INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Extension of Odyssée to the MPI library - Direct mode -

Christèle Faure, Patrick Dutto

N° 3715

Juin 1999

THÈME 2

ISSN 0249-6399

INRIA
SOPHIA ANTIPOLIS

Extension of Odyssée to the MPI library - Direct mode -

Christèle Faure*, Patrick Dutto†

Thème 2 — Génie logiciel
et calcul symbolique
Projet TROPICS

Rapport de recherche n° 3715 — Juin 1999 — 44 pages

Abstract: Odyssée is an automatic differentiation (AD) package developed at INRIA. 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 in order to help the user compile and execute the generated code. 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.

We have tested the adapted version of Odyssée on a sample code that uses data partitioning to compute a polynomial. In the Appendix A, we present the results obtained on this example. The size of this example allows us to show in the report the whole code. We have also tested the adapted version of Odyssée on a pre-industrial code to show the feasibility. The results are presented in Chapter 4.

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

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

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 direct -

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 une extension possible 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. Nous avons testé cette extension sur un code exemple permettant une description complète du code dans ce rapport (voir annexe A). Nous l'avons aussi testé sur un code pré-industriel pour tester la faisabilité de la méthode sur un cas réel. Les résultats de ces tests sont présentés chapitre 4.

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

[§] Ce travail a été partiellement financé par le projet ESPRIT DECISION (EP 25 058) ainsi que par le programme GENIE de Collaboration entre Dassault-Aviation, Aérospatiale et l'INRIA.

Contents

Chapter 1

Introduction

In this report, we present an application of automatic differentiation to parallelization. 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[1].

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. In this study, we have restricted ourselves to the direct mode of automatic differentiation in order not to get into transposition and storage problems.

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 is described in Chapter 2. A library of derivative of MPI commands has been written in order to help the user compile and execute the generated code. This library is described in Chapter 3. 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.

We have tested the adapted version of Odyssée on a sample code that uses data partitioning to compute a polynomial. In the Appendix A, we present the results obtained

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

on this example. The size of this example allows us to show in the report the whole code. We have also tested the adapted version of Odyssée on a pre-industrial code to show the feasibility. The results are presented in Chapter 4.

1.1 Automatic Differentiation

Automatic Differentiation (see [3] 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 [2]) 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.

1.2 External library for Odyssée

The approach we have chosen for sake of simplicity is 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.

In this way this study can be generalized to any parallelization package for message passing and any AD tool by source transformation. To get the same behavior from any 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. That is what is simulated using the information base in Odyssée.

An AD tool can differentiate a code that contains parallelization commands if it is able to:

1. follow the transmitted variables through the commands,
2. propagate "activity" through this dependency graph,
3. generate calls to derivatives of those commands.

In order to take these actions, the system must be able to analyze the routines of the library as well as the user routines. In this case the library (MPI or PVM) is not written in Fortran 77 but the code is written in Fortran 77 so the library and the user's code can not be analyzed within the same AD tool.

One important feature of Odyssée is a facility for the user to incorporate routines that can not be analyzed by the system. We have used this feature as a base of this study. The user must provide Odyssée with a description of each routine whose code is not available. Otherwise, the system will generate default informations considering that: all dummy arguments are input and output, and each output depends on all input. By default the system considers that no global variable is read or written by a black-box routine.

The Odyssée commands (see [2] for more details) for describing black-box routines are:

setdummys `unit var_list` Sets the list of the arguments of the unit of name `unit` to the list of variables `var_list`.

setglobals `unit var_list` Sets the list of the global variables of the unit of name `unit` to the list of variables `var_list`.

setinout `unit var_list_1 var_list_2 var_list_3 var_list_4` Sets the lists of the input and output dummies and global variables of the unit of name `unit`.

The list of input `dummies` is set to the list of variables `var_list_1`, the list of output `dummies` to the list `var_list_2`, the list of input `global` to the list `var_list_3`, and the list of output `global` to `var_list_4`.

As a side effect, `setinout` sets the dependencies between variables to their default value: each output variable depends on all the input variables.

setabstract `unit var_list_list` Sets the lists of the dependencies between the input and the output variables of the compilation unit of name `unit`.

The Figure 1.4 shows the differences between the derivatives generated by Odyssée on a simple code shown in Figure 1.1 using or not some user known information on `g`. In the code shown in Figure 1.4(a) with the default information for `g`, a derivative is associated with `y` as it may depend on `x` through the call to `g`, and then `k` becomes also active. In the code shown in Figure 1.4(b) generated using the information declared in Figure 1.2, `y` and

```
subroutine f (x,y,z,k)
real x,y,k
real l1,l2
integer z

l1 = x+y
l2 = x*y
call g (l1,l2,x,y,z)
k = y**2
return
end
```

Figure 1.1: Source code of `f`

```
setdummys g (g1 g2 g3 g4 g5)
setinout g (g1 g2 g3 g4) (g3 g5) () ()
setabstract g (g5 (g1 g2 g3 g4)) (g3 (g1 g2 g3))
```

Figure 1.2: Information base for g

```
# loads the source code
          load f
# differentiate f
          diff -h f
#              in tangent mode
               -tl
#              with respect to x
               -vars x
```

(a) without information base

```
# loads the information base for g
          loadibasis BI_g
# loads the source code
          load f
# differentiate f
          diff -h f
#              in tangent mode
               -tl
#              with respect to x
               -vars x
```

(b) with information base

Figure 1.3: Differentiation of `f` with respect to `x`

`k` are not active because the third and forth arguments of g are no more dependent through the call of g.

