# AGRIF: Adaptive Grid Refinement In Fortran

Laurent Debreu, Eric Blayo

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

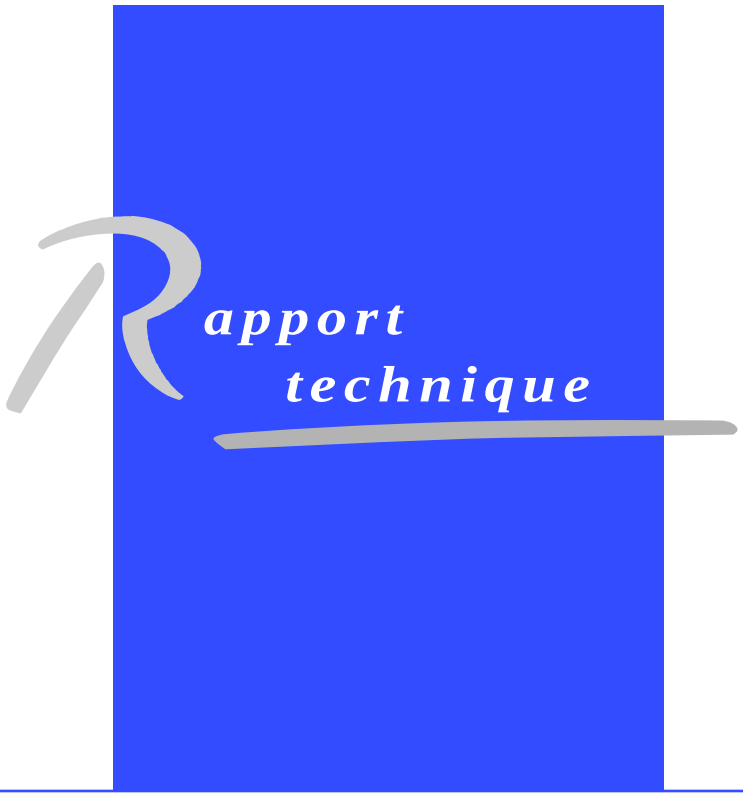
# *AGRIF: Adaptive Grid Refinement In Fortran*

Laurent Debreu — Eric Blayo

**N° 0262**

Juin 2002

THÈME 4




ISSN 0249-0803

# AGRIF: Adaptive Grid Refinement In Fortran

Laurent Debreu , Eric Blayo

Thème 4 — Simulation et optimisation
de systèmes complexes
Projet IDOPT

Rapport technique n° 0262 — Juin 2002 — 16 pages

**Abstract:** This report presents AGRIF a Fortran90 package for the integration of adaptive mesh refinement (AMR) features within a finite difference numerical model. The package mainly consists in two parts. It first provides model-independent Fortran90 procedures containing the different parts of an AMR process: time integration algorithm of the grid hierarchy, clustering algorithm, refinement algorithm, interpolation procedures... In a second part, a Fortran90 model-dependent code is created via the analysis of an entry file written by the user. Both model-dependent and model-independent parts are then linked into a library.

**Key-words:** adaptive mesh refinement, Fortran 90, pointers

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 Montbonnot-St-Martin (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

# AGRIF: Adaptive Grid Refinement In Fortran

**Résumé :** Cet rapport présente AGRIF un paquetage Fortran 90 pour l'intégration de potentialités de raffinement adaptive de maillage dans un modèle aux différences finies. Le paquetage est décomposé en deux parties principales. Il contient tout d'abord un ensemble de procédures Fortran 90 indépendantes du modèle qui contiennt les différentes parties d'un procédé de raffinement de maillage: intégration temporelle de la hiérarchie de grilles, algorithme de clustering, algorithme de remaillage, procédure d'interpolation,... Dans un second temps, le code fortran dépendant du modèle est généré à travers l'analyse d'un fichier de configuration écrit par l'utilisateur. Les deux parties dépendantes et indépendantes du modèle sont ensuite reliées pour créer une librairie.

