	$(1, void)$	$(1, 77)$	$(1, 804)$

Table 4.2: Map function from global indexes to local indexes



(a) Tangent code

(b) Cotangent code

Figure 4.5: Derivative codes of `ComputeFluxTotal`

$$\forall p \in \mathcal{P} \quad \forall J \in \mathcal{G} \quad \forall i \in [1..5] \quad ua_p(i,j) = ua(i,J) \quad \text{where } j = \text{map}(J)$$

Figure 4.6: Split function: $ua \rightarrow \{ua_p\}$

$$\forall p \in \mathcal{P} \quad \forall J \in \mathcal{G} \quad \forall i \in [1..5] \quad ua'_p(i,j) = ua'(i,J) \quad \text{where } j = \text{map}(J)$$

Figure 4.7: Tangent split function: $ua' \rightarrow \{ua'_p\}$

P_0	P_1	P_2		Global
$uattl_0(1, void)$	$uattl_1(1, 77)$	$uattl_2(1, 804)$		$cettl(2, 34)$
$void$	1.	1.	$\Longrightarrow$	0.12663781784448710499

Table 4.3: Derivatives in direct mode

$$\begin{array}{llll} \forall J \in \mathcal{G} & \forall i \in [1..5] & \bar{ua}(i,J) = \sum_{p \in \mathcal{P}} \bar{ua}_p(i,j) & \text{where } j = \text{map}(J) \\ \forall p \in \mathcal{P} & \forall i \in [1..5] & \bar{ua}_p(i,j) = 0. & \end{array}$$

Figure 4.8: Cotangent split function: $\{\bar{ua}_p\} \rightarrow \bar{ua}$

The split function divides the input vector (ua in the example) into one output vector ua_p for each processor $p \in \mathcal{P}$ such that the coherency constraint shown in Figure 4.6 is full-filled. This constraint is described at the component level, for each global component of $\mathcal{G}$.

In direct mode, the derivative of the split function applied on ua does split the derivative ua' as shown in Figure 4.7.

We have been able to write by hand a specific $\text{split}'_{(2,34)}$ function that does not split the input of the function ua_0, ua_1, ua_2, but only splits the corresponding directions ua'_0, ua'_1, ua'_2. One must notice that this $\text{split}'_{(2,34)}$ function is really specific and only splits variables that contains 1 on the global component $(2, 34)$ and 0 elsewhere. In this way, the composition of this specific function $\text{split}'_{(2,34)}$ with the tangent code of `ComputeFlux` (described in Figure 4.5(a)) computes a vector of partial derivatives in which we only pick one partial.

In reverse mode, the derivative of the split function is defined as shown in Figure 4.8 We have not written this derivative $\bar{split}$, but we know from the formulation presented in Figure 4.8 how to compute the resulting derivative from the $\{\bar{ua}_p\}$. The graphical represen-

Global		P_0	P_1	P_2
$ceccl(2,34)$		$uaccl_0(1,void)$	$uaccl_1(1,77)$	$uaccl_2(1,804)$
1.	$\Longrightarrow$	$void$	0.12663781784448682743	0.

Table 4.4: Derivatives in reverse mode

tation Figure 4.5(b) shows how to compose this non implemented part with the implemented ones. On the example, the $\bar{split}$ function would add the $uaccl_1(1,77)$ and $uaccl_2(1,804)$.

From Table 4.3 and Table 4.4, one can verify that the value of the partial derivative computed by both codes at the first iteration is:

$$\frac{\partial ce(2,34)}{\partial ua(1,299)} = 0.12663781784448$$

Figure 4.9 shows the logarithmic value of the relative difference between the partial derivative computed by the tangent and cotangent codes all along the simulation (47 iterations). From this figure one can see that both values have at least 11 common digits and at most 14 common digits.