In order to compile and run the program using `ftl`, the user will have to provide the source (or compiled) code for `gtl` that could not be generated by `Odyssée`.

```
COD Active IN   dummys:  x
COD Active OUT  dummys:  k x y
COD Dependencies between IN and OUT:
COD k <--   x
COD x <--   x
```

```
SUBROUTINE ftl (x,y,z,k,xttl,yttl,kttl)

IMPLICIT NONE

REAL x,y,k
INTEGER z
REAL l1,l2
REAL xttl,yttl,kttl
REAL l1ttl,l2ttl

l1ttl = xttl
l1 = x+y
l2ttl = y*xttl
l2 = x*y
CALL gtl(l1,l2,x,y,z,
         l1ttl,l2ttl,xttl,yttl)
kttl = 2*yttl*y
k = y**2
END
```

(a) without information base

```
COD Active IN   dummys:  x
COD Active OUT  dummys:  x z
COD Dependencies between IN and OUT:
COD x <--   x
```

```
SUBROUTINE ftl (x,y,z,k,xttl)

IMPLICIT NONE

REAL x,y,k
INTEGER z
REAL l1,l2
REAL xttl
REAL l1ttl,l2ttl

l1ttl = xttl
l1 = x+y
l2ttl = y*xttl
l2 = x*y
CALL gtl(l1,l2,x,y,z,
         l1ttl,l2ttl,xttl)

k = y**2
END
```

(b) with information base

Figure 1.4: Derivation of `f` with respect to `x`

1.3 Classification of MPI commands

Among the MPI commands (see [5] for an exhaustive description), one can distinguish two sets of routines; constant commands and potentially active commands.

The constant routines do not pass value, they will never be active and do not have an associated derivative routine in the library. Some MPI commands are only dealing with the parallelization environment of the program and do not compute any real value, they are then constant for the differentiation process. For example, the `MPI_init`, `MPI_finalize` commands belong to this first class.

The potentially active routines have only one possible active pattern, it is then easy to defined one derivative for each. For example, `MPI_send` , `MPI_recv` and `MPI_reduce` are potentially active routines.

The potentially active commands can be separated into two sub-classes:

linear routines In this context, we call linear commands the commands which result (even if it is only side-effect) depend linearly on the real input. For example, the `MPI_send` and `MPI_recv` commands seen as assignment of global variables are linear.

potentially non linear routines On the contrary, the command `MPI_reduce` can be linear if the operation performed is `MPI_sum`, but is not if the operation is `MPI_prod`. The `MPI_reduce` commands is then considered as a non linear command.

1.4 Conclusion

As we have decided to treat the message-passing library as an external library, we have not modified the kernel of Odyssée. We have firstly defined an information base for the MPI commands and secondly hand written the derivative library for those commands.

Chapter 2

MPI Information base

In order to follow the variables influence through the code, and particularly through call to library commands, one has to describe those commands within an information base. In this section, we mainly describe various ways of writing this information base.

The information base is a sequence of call of Odyssée commands which are gathered in one file named MPI.bi. Each MPI command is described by some Odyssée commands described in Section 1.2. For the **constant commands**, there is no problem of definition as they have no input nor output in the sense of differentiation. The **potentially active commands** must be defined such that the propagation of active variable can be correctly performed.

2.1 Constant commands

Constant commands do not pass any value, but define the context of execution of the parallel code. In this category, we may list the following commands:

```
setdummys MPI_init (ierror)
setinout MPI_init () () () ()

setdummys MPI_finalize (ierror)
setinout MPI_finalize () () () ()

setdummys MPI_initialized (flag ierror)
setinout MPI_initialized () () () ()


setdummys MPI_abort (comm errorcode ierror)
setinout MPI_abort () () () ()
```

```
setdummys MPI_comm_size (comm size ierror)
setinout MPI_comm_size () () () ()

setdummys MPI_comm_rank (comm rank ierror)
setinout MPI_comm_rank () () () ()

setdummys MPI_comm_split (comm color key newcomm ierror)
setinout MPI_comm_split () () () ()

setdummys MPI_comm_free (comm ierror)
setinout MPI_comm_free () () () ()

setdummys MPI_error_string (errorcode string resultlen ierror)
setinout MPI_error_string () () () ()

setdummys MPI_get_count (status datatype count ierror)
setinout MPI_get_count () () () ()

setdummys MPI_wait (request status ierror)
setinout MPI_wait () () () ()
```

2.2 Point to point communications

The first problem encountered is that message passing commands perform mainly side-effects. One default in Odyssée is that a routine whose source is not available cannot modify variables other than its arguments. In message passing routines such as `MPI_send` and `MPI_recv` global side-effects have to be declared in the information base. In order to do so, we intend to use a fictitious global variable. For example, the `MPI_send` command sends a value stored in a buffer from the current processor to some other one. In order to simulate such a behavior, we use a fictitious global variable named `link_send_recv`.

```
setdummys MPI_send (buf p2 p3 p4 p5 p6 p7)
setglobals MPI_send (link_send_recv)
setinout MPI_send (buf) () () (link_send_recv)

setdummys MPI_recv (buf p2 p3 p4 p5 p6 p7 p8)
setglobals MPI_recv (link_send_recv)
setinout MPI_recv () (buf) (link_send_recv) ()
```

Figure 2.1: First description of `MPI_send` and `MPI_recv`.

```
msg = x

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

source = 0
call MPI_RECV(msg,1,MPI_REAL,source,tag(rank),comm,stat,ierr)
```

Figure 2.2: Sample piece of code `C`

```
msgttl=xttl
msg = x

if (rank .eq. 0) then
  do dest=0, 2
     call MPI_SENDTL(msg,1,MPI_REAL,dest,tag(dest),comm,ierr,msgttl)
  end do
end if

source = 0
call MPI_RECVTL(msg,1,MPI_REAL,source,tag(rank),comm,stat,ierr,msgttl)
```

Figure 2.3: Derivative of `C` with respect to `x`

Using the description shown in Figure 2.1, we are identifying a call to `MPI_send` (`buf`) to an assignment of `buf` to the fictitious variable `link_send_recv`. For example, the piece of code shown in Figure 2.2 can be differentiated correctly with respect to `x`, the generated code is shown in Figure 2.3.