**Mots-clés :** raffinement adaptatif de maillage, Fortran 90, pointeurs

# 1 Introduction

Since the work of Berger and Oliger [2], adaptive mesh refinement (AMR) for structured grids has become very popular. Its domains of application are numerous and its efficiency has been clearly demonstrated. However dealing with AMR is a difficult task, which often keeps people from investigating the power of the method. As a matter of fact, AMR is intrinsically a dynamic method which requires dynamic memory management and structured types.

**AMRCLAW** [1] is a Fortran77 AMR package for conservation laws that was developed upon the Berger's AMR algorithms and the **CLAWPACK** package of R. LeVeque, and which incorporates all the primary features of the method proposed by Berger. However, with the quite recent apparition of object oriented languages, programming AMR methods has become simpler, and several AMR packages have then been developped. Most of them are implemented in C++, which appeared before Fortran 90. We can cite for instance **Overture** [6] a C++ package subtitled *Object-Oriented Tools for Solving CFD and Combustion Problems in Complex Moving Geometry*, and **DAGH** (*Distributed Adaptive Grid Hierarchy* [5]).

Concerning similar tools in Fortran90, we can cite **PARAMESH** [7], a package actually doing cells refinement, and which differs quite a lot from the original idea of Berger. There exist also AMR packages which actually try to implement C++ features in Fortran 90.

In this paper, we present a new idea for implementing AMR features in an existing numerical model written in Fortran (77 or 90). This package uses the full compatibility between Fortran77 and Fortran90, and thus eliminates all kind of interfaces.

The paper is organized as follows. The main operations involved in an AMR process are briefly reminded in section 2. Then the basic ideas of the package are presented in section 3, and section 4 explains how to use the program which produces the model-dependent Fortran code. Finally, an example is treated in section 5.

# 2 Brief description of the AMR method

In this section, we briefly remind the ideas of the adaptive mesh refinement method for structured grids as described in several papers (e.g. [2] and [3]). The AMR method is designed for the solution of systems of partial differential equations using finite difference techniques. The basic idea is to adjust locally the numerical resolution following a mathematical and/or physical criterion, and therefore to create refined grids (or to remove existing ones) where and when necessary. Moreover, this approach is recursive in the sense that fine grids can contain even finer grids. So the AMR strategy features a hierarchy of resolution levels, each of which contains a set of grids (cf figure 1).

Every grid is covered by a parent (coarser) grid, the root level consisting of one coarse resolution grid covering the entire domain of computation. The spatial and temporal resolutions are divided by a given integer $r$ when skipping from a given level $l$ to the finer level $l+1$ (typically $r = 2, 3$ or 4). With such a convention, once the grid spacing and the time step are chosen on the root grid, the corresponding CFL criterion is automatically verified for

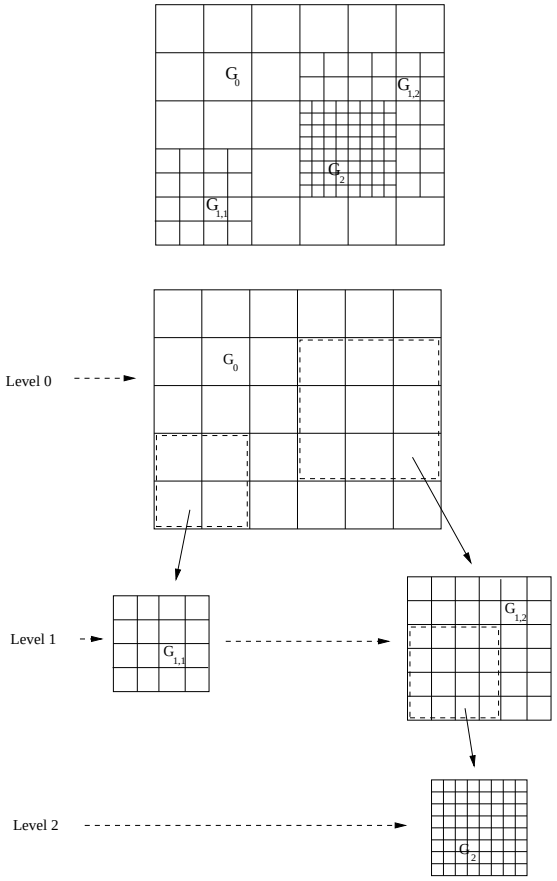



Figure 1: An example of grid hierarchy

all grids. Note of course that, if the stability condition of the model is not a CFL, different refinement ratios in space and time can be used.