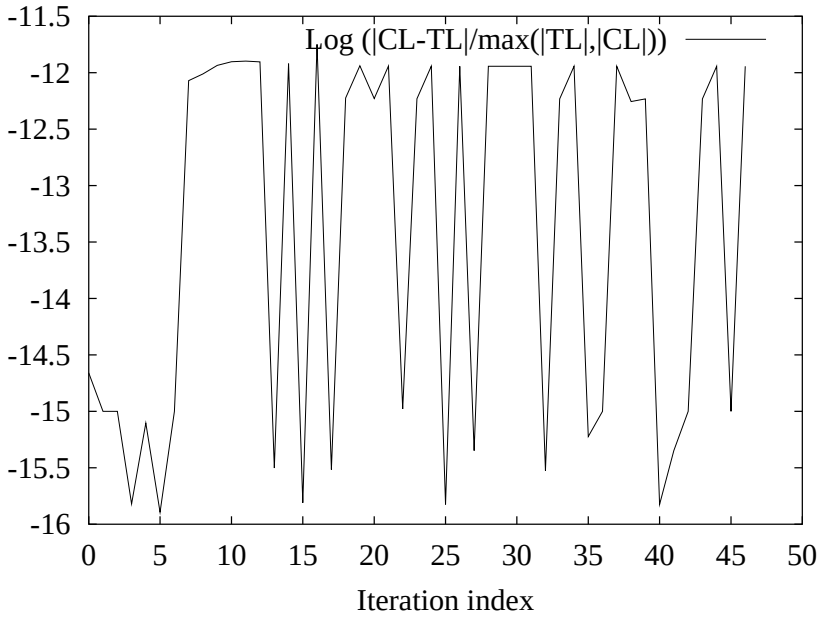


Figure 4.9: Logarithmic difference between the tangent TL and cotangent CL values

Chapter 5

Conclusion

The study described in the report is the second phase of a global study on the extension of an AD tool to a message passing library. In our study, we have chosen Odyssée and MPI for sake of simplicity.

During the first phase, we have tuned Odyssée to make it handle MPI commands in direct mode. This has been done without modifying the kernel of the system, but by declaring MPI to be an external library, and defining an information base for all the commands. A general (under optimal) strategy has been adopted for the differentiation: each message passing command is active. Using this strategy the system has been able to differentiate automatically original codes containing MPI commands. We have also defined the derivatives (in direct mode) of all potentially active commands within an MPI derivative library. We have seen that at least two strategies for passing the original values as well as the derivative values can be adopted. This leads to two different libraries. We have also noticed that the source code (the variable declaration) must be modified depending on the MPI_type of the values passed. We have tested the adapted version of Odyssée on a sample code that uses data partitioning to compute a polynomial as well as on a pre-industrial code (NS3D). On the sample code, we knew the exact value of the partial derivatives, so the validation of the derivatives was trivial. For the large code NS3D, the validation has been performed by comparison to precise finite differences. The Report [3] presents extension of Odyssée and the results obtained on these examples.

During the second phase, we have extended Odyssée to MPI for the reverse mode. We have used the information base defined during the first one, as well as the differentiation strategy. Those basics must be the same in direct and reverse mode. The differentiation of the code was then straightforward from the first phase. We have studied how the derivative library had to be written. This part of the work was non trivial and we have used some work presented in [5]. At the stage, the problem was (as usually in reverse mode) to check the correctness of the derivatives. The hypothesis was that the tangent code was correct (proven in the first phase), we have then decided to use the scalar product test. This test computes the misfit between the tangent and cotangent code. It is based on the computation of two

scalar products which leads to numerical errors. We have studied a precise scalar product that can be applied for a parallel code. We have tested this extension of Odyssée on the same codes as in the first phase. The sample code was really easy to validate as we knew the values of the partial derivatives. The second code was validated using the scalar product, and the exactness of the derivatives was demonstrated. The misfit was at the level of the machine precision which is numerically zero.

The result of this study is an extension of Odyssée to the MPI library. It consists in some files specific to MPI: and information base, and two derivative libraries (direct/reverse mode) where the derivatives of the MPI commands are written using various strategies. This extension must be used in connection with Odyssée, but any future version of the software will do. The robustness of the derivative libraries could be improved as well as its efficiency. We have pointed out during the first phase that the differentiation was in some sense maximal using this extension because all the passed messages where active (associated with derivative). The generated code was not optimal in the number of derivatives computed. This problem is difficult to solve without modifying the kernel of Odyssée. The second phase has shown an other problem: the generated code is not optimal in the storage necessary for the trajectory. This problem is general and is studied in some other context. Using the future versions of Odyssée with this MPI extension will solve this problem. Another problem arised during this phase, this is the treatment of the non-blocking commands. The problem is that the schedule of send/receive/barrier is known at runtime and not at compile time. So it is impossible to reverse the execution from the original code. The only way to deal with these commands would be to modify the kernel of Odyssée to have a dynamic structure that stores the execution of the message passing commands into some pile and to reverse this pile instead of the original code. This extension of Odyssée does not deal with non-blocking commands.