2.2.1 Point-to-point blocking communications

Using the information base shown in Figure 2.1 does enable the system to analyze correctly a code with `MPI_send` or `MPI_recv` commands. But, this analysis does not enable the system to understand what happens on one specific processor, but what happens globally on some processors.

The problem of detecting, at compile time, what happens on each processor can be thought of as an array analysis with regards to a component analysis. This is a difficult problem for AD tools (source-to-source), and cannot be solved easily.

If one looks at the well known "ping-pong problem" where one processor A computes a value and sends it to a processor B that sends some other value back to A, the formal previous description does not give correct results. An example of such a program is shown in Figure 2.4, and its hand written derivative with respect to x is shown in Figure 2.5. In the next sections, we will try to make Odyssée generate this derivative which is correct and optimal in the number of non zero derivatives computed.

```
msg1 = x
msg2 = y
if (rank.eq.0) then
   dest = 1
   source = 1
   call MPI_send (msg1,1,MPI_REAL,dest,tag1,MPI_COMM_WORLD,ierr)
   call MPI_recv (msg2,1,MPI_REAL,source,tag2,MPI_COMM_WORLD,stat,ierr)
else
   dest = 0
   source = 0
   call MPI_recv (msg1,1,MPI_REAL,source,tag1,MPI_COMM_WORLD,stat,ierr)
   call MPI_send (msg2,1,MPI_REAL,dest,tag2,MPI_COMM_WORLD,ierr)
end if
```

Figure 2.4: "ping-pong" problem

```
msg1ttl = xttl
msg1 = x
msg2 = y
if (rank.eq.0) then
   dest = 1
   source = 1
   MPI_sendtl (msg1,1,MPI_REAL,dest,tag1,MPI_COMM_WORLD,ierr,msg1ttl)
   MPI_recv (msg2,1,MPI_REAL,source,tag2,MPI_COMM_WORLD,stat,ierr)
else
   dest = 0
   source = 0
   MPI_recvtl (msg1,1,MPI_REAL,source,tag1,MPI_COMM_WORLD,stat,ierr,msg1ttl)
   MPI_send (msg2,1,MPI_REAL,dest,tag2,MPI_COMM_WORLD,ierr)
end
```

Figure 2.5: Hand written derivative for the "ping-pong" problem with respect to x

First solution

Using the information base described in Figure 2.1, the generated code is correct if the strategy of parallelization leads to the execution of the same code on all the processors, otherwise it may give wrong results. It is easy to understand from the source code of the ping-pong problem why this method does not work and why it is difficult to analyze correctly such a code at compile time. In fact the time of execution of the instructions can not be easily deduced from the source code except if one knows that such a loop is really a parallel loop.

Using the given formal description will lead for the AD tool to identify the previous code with:

```
if (rank.eq.0) then
   link_send_recv := msg1
   msg2 := link_send_recv
else
   msg1 := link_send_recv
   link_send_recv := msg2
end if
```

instead of a parallel execution of:

```
0: link_send_recv := msg1
1: msg1 := link_send_recv

1: link_send_recv := msg2
0: msg2 := link_send_recv
```

Then, the derivative code with respect to `x` will be:

```
msg1ttl = xttl
msg1 = x
msg2 = y
if (rank.eq.0) then
   dest = 1
   source = 1
   MPI_sendtl (msg1,1,MPI_REAL,dest,tag,MPI_COMM_WORLD,ierr,msg1ttl)
   MPI_recvtl (msg2,1,MPI_REAL,source,tag,MPI_COMM_WORLD,stat,ierr,msg2ttl)
else
   dest = 0
   source = 0
   MPI_recv (msg1,1,MPI_REAL,source,tag,MPI_COMM_WORLD,stat,ierr)
   MPI_send (msg2,1,MPI_REAL,dest,tag,MPI_COMM_WORLD,ierr)
end
```

This code does not compute the derivative of the "ping-pong" program. For the ping-pong program, this information base cannot be used.

Second Solution

An other approach should be to make the system consider that all the message-passing commands are active whatever value they are passing. Such a choice can be declared by modifying the information base such that the fictitious variable `link_send_recv` (resp. buf) is an input (resp. output) in both the `MPI_send` and `MPI_recv` commands, and to differentiate the code with respect to the variables chosen by the user plus this fictitious variable.

```
setdummys MPI_send (buf p2 p3 p4 p5 p6 p7)
setglobals MPI_send (link_send_recv)
setinout MPI_send (buf) (buf) (link_send_recv) (link_send_recv)

setdummys MPI_recv (buf p2 p3 p4 p5 p6 p7 p8)
setglobals MPI_recv (link_send_recv)
setinout MPI_recv (buf) (buf) (link_send_recv) (link_send_recv)
```

Figure 2.6: Second description of `MPI_send` and `MPI_recv`.

The Figure 2.7 shows the derivative generated using this second information base (shown in Figure 2.6), and differentiating the source example with respect to `x` and `link_send_recv`.

```
msg1ttl = xttl
msg1 = x
msg2ttl = 0.
msg2 = y
if (rank.eq.0) then
   dest = 1
   source = 1
   MPI_sendtl (msg1,1,MPI_REAL,dest,tag,MPI_COMM_WORLD,ierr,msg1ttl)
   MPI_recvtl (msg2,1,MPI_REAL,source,tag,MPI_COMM_WORLD,stat,ierr,msg2ttl)
else
   dest = 0
   source = 0
   MPI_recvtl (msg1,1,MPI_REAL,source,tag,MPI_COMM_WORLD,ierr,msg1ttl)
   MPI_sendtl (msg2,1,MPI_REAL,dest,tag,MPI_COMM_WORLD,stat,ierr,msg2ttl)
end
```

Figure 2.7: Generated derivative for the "ping-pong" problem with respect to `x`

As we said before, Odyssée has generated a derivative code where all the calls to `MPI_send` and `MPI_recv` are active. This leads to useless communications (in the example `msg2ttl` is passed whereas it is zero) but also to useless computations, i.e. computations of zero

derivatives. In the example if `msg2` is used to compute `z` after the execution of the branch, the derivative of `z` will be computed even if it is actually constant.