## 2.1 Integration of the grid hierarchy

The integration of the grid hierarchy starts at the root level. The solution on the root grid is first advanced one coarse time step $\Delta t_0$. Then this solution is used to provide (by interpolation) boundary conditions for the grids at level 1, which can then be advanced $r$ time steps $\Delta t_1 = \Delta t_0/r$, and so on recursively for grids at deeper levels. Once a solution is computed on a grid at a given level, it is used to update its parent grid solution.

```
Recursive Procedure INTEGRATE(l)

If l == 0 Then nbstep = 1
Else nbstep = refinement-ratio
Endif
Repeat nbstep times
     Step on all grids at level l
     If level l+1 exists Then
          Compute boundary conditions at level l+1
          INTEGRATE(l+1)
          update level l
     Endif
End Repeat
End Procedure INTEGRATE
```

Figure 2: Integration algorithm of the AMR method

The integration algorithm is summarized on figure 2. It is a recursive procedure, which must be called at level 0.

### 2.2 Regridding

As mentioned previously, the grid hierarchy may change from one coarse time step to the other (or at regular intervals, for example every $N$ coarse time steps). A refinement criterion is applied, which detects grid points where a finer grid is necessary, or where an existing fine grid is no more useful. Figure 3 shows the different stages of mesh refinement.

An important part of the regridding procedure consists in the restoration of fine grid values. As a matter of fact, when a new grid is created somewhere in the domain of computation, some points of this new grid may have been included in other grids at the same resolution level in the preceding grid hierarchy. At those points, the values of the different arrays can be restored to avoid an interpolation.

## 3 Adaptive mesh refinement in Fortran90

This section presents the key ideas for implementing AMR in Fortran90 within an existing numerical model. The underlying aim is to do as little changes as possible in the existing Fortran code. Note however that the whole contents of this section can be easily extended to another language being able to deal with pointers (like C or C++).

```
Procedure REGRID
level = levelmax(t)
While level ≥ 0
     Detect-area-to-refine(level)
     If level level+1 exists
          Add-detected-area(level+1)
          Save-values(level+1)
          Clear-level(level+1)
     level = level−1
level = 0
While level ≤ levelmax(t)
     Cluster(level)
     If level level+1 has been created
          Restore-values(level+1)
          Init-grids(level+1)
     level = level+1
End Procedure REGRID
```

Figure 3: Regridding

## 3.1 Notion of pointers

The pointers have been existing for a while in the C language, and were added in the standard of Fortran90. A pointer variable is able to address different locations in the memory during the computation. This is actually exactly what we would like to do in AMR, when skipping from the computation on one grid to the computation on another grid. In our implementation, a unique set of pointers will address sucessively the memory locations corresponding to the variables of the different grids, in the order prescribed by the integration algorithm given in Figure 2.

This is illustrated on Figure 4. Let $u$ and $v$ be two distinct variables of the original numerical model, defined in a common area (as it is nearly always the case). In the adaptive version of the model, $u$ and $v$ are declared as pointers, and each grid of the grid hierarchy is represented by a structured type variable containing the local values of $u$ and $v$ on this grid. Then, each time a grid must be integrated, the pointers $u$ and $v$ are linked to the corresponding local variables of the grid. For example, on the left of Figure 4, the pointers are linked to the variables of grid $G_0$ (e.g. the root grid). This means that each access to a value of the array $u$ or the array $v$ is equivalent to an access to its value on the grid $G_0$.

