# Parallélisation de modèles météorologiques et climatologiques chez SGI

**V. Balaji**
**SGI/GFDL**

**24 mars 2000**

# GFDL

GFDL est une centre specialisé à la modélisation du climat: surtout avec les modèles couplés: atmosphère, océan, terre, ... participant de base dans IPCC.

Puissance de calcul actuellement disponible aux chercheurs: Cray T90 24p, Cray T3E 128p.

# Modèles de GFDL

- MOM: Modular Ocean Model.
- FMS: Flexible Modeling System.
- Hurricane model.
- HIM: isopcynal model.
- 2 modèles non-hydrostatiques atmosphériques.
- Anciens modèles: SKYHI, Supersource.

# Modernisation

- Valoriser le calcul en parallèle sans trop compromettre la performance vectorielle.

- Formulation de modèles pour avoir des noyaux dynamiques interchangeables, aussi bien que des routines physiques modulaires. Plusieures ensembles des modules physiques sont actuellement disponibles.

- Fortran90.

# FMS: Flexible Modeling System

Noyaux dynamiques:

- Atmosphére:
  - Spectral hydrostatique
  - Grille B de Arakawa hydrostatique
  - Grille C hydrostatique (*)
  - Grille C non-hydrostatique (*)
- Océan:
  - Grille B
  - Grille