2.2.2 Point-to-point non blocking communications

If one looks at non blocking communications the previous problem of formal execution of a code is more difficult to handle. In the case of non blocking communications, the matching between `MPI_send` or `MPI_isend` and `MPI_recv` or `MPI_irecv` is even more complicated to perform at compile time than for blocking ones.

The Figure 2.8 shows an example where some blocking and non-blocking communications are used.

```
do i=1, n
   call MPI_irecv (rbuf(i),1,type,node(i),rpid(i),myname,ierr)
end do
do i=1, n
   sbuf(i) = ce(i)
   call MPI_send (sbuf(i),1,type,node(i),myname,rpid,ierr)
end do
do i=1, n
   call MPI_wait (rpid(i),istat,ierr)
end do
do i=1, n
   ce(i) = rbuf(i)
end do
```

Figure 2.8: Example of code with non blocking communications

From this example (see Figure 2.8), one can see that the `MPI_irecv` command seems to be executed before the `MPI_send`, therefore no dependencies between `sbuf` and `rbuf` are detected at compile time, moreover even if `ce` is active, `rbuf` will be passive.

For non blocking communications, there is no other way but to generate a code where all the point-to-point communications are active by defining the information for the `MPI_irecv` and `MPI_isend` commands to have both `link_send_recv` as input and `buf` as output shown in Figure 2.9.

Using the information base shown in Figure 2.9, the generated code is correct even if as we said before it is not optimal in the number of non zero derivative to be computed.

```
setdummys MPI_isend (buf p2 p3 p4 p5 p6 p7 p8)
setglobals MPI_isend (link_send_recv)
setinout MPI_isend (buf) (buf) (link_send_recv) (link_send_recv)

setdummys MPI_irecv (buf p2 p3 p4 p5 p6 p7 p8 p9)
setglobals MPI_irecv (link_send_recv)
setinout MPI_irecv (buf) (buf) (link_send_recv) (link_send_recv)
```

Figure 2.9: Information base for non blocking communications

2.3 Collective communication

Using collective communications, there is no problem of match between send and receive as each communication stands for send and receive of the message. Then, there is no need for the use of any fictitious global variable.

```
setdummys MPI_allreduce (sendbuf recvbuf count datatype op comm ierror)
setinout MPI_allreduce (sendbuf) (recvbuf) () ()

setdummys MPI_reduce (sendbuf recvbuf count datatype op root comm ierror)
setinout MPI_reduce (sendbuf) (recvbuf) () ()

setdummys MPI_bcast (buf count datatype root comm ierror)
setinout MPI_bcast (buf) (buf) () ()
```

Figure 2.10: Information base for collective communications

Using the information base shown in Figure 2.10, the generated code is correct and optimal in the number of non zero active variables generated because there is no over approximation over the dependencies.

2.4 Conclusion

In the previous sections, we have shown the information base for all kind of potentially active MPI commands. For the blocking commands we have given two solutions; the first solution can be used if one knows that the parallelization strategy which is used consists of executing the same code on all the processors. The second solution is more general and can be used in any parallelization context, but leads to the generation of non optimal derivative codes. For the non blocking commands, the only solution is to over-estimates the dependencies and to differentiate the code with respect to the fictitious variable. For the

collective communications, there is no problem to define the base and generate an optimal code in the number of active variables generated.

Using the information base, the system is able to generate a correct derivative code. But one has to define the derivatives of all the MPI commands to compile and run the generated code.

Chapter 3

MPI Derivative library

We have restricted ourselves to the direct mode of automatic differentiation in order not to get into transposition and storage problems. The derivative of each MPI routine called in the code generated by Odyssée has the same name as the initial routine suffixed by `tl` in direct mode and takes as arguments the previous arguments plus their derivatives.

For example the derivative call of an active call of:

```
CALL MPI_send(buf,lbuf,MPI_double_precision,node,tagid,com,ierror)
```

will be:

```
CALL MPI_sendtl(buf,lbuf,MPI_double_precision,node,tagid,com,ierror,bufttl)
```

We have gathered all the derivative of the potentially active routines into a derivative library written in Fortran 77 and named `MPI_tl.f`. One can write this library using two different strategies for the linear MPI commands; the initial message and the derivative message can be passed in the same communication or they can be passed in different communications.

3.1 First solution for the linear commands

If one chose to concatenate the derivatives with the initial value to maintain the number of messages passed, this correspond to the strategy of computing on each processor the function and its derivatives.

One must notice that there is one problem in this definition for declaring the size of the intermediate buffer `newbuffer`. Standard Fortran compiler do not accept an argument of a routine to be the size of a local variable. A maximal size `lbmax` for the send/recv buffers must be set by the user to a correct value.

For example, the derivative of `MPI_sendtl` and `MPI_recvtl` is defined as in Figures 3.1 and 3.2.

```
subroutine MPI_sendtl(buffer,lbuf,type,node,tag,name,ierror,bufferttl)

implicit none
INTEGER lbuf, ierror, tag, name, type
REAL*8 buffer(lbuf), bufferttl(lbuf)

integer lbmax
PARAMETER (lbmax = ??)
REAL*8 newbuffer(2*lbmax)
integer i

do i=1, lbuf
   newbuffer(i) = buffer(i)
   newbuffer(i+lbmax) = bufferttl(i)
end do

call MPI_send (newbuffer,2*lbmax,type,node,tag,name,ierror)
return
end
```

Figure 3.1: Derivative of `MPI_send` using one message.