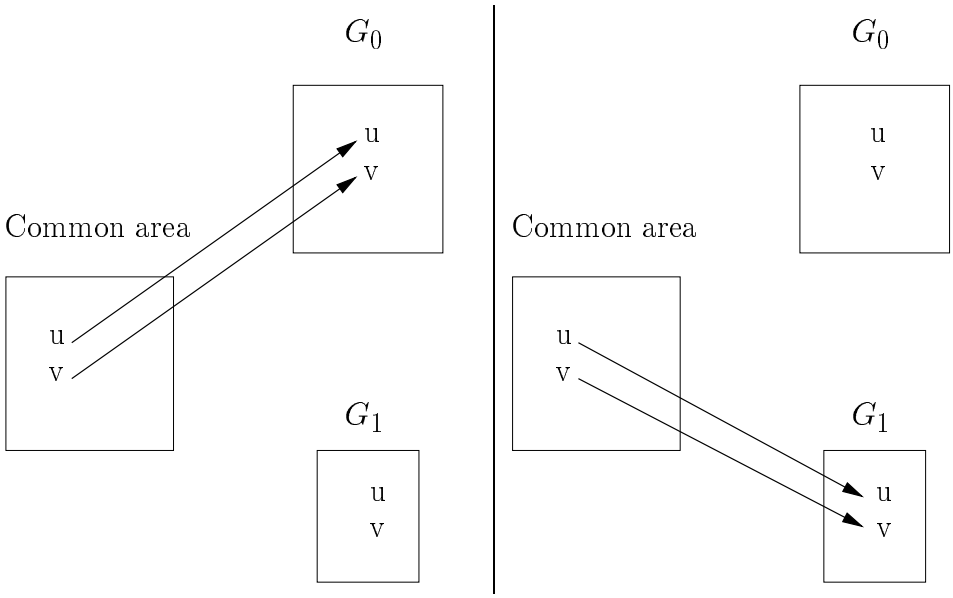



Figure 4: Pointers instantiation

## 3.2 Fortran90 implementation

In Fortran90, the above specifications can be implemented in a very simple way. The grid hierarchy is organized as a classical tree structure. What we add is the declaration of the auxiliary variables (e.g. $u$ and $v$ in the above example).

```
type AMR_grid
----
REAL, DIMENSION(:,:), POINTER :: u,v
----
end type AMR_grid
```

In the common area (e.g. file 'common.h') the two following lines must be present :

```
common/model/u,v
REAL, DIMENSION(:,:), POINTER :: u,v
```

Now before the integration of a grid, we are going to do an instance of the variables in common to the variables of the grid. This can be done by a call to a subroutine written like follows:

```
Subroutine instance(Grid)
  type AMR_grid, Pointer :: Grid
  include "common.h"
    u=>Grid%u
    v=>Grid%v
End Subroutine instance
```

This subroutine is called any time we change the current grid in the integration algorithm (cf Figure 2).

### 3.3 What can a library help for ?

In order to introduce AMR within an existing numerical model, following the previous remarks, a number of tasks must be done. In addition to the model-independent parts of the algorithm (mesh refinement, time integration, clustering ...), the new code must deal with dynamic memory allocation, the previous instanciation subroutine, and several operations on the values of the model variables on the different grids. That is why our aim in creating our package was not only to write a library of subroutines performing all the model-independent tasks of AMR, but also to provide tools that can render the creation of the model-dependent subroutines as easy and automatic as possible. So AGRIF has been designed in such a way that, provided some information on the original model given by the user, it produces some Fortran90 code implementing all the model-dependent part of the AMR. So an additional feature of AGRIF is that, at the end of the process, the user is left with additional simple code in Fortran, that he can of course change directly without using the library again.

## 4 Description of AGRIF

The AMR package AGRIF is a tool to implement AMR within an existing finite difference model. It automatically generates and controls the grid hierarchy and its evolution with time, as well as the integration of the numerical model on the different grids.

To run the package, the user must fill in a pre-defined entry file which contains some parameters of the AMR method and some information on the model (name of the variables, shape of the domain, etc). This file is then read by an executable program, and some lines of Fortran 90 code are created. The user can also write a few additional Fortran 90 subroutines to specify complementary aspects of the AMR. Only one such subroutine is compulsory (the one defining the refinement criteria). The package can then be compiled. This creates a library, and the package is ready to be used.