The methods and choices described in this report have been validated in the Odyssée/MPI context, but could be the base of some extension of Odyssée to other parallel libraries and even to other automatic differentiation tools.

The interpretation of the derivatives computed on a parallel code is to be studied. The interpretation of the direct mode follows exactly the interpretation of the original code, but the interpretation of the derivatives in reverse mode is really a problem. Indeed, the function that splits the initial data to supply each processor with its inputs does not belong to the source code to be differentiated. This problem depends neither on the AD tool, nor on the parallel library, but is related to the parallelization strategy. A general methodology to interpret the derivatives should be studied for each parallelization strategy.

Appendix A

Source code of the derivative library

A.1 Derivative of the MPI_send command

```
SUBROUTINE MPI_RECVCL(rbuff,lgbuff,type,irproc,idtype,comm,status,ierror,rbuffccl)

implicit none
INCLUDE 'mpif.h'

INTEGER  lgbuff, type, comm, idtype, ierror, status(MPI_STATUS_SIZE)
INTEGER irproc, i
REAL*8  rbuff(lgbuff), rbuffccl(lgbuff)

CALL MPI_SEND(rbuffccl, lgbuff,type,irproc,idtype,comm,ierror)
do i = 1, lgbuff
   rbuffccl(i) = 0.
end do

END
```

A.2 Derivative of the MPI_recv command

```
SUBROUTINE MPI_SENDCL(sbuff,lgbuff,type,isproc,idtype,comm,ierror,sbuffccl)

implicit none
```

```
INCLUDE 'mpif.h'

INTEGER lgbuff, type, comm, idtype, ierror, status(MPI_STATUS_SIZE)
INTEGER isproc, i, bufmax
PARAMETER (bufmax = ??)
REAL*8  rbuff(lgbuff), rbuffccl(lgbuff), buffer(bufmax)

CALL MPI_RECV(buffer,bufmax,type,isproc,idtype,comm,status,ierror)

do i= 1,lgbuff
   sbuffccl(i) = buffer(i) + sbuffccl(i)
enddo

END
```

A.3 Derivative of the `MPI_reduce` command

```
SUBROUTINE MPI_REDUCECL(sbuff, recvbuff, count, datatype,op, root,comm,
                        ierror, sbuffccl, recvbuffccl)

implicit none
INCLUDE 'mpif.h'

INTEGER comm, op, root, count, datatype, i, ierror
REAL*8  sbuff(count), recvbuff(count), sbuffccl(count), recvbuffccl(count)

if (op .eq. MPI_SUM) then
   call MPI_BCAST(recvbuffccl, count, datatype,root,comm, ierror)
   do i = 1,count
      sbuffccl(i) = sbuffccl(i) + recvbuffccl(i)
   end do
else
   write(6,*) ' non implemented operation'
endif
END
```

Bibliography

[1] M. Berz, C.H. Bischof, G.F. Corliss, and A. Griewank, editors. *Computational Differentiation: Applications, Techniques, and Tools.* SIAM, Philadelphia, 1996.

[2] I. Charpentier and M. Ghemires. Efficient adjoint derivatives: Application to the atmospheric model meso-nh. *Optimization Methods and Software*, 1999. To appear.

[3] C. Faure and P. Dutto. Extension of Odyssée to the MPI library -Direct mode-. Rapport de recherche 3715, INRIA, June 1999.

[4] C. Faure and Y. Papegay. Odyssée User's Guide. Version 1.7. Rapport technique 0224, INRIA, September 1998.

[5] C. (Ed.) Faure. Automatic differentiation for adjoint code generation. Rapport de recherche, INRIA, November 1998.

[6] A. Griewank and G.F. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Applications.* SIAM, Philadelphia, 1991.

[7] S. Lanteri. Parallel Solutions of Three-Dimensional Compressible Flows. Research report 2594, INRIA, June 1995.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399