Using one message, an intermediate variable is created which can be big sometimes. Moreover as we said before its size is over approximated by `lbmax`.

3.2 Second solution for the linear commands

One can also write the derivative sending the initial buffer as well as the derivative one but using two different tags. In order not to change the call to `MPI_sendtl`, one has to introduce a common block that contains the new tags, and the user has to give a correct value to this new tag (`tagbis` in the example).

Using this strategy, the derivative code for `MPI_sendtl` and `MPI_recvtl` is shown in Figures 3.3 and 3.4.

Using two messages, there is no intermediate buffer to be created, but two transmissions are used and a new tag must be created.

3.3 Derivative of the non linear commands

For example, one can see that the definition of the derivative of `MPI_reduce` is a lot more complicated than for the linear command.

```
subroutine MPI_recvtl(buffer,lbuf,type,node,tag,name,ierror,bufferttl)

implicit none
INTEGER lbuf, ierror, tag, name, type, node
REAL*8 buffer(lbuf), bufferttl(lbuf)

integer lbmax
PARAMETER (lbmax = ??)
REAL*8 newbuffer(2*lbmax)
integer i

call MPI_recv (newbuffer,2*lbmax,type,node,tag,name,ierror)
do i=1, lbuf
   buffer(i) = newbuffer(i)
   bufferttl(i) = newbuffer(i+lbmax)
end do
return
end
```

Figure 3.2: Derivative of `MPI_recv` using one message.

```
subroutine MPI_sendtl(buffer,lbuf,type,node,tag,name,ierror,bufferttl)

implicit none
INTEGER lbuf, ierror, tag, name, type

REAL*8 buffer(lbuf), bufferttl(lbuf)

INTEGER tagbis
common /newtag/tagbis

tagbis = ??
call MPI_send (bufferttl,lbuf,type,node,tagbis,name,ierror)
call MPI_send (buffer,lbuf,type,node,tag,name,ierror)
return
end
```

Figure 3.3: Derivative of `MPI_send` using two messages.

First, it cannot treat independently the initial buffer and the derivative one as the value of the initial buffer is needed as well as the derivative one. Second, it must execute the initial operation on the buffer components and the corresponding derivative operation on the components of the derivative of the buffer. It must then test the operation and call the

```
subroutine MPI_recvtl(buffer,lbuf,type,node,tag,name,ierror,bufferttl)

implicit none
INTEGER lbuf, ierror, tag, name, type, node
REAL*8 buffer(lbuf), bufferttl(lbuf)

INTEGER tagbis
common /newtag/tagbis

call MPI_recv (bufferttl,lbuf,type,node,tagbis,name,ierror)
call MPI_recv (buffer,lbuf,type,node,tag,name,ierror)
return
end
```

Figure 3.4: Derivative of `MPI_recv` using two messages.

associated derivative operation. This leads to problems for user defined operations because the definition of `MPI_reducetl` has to be modified and the derivative of the operation has to be written by the user.

For example, the derivative function for the `MPI_prod` operation can be naively defined in Fortran 77 as shown in Figure 3.5.

3.4 Conclusion

The MPI derivative library can be written in several ways. We have chosen for the linear command to gather the initial values as well as the derivatives in one message to maintain the number of communications (from the source code). This library can be modified by the user to fit some specificity of his code. The definition of the library changes with the type of the passed message, so we have written a library in single precision and one for double precision. The naive definitions of MPI derivatives presented in this chapter can be optimized using some `mpi_type` definition to gather the value and its derivative in a couple and defining new operations on those couples.

```
subroutine prodtl (sbuf,rbuf,count,datatype,op,root,comm,sbufttl,rbufttl)

call MPI_reduce (sbuf,rbuf,count,datatype,op,root,comm)
call MPI_bcast (rbuf,count,datatype,root,comm)
do i=1, count
   aux(i) = rbuf(i)/sbuf(i)*sbufttl(i)
end do
call MPI_reduce (aux,rbufttl,count,datatype,MPI_sum,root,comm)
return
end

subroutine MPI_reducetl (sbuf,rbuf,count,datatype,op,root,comm,
                         sbufttl,rbufttl)

C Case of the linear operation MPI_sum
if (op.eq.MPI_sum) then
   call MPI_reduce (sbufttl,rbufttl,count,datatype,op,root,comm)
   call MPI_reduce (sbuf,rbuf,count,datatype,op,root,comm)
endif

C Case of a non-linear operation
if (op.eq.MPI_prod) then
   call prodtl (sbuf,rbuf,count,datatype,op,root,comm,sbufttl,rbufttl)
endif
return
end
```

Figure 3.5: Naive implementation of the derivative of `MPI_reduce`

Chapter 4

Test on a three-dimensional compressible Navier-Stokes solver

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 [4]

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).

4.1 Description of the source code

The source code can be described in a really simple manner by the description of the main loop shown in Figure 4.1. You can see from Figure 4.2 that it is much more complicated than that.

In this section, we will describe the directional derivatives of this code. We wanted to get the partial derivative of the flux with respect to the solution at each time step. From the source code that means the derivation of the routine `ComputeFlux` shown in Figure 4.1.

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

Figure 4.1: Structure of the source code

4.2 Differentiation of `computeflux`

We were interested in the derivative of the flux `ce` computed by the routine `computeflux` with respect to the physical state `ua` at each time step. The routine `excflux` used nonblocking receives, we have decided to rewrite it using blocking receives in order to differentiate it automatically. Thereafter we were able to differentiate the routine `computeflux` with respect to the variable `ua` using the script file shown after:

```
# 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 computefluxtl
diff -tl -vars ua -h Computeflux -o computefluxtl
```

The tangent code of `ComputeFlux` generated using `Odyssée` computes the directional derivatives 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:

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

where n is the total number of nodes.