Some details on the use of this package are given below.

## 4.1 AMR parameters

The AMR library needs some information concerning the refinement parameters, the numerical model, and the domain of computation.

The part of the entry file concerning the global AMR parameters should look this way:

```
commonfile common.h ;        %   name of the common file          %
coeffrefx 3 ;                %     refinement ratio in x          %
coeffrefy 3 ;                %     refinement ratio in y          %
coeffreft 2 ;                %   refinement ratio in time         %
maxlevel 2 ;                 %  maximum number of levels          %
regridding 10 ;              %  interval between regriddings      %
efficiency 80 ;              %  efficiency of the clustering      %
minwidth 10 10 ;             %  minimum size of the grids         %
```

## 4.2 Grid variables

Grid dependent variables used during the calculation must be declared in the entry file. First of all, the name of the variables containing the size of the grid (number of meshes in each direction) :

```
2D nx,ny ;        %   dimensions of the domain   %
```

For the other grid dependent variables (mostly arrays), two kinds of declaration are possible, depending on their use in the model.

If no specific treatment is required (for example if the variable is only a work array or if its values can be deduced from the values of other variables), a simple declaration is sufficient:

**Ex:**
```
REAL(1:nx,1:ny) ::  work1;
INTEGER(0:ny) ::  work2 ;
```

On the opposite, if some specific treatment of the variable is required (e.g. initialization, update, boundary correction, restoration — see section 4.3) , this variable must be "fully declared". This means that the type of the variable and the indices of the first point in the domain must be declared. This type of declaration will be called "explicit declaration".

**Ex:**
```
TYPE1(1:nx,1:ny)[1,1] ::  v1;
TYPE2(0:nx,1:ny)[1,1] ::  v2;
```

The first thing to specify is the position of the variable on the grid, which is represented by the four different keywords `TYPE1, TYPE2, TYPE3, TYPE4`. Figure 5 shows those positions in one cell : `TYPE1` corresponds to the corners of the cells, `TYPE2` to the center of the edges in the $x$-direction, `TYPE3` to the center of the edges in the $y$-direction, and `TYPE4` to the center of the cell.

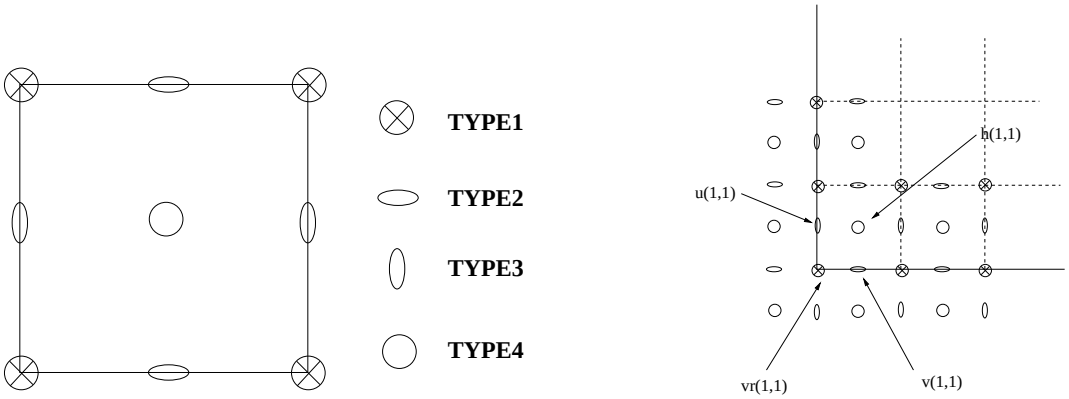



Figure 5: Position of the grid variables, indices in the domain

The indices of the first point of the domain (bottom left corner) must also be declared, as depicted by the examples of Figure 5.

Note that the same types of declarations also exist for 1D arrays.

## 4.3 Working on the values of the variables

The package is able to deal with the following problems :

- initialization of the grid variables when a new grid is created
- update of the grid variables
- correction of the grid variables on or near the boundaries
- restoring the values from an old grid when a new grid is created at the same location

In order to do those operations, the library requires some additional information about the variables involved in those processes. First of all those variables must have been declared in an explicit way (see section 4.2). Then the part of the entry file corresponding to those operations should look like :

```
init u1 u2:SPLINE, fx:LAGRANGE ;   %  variables to be interpolated and inter-  %
                                       polation method for init.
update fx fy:COPY, u2 AVG ;        %  variables to be updated and update      %
                                       method
correct u1(0), v(-1:1) ;           %  variables to be corrected and type of   %
                                       correction
restore u1 u2 ;                    %  variables to be restored                %
```

Note that the boundary values of a variable are interpolated using the same method as for the initialization, except if the user specify another method using the keyword `bcinterp`.

In the AMR method, boundary conditions must be prescribed on the refined grids. The area where this must be done depends on the variables and on the numerical schemes. This information is given to the package by the use of the keyword `correct`, which indicates that some values of a variable on (or near) the boundary must be corrected after each time step on a grid. An index between parentheses indicates the exact location of the correction, which may occur on the boundary (index 0), or one mesh inwards (-1) or outwards (1). Figure 6 shows, for the different types of variables, the positions which will be corrected, depending on the choice of the user.

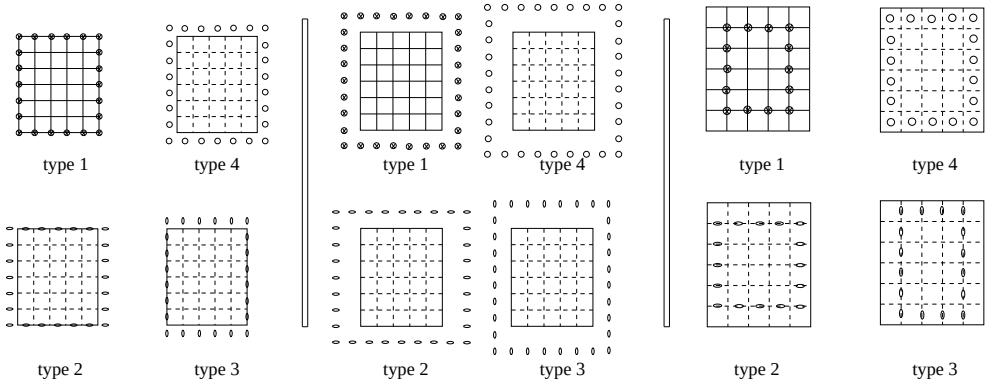



Figure 6: Different possible corrections in the vicinity of the boundary. Left: one mesh inside the grid; Middle: on the boundary; Right: one mesh away from the boundary

One of the important point in the AMR algorithm lies in the fact that, when possible, the values of the variables after a regridding process are not computed by interpolation but simply restored by a copy of previous values. Since we wanted to avoid doubling the occupied memory by managing at the same time all the old values and all the new allocated variables, the package makes the restoration variable by variable in an efficient way. This means that the package proceeds to the following operations variable by variable : allocation of the memory on all new grids, restoration of the old values, de-allocation of the memory on all old grids.

## 4.4 Using fixed grids

One additional interesting feature of this AMR package is its ability to deal with fixed fine grids. If the user wants to use fixed fine grids in the domain, he has to put the keyword `use FIXED_GRIDS` in the entry file. Any number of fixed grids can be declared at any level in the

grid hierarchy. Moreover the user can inhibit the adaptivity while using the keyword `use ONLY_FIXED_GRIDS`. In this case the AMR method becomes a classical nested grids method, with the ability to manage an arbitrary number of fixed grids.

The fixed grids must be declared in a file called AMR_FixedGrids.in, where the user can specify the levels and the locations of the different fixed grids.

When the keyword `use FIXED_GRIDS` is in the entry file, the grid hierarchy manages simultaneously fixed grids and moving grids. Note that when the clustering algorithm creates a new grid that intersects a fixed grid at the same level, this new grid will be shortened or cut in several smaller grids to prevent from grid superpositions. If the grids created through this intersection process are too small (i.e. below the minimum size declared in the entry file), they are removed.