In order to compute one directional derivative, one has to modify the driver named `nsc3dm` to call the routine `computefluxtl` instead of `computeflux`, but also to initialize

```
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° 3715                     +- glisum +- (MPI_allreduce)
```

Figure 4.2: Call graph of the source code

the direction `uattl` and print the directional derivatives `cettl`. If one wants to get partial derivatives, let say for example

$$\forall i \in [1..5] \quad \forall j \in [1..n] \quad \frac{\partial ce(i,j)}{\partial ua(2,37)}$$

then the direction must be set to 1 in the direction $ua(2,37)$ which is in practice an assignement of $uattl$ such that:

$$\begin{array}{lll} \forall i \in [1..5]\ \forall j \in [1..n] & i \neq 2, j \neq 37 & uattl(i,j) = 0. \\ & i = 2, j = 37 & uattl(i,j) = 1. \end{array}$$

4.3 Hand writing of the driver

In this section, we consider the computations performed globally; that means from one direction mapped on all the processors we get one direction reconstructed from all the directions (local to the processors). The routine that initializes the direction in a coherent manner on all the processors is called `setdirection`. But we have also written a routine that assembles the local (to each processor) values of those partial derivatives from all the processors to processor zero, and prints it on a file. The routine that assembles the direction is called `getdirection`. The routine that sets the initial direction, computes one directional derivative and gets the final direction is shown in Figure 4.8.

We have also modified the main program to compute the partial derivatives at each time step, just to get several derivatives and be able to check the result. We have then replaced the initial call to `ComputeFlux` to `ComputeFlux_ad` as shown in Figure 4.7.

4.4 Comparison of directional derivatives

In Figure 4.4 and Figure 4.5, we have printed all along the simulation (47 time steps), the five components of one partial derivative chosen at random in the grid (spatial nodes 1084 and 42) with respect to $ua(2,37)$. The index of those components are the global definitions of the derivatives on.

To validate those derivatives, we have compared the values computed using centered divided differences (DD) to those computed using AD. We have written a second driver namely `ComputeFlux_dd` presented in Figure 4.9. This driver computes at the same point the initial function and the approximation of the one partial derivative. We have printed on various spatial point both derivatives all along the simulation on the same figure, and both curves are identical. For example in Figure 4.6 the values at spatial node 42 for the second component of ce have been plotted and the curves are exactly the same.

We have used the function for execution time measurement already existing in the source code to try to evaluate the increase in execution time due to the computation of the derivatives. To do so, we have compared the initial function (I), the modified version that uses centered divided differences (DD) and the two versions that uses automatic differentiation

(AD) using only one message (AD_1) or two messages (AD_2). The cluster on which those time evaluation have been performed consists of several `PC` (Pentium Pro 200 Mhz), but we only use three. The execution time measured in this context are then closely dependent on the time necessary to communicate from one machine to another one. The Figure 4.3 presents in the column `Simulation time` the average (in seconds) over ten executions of: the total simulation time labeled `Total`, the fluxes computation time labeled `Flux` (without communication), and the communication time labeled `Comm.`. In the column labeled `Ratio`, we present the ratio of the execution time between the function and its derivative. One can see from these results that the ratios in computation time are 2.78 using divided differences and should be 3 because the driver calls `ComputeFlux` three times. This shows that the times shown are approximately correct. Due to the error in the time evaluation, we consider that, as expected, the two execution times shown using AD are the same. The computation of the derivative of the flux is then $2.10 - 2.26$ times as expensive as the initial function which is correct in comparisons with other results obtained on sequential codes. As for the communication time, the small number of communications of the flux (and its derivative) explains the ratio of $1.10 - 1.15$ obtained instead of 3 using divided differences, 2 using AD_2-AD_1. We have only tested those codes on 3 processors and this does not make the difference between AD_2 and AD_1 visible. One must notice that within AD_1 the size of the new buffer that contains the value and its derivative is over approximated and this also should make a difference and perhaps explains the ratio of 1.8. The column `Runtime sizes` of Figure 4.3 shows all the segment or section size in bytes of the four runtimes we have compared. The size of the `text`, `data`, and `bss` (uninitialized data) segments (or sections) and their total are presented (as given by the `size` unix command). From these results, one can see that there is no great difference between the sizes of those four runtimes. We just want to say that the storage of the derivatives increases the `bss` segment, in this example the memory necessary for the storage of the derivatives is negligible compared to the rest of the variables.

Code	Simulation time			Ratio		Runtime sizes			
	Flux	Comm.	Total	Flux	Comm.	text	data	bss	total
`I`	6.945	3.617	70.024	1	1	606538	11552	5498648	6116738
`DD`	19.291	3.972	88.908	2.78	1.10	658210	12640	6188152	6859002
AD_1	15.712	6.528	81.666	2.26	1.80	660090	12600	6303352	6976042
AD_2	14.602	4.179	81.579	2.10	1.15	621538	12484	6299448	6933470

Figure 4.3: Time/size comparison of the four executions.

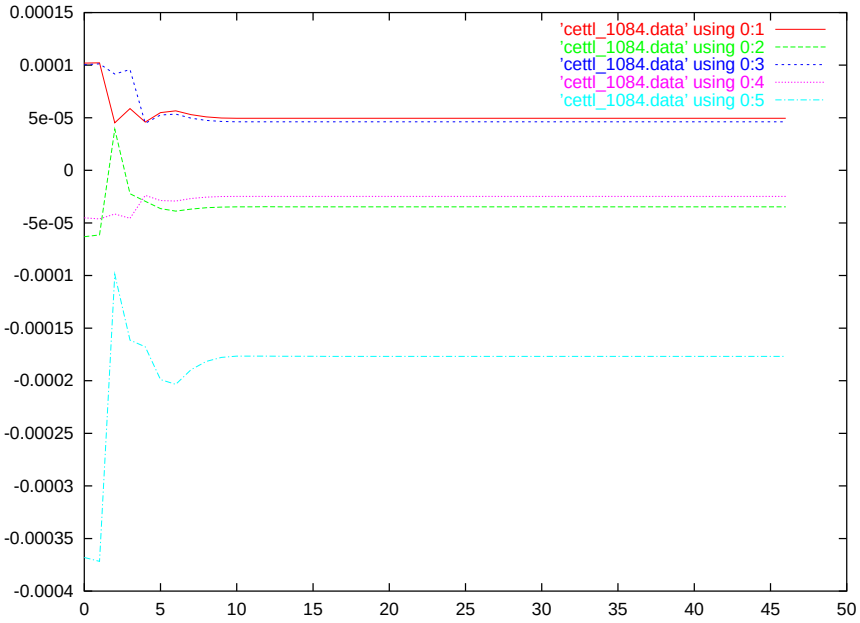

Figure 4.4: Derivative of the five components of ce at node 1084

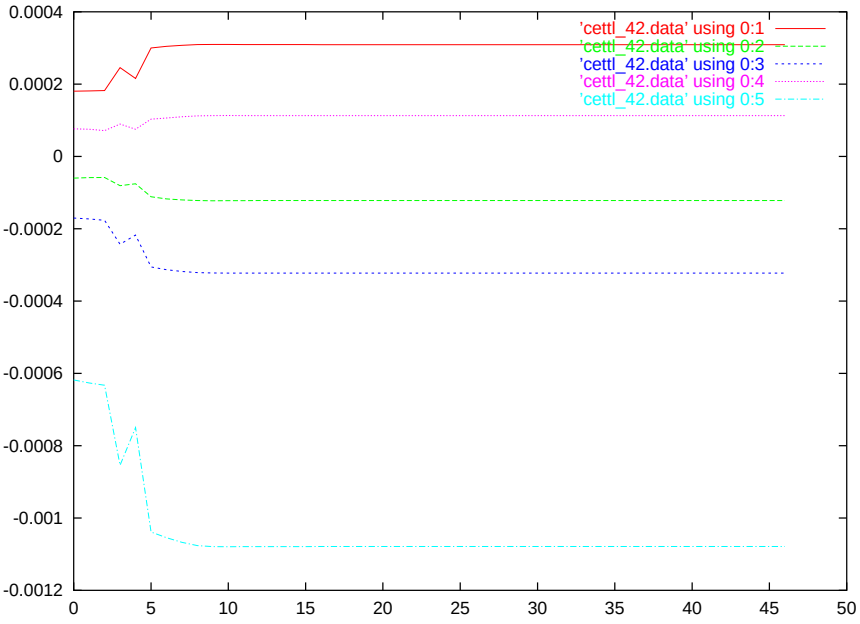

Figure 4.5: Derivative of the five components of ce at node 42

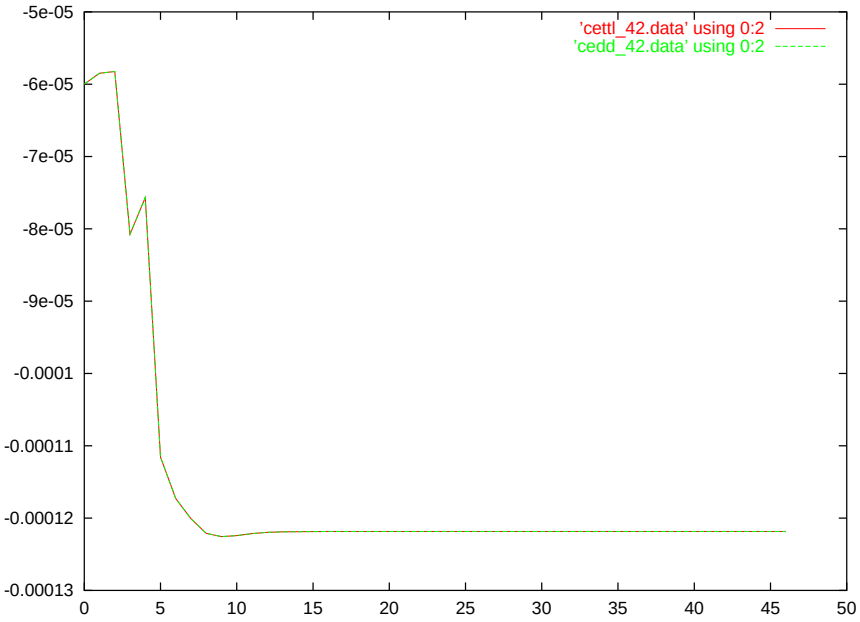

Figure 4.6: Partial derivative using AD and DD

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

Figure 4.7: Structure of the derivative code

```
subroutine ComputeFlux_ad

INCLUDE 'Param3D.h'
INCLUDE 'Paral3D.h'

INTEGER i1, i2, j1, j2, i, j

REAL*8 uattl(5,nsmax),unttl(5,nsmaxg),cettl(5,nsmax)
REAL*8 dxttl(5,nsmax),dyttl(5,nsmax),dzttl(5,nsmax)
COMMON /rsol1ttl/uattl, unttl, cettl, dxttl, dyttl, dzttl

do i=1, 5
   do j=1, nsmax
      uattl(i,j) = 0.
      cettl(i,j) = 0.
   end do
   do j=1, nsmaxg
      unttl(i,j) = 0.
   end do
end do

C     choice of the input direction
i1 = 2
j1 = 37

call setdirection (uattl,i1,j1)
call computefluxtl ()

C     choice of the output direction
CALL getdirection (cettl)
return
end
```