### 4.5 Refinement criteria

Several criteria can be used to decide where the grids must be refined. They can be based for instance on mathematical estimates of the numerical error, or on purely physical considerations (e.g. energy, enstrophy...). That is why the user has to write his own subroutine defining the refinement criteria which will be used in the AMR method. Basically this subroutine fills in a logical array `tab_detect`, so that `tab_detect(i,j)=TRUE` if refinement is needed at the point (i,j). An example of such a routine is given in section 5.2.

### 4.6 Subroutines provided by the package

In order to run the AMR library within an existing numerical model, this one must call two subroutines : *AMR_INIT* and *AMR_STEP*.

*AMR_INIT* makes the grid hierarchy initialization. It must be called only once to create the root (coarser) grid. So the size variables of the coarse mesh must be initialized before calling this routine.

*AMR_STEP* corresponds to the integration procedure described on Figure 2. It integrates the model forward on all grids, and performs a regridding at regular intervals.

Additional subroutines are also provided by the library, which give an easy access to interesting information (e.g. current grid level, current number of time steps on a grid, parent grid of the current grid...).

## 5 An example: A shallow water model

In this section, we give an example of the use of AGRIF in the context of ocean modelling. The model of interest is a basic two dimensional shallow water model.

## 5.1 Equations and discretization

$$\left\{ \begin{array}{lcl} \dfrac{\partial u}{\partial t} & = & (\xi + f)\, v - \dfrac{\partial B}{\partial x} + \dfrac{\tau_x}{\rho_0 h} + A\Delta u - ru \\ \dfrac{\partial v}{\partial t} & = & -(\xi + f)\, u - \dfrac{\partial B}{\partial y} + A\Delta v - rv \\ \dfrac{\partial h}{\partial t} & = & -\dfrac{\partial}{\partial x}(hu) - \dfrac{\partial}{\partial y}(hv) \end{array} \right.$$

where the state variables are $u$ and $v$ (components of the horizontal velocity) and $h$ (surface elevation). $B = \frac{1}{2}(u^2 + v^2) + gh$ is the Bernoulli function, $\xi = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}$ the relative vorticity, $f$ the coriolis parameter (depending of the latitude). $A$ is the laplacian diffusivity coefficient, $r$ the bottom friction coefficient, and $\tau_x$ the zonal wind stress.

The model is discretized using second-order centered finite difference schemes in space and time on an Arakawa C-grid. Figure 7 shows the mesh and the position of the variables on the C-grid.

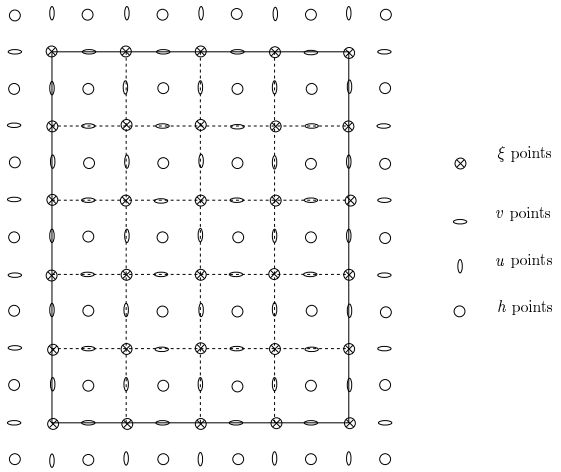



Figure 7: The shallow water model grid

The declaration of the variables is made as follows :
We define by $nx$ and $ny$ the number of cells in the $x$ and $y$ directions (e.g. $nx = 4, ny = 5$ on Figure 7).