Figure 4.8: Driver for the derivatives by automatic differentiation

```
subroutine ComputeFlux_dd ()

C Storage of all the inputs than can be modified
CALL store_state ()

C     choice of the local input direction
i1 = 2
j1 = 10
epsilon = 1E-8

C     computes the first local point ce_p
ua(i1,j1) = ua(i1,j1) + epsilon
call computeflux ()
do i2 = 1,5
   do j2 = 1,nsmax
      ce_p(i2,j2) = ce(i2,j2)
   end do
 end do

C     restores the correct values of the input
CALL restore_state ()

C     computes the second local point ce_m
ua(i1,j1) = ua(i1,j1) - epsilon
call computeflux ()
do i2 = 1,5
   do j2 = 1,nsmax
      ce_m(i2,j2) = ce(i2,j2)
   end do
end do

C  computes the local divided differences of all the components of ce
do i2 = 1,5
   do j2 = 1,nsmax
      dd(i2,j2) = (ce_p(i2,j2) - ce_m(i2,j2)) / (2 * epsilon)
   end do
end do

C     restores the correct values of the input
CALL restore_state ()

C     computes the initial function
call computeflux ()
end
```

Figure 4.9: Driver for the derivatives by divided differences

Chapter 5

Conclusion

We have already tuned Odyssée to make it handle parallel codes. A general (under optimal) strategy has been adopted for the differentiation, and it considers each variable passing command to be active. We have also defined the derivatives of all potentially active commands, and chosen a strategy for passing the derivative values.

An information base file is now available and can be used to differentiate a code. A library of derivative for MPI commands written in Fortran 77 is also available. We have chosen to chose the robust strategy even if it may be under optimal, the user can refine the derivative code by hand easily. There is not that many call to MPI commands in a program that the user is enable to correct the generated code.

We have tested the adapted version of Odyssée on a sample code that uses data partitioning to compute a polynomial. In the Appendix A, we present the results obtained on this example. The size of this example allows us to show in the report the whole code. We have also tested the adapted version of Odyssée on a one piece of a larger code to show the feasibility. The results are presented in Chapter 4 and show the correctness of this adapted version of Odyssée.

The method is validated in direct mode and we are now able to work on the robustness of the derivative library. For example, we could test the actual type of values passed in order to check if the version of the derivative fits the context of call (real or double).

An other research axe will be the extension of the methodology to the reverse mode. In reverse mode, the storage of all intermediate computation as well as the transposition ask for some detailed study.

Bibliography

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

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

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

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

[5] MPI Forum. *MPI: a Message Passing Interface Standard*, 1995.

Appendix A

Simple test

We have tested the adapted version of Odyssée on a sample code and present the results. In this section we present the derivation of this code from the user point of view step by step.

A.1 Step by step differentiation

1. Choice of the example: We have implemented the evaluation of a polynomial in parallel by data partitioning. The root processor send the data to all the processors, each processor computes one term and the terms are summed using a reduce command.

2. Generation of the code: We have chosen not to use the user interface, but a batch file that could be executed from a makefile. We show this file in the following lines.

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

# Reads all the input files
load test.f

# Reads the information base from MPI.bi
loadibasis MPI

# Differentiates poly with respect to x and prints it in the file polytl
diff -tl -vars x link_send_recv -h poly -o polytl
```

3. Compilation

```
mpif77   -I/usr/local/mpi/include   polytl.f   MPI_tl.f   -o   test_poly
```

A.2 Source code of the initial program

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

implicit none
include 'mpif.h'

integer numtasks,rank,ierr,dest,source,stat,tag(0:2)
real alpha(0:2),x(0:2),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
```

A.3 Source code of the derivative

```
COD Compilation unit : polytl
COD Derivative of unit :  poly
COD Dummys:  x alpha resu tag rank numtasks
COD Active IN   dummys:  x link_send_recv
COD Active OUT  dummys:  resu x
COD Active OUT globals:  link_send_recv


SUBROUTINE POLYTL (x,alpha,resu,tag,rank,numtasks,xttl,resuttl)

IMPLICIT NONE
INCLUDE 'mpif.h'
INTEGER numtasks,dest,stat,ierr,source,rank,tag(0:2)

REAL y,yttl,resu,resuttl,x(0:2),xttl(0:2),alpha(0:2)
```

```
IF (rank.EQ.0) THEN

  DO dest = 0,2
     CALL MPI_SENDTL(x(rank),1,MPI_REAL,dest,tag(dest),MPI_COMM_WORLD,ierr,
               xttl(rank))
  END DO
END IF
source = 0
CALL MPI_RECVTL(x(rank),1,MPI_REAL,source,tag(rank),MPI_COMM_WORLD,stat,ierr,
          xttl(rank))

yttl = alpha(rank)*xttl(rank)
y = alpha(rank)*x(rank)
CALL MPI_REDUCETL(y,resu,1,MPI_REAL,MPI_SUM,0,MPI_COMM_WORLD,ierr,
            yttl,resuttl)
RETURN
END
```

A.4 Main program

The main program initializes the source variables, but also the direction and prints the result:

```
program eval_polytl

implicit none
include 'mpif.h'

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

real alpha(0:2), x(0:2), resu
real xttl(0:2), resuttl

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

xttl(0) = 0.
xttl(1) = 1.
xttl(2) = 0.
```

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

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

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 polytl (x,alpha,resu,tag,rank,numtasks,xttl,resuttl)

if (rank .eq. 0) then
   write (6,*) 'The value of the polynomial is ',resu
   write (6,*) 'The value of the derivative is ',resuttl
endif

call MPI_FINALIZE(ierr)

end
```

A.5 Test on the derivative values

- Execution

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

- Result of the execution

```
Number of tasks=  3 My rank=  2
Number of tasks=  3 My rank=  1
The value of the polynomial is :     26.0000
The value of the derivative is :     2.00000
```

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