The state variables are : $(nxp1 = nx + 1, nyp1 = ny + 1)$

$$\begin{cases} u(1:nxp1, 0:nyp1) & \text{horizontal velocity, } x\text{-direction} \\ v(0:nxp1, 1:nyp1) & \text{horizontal velocity, } y\text{-direction} \\ h(0:nxp1, 0:nyp1) & \text{surface elevation} \end{cases}$$

The additional work arrays are:

$$\begin{cases} vr(1:nxp1, 1:nyp1) & \text{relative vorticity} \\ b(1:nx, 1:ny) & \text{Bernoulli function} \\ lapu(2:nx, 1:ny) & \text{horizontal laplacian for } u \\ lapv(1:nx, 2:ny) & \text{horizontal laplacian for } v \\ fv(1:nyp1) & \text{coriolis parameter, } u\text{-points} \\ fu(0:nyp1) & \text{coriolis parameter, } v\text{-points} \\ taux(0:nyp1) & \text{wind stress, } x\text{-direction} \end{cases}$$

Finally the arrays $(um, uc, up)$, $(vm, vc, vp)$, $(hm, hc, hp)$ are declared in the same way as $u$, $v$ and $h$ for the time integration scheme.

## 5.2 Refinement criterion

Here is a simple example of a detection routine that is based on the value of the relative vorticity.

```
SUBROUTINE detect(tab_detect,nnx,nny)
INCLUDE 'common.h'
LOGICAL :: tab_detect(nnx,nny)
INTEGER :: i,j

vrmax=1.E-4

tab_detect=.FALSE.

DO j=1,nyp1
  DO i=1,nxp1
    IF (ABS(vr(i,j)).GE.vrmax) THEN
      tab_detect(i,j)=.TRUE.
    END IF
  END DO
END DO
END
```

### 5.3 The entry file

According to the previous remarks, the entry file is

```
2D nx,ny;
commonfile common.h ;
coeffrefx 3;
coeffrefy 3;
coeffreft 3;
maxlevel 2;
regridding 10;
TYPE3(1:nxp1,0:nyp1)[1,1] :: up,u,um,uc;
TYPE2(0:nxp1,1:nyp1)[1,1] :: vp,v,vm,vc;
TYPE4(0:nxp1,0:nyp1)[1,1] :: hp,h,hm,hc;
YVECTOR1(1:nyp1)[1] :: fv;
YVECTOR2(0:nyp1)[1] :: fu,taux;
REAL(1:nx,1:ny) :: b;
REAL(2:nx,1:ny) :: lapu;
REAL(1:nx,2:ny) :: lapv;
REAL(1:nxp1,1:nyp1) :: vr;
restore :: u v h;
init :: fu fv taux u v h :SPLINE;
update :: u v h:AVG;
correct :: u :(0),v :(0),h :(0);
xspacename ds;
yspacename ds;
timename dt;
minwidth 16;
```

## 6 Conclusion and Future Work

We have presented in this paper a Fortran 90 package designed to add adaptive mesh refinement features to any existing finite difference numerical model. The package is based on the use of pointers which induces almost no changes in the original code. The use of Fortran 90 prevents the user from dealing with any kind of somehow complicated interfaces with C++. The package already contains model-independent subroutines, and most of the model-dependent part of the additional code is created by a program using a simple entry file written by the user.

The latest version of the package can be obtained by sending an e-mail to the authors.

*Acknowledgements.* We are grateful for financial support from the French Navy under contract EPSHOM 30/97. IDOPT is a joint CNRS/University of Grenoble/Institut National Polytechnique de Grenoble/INRIA project.

## References

[1] AMRCLAW:
Home Page:http://www.amath.washington.edu/~rjl/amrclaw

[2] Berger M. and J. Oliger, 1984: Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comp. Phys.*, **53**, 484-512.

[3] Berger M. and P. Colella, 1989: Local adaptive mesh refinement for shock hydrodynamics. *J. Comp. Phys.*, **82**, 64-84

[4] Berger M. and I. Rigoutsos, 1992: An Algorithm for Point Clustering and Grid Generation. *IEEE Trans. on Systems Man and Cybernetics*, **21**, 1278-1286.

[5] DAGH: http://www.caip.rutgers.edu/~parashar/DAGH

[6] Overture: http://www.llnl.gov/CASC/Overture

[7] PARAMESH: http://www.cs.duke.edu/~emc/Users_manual/amr.html

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 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 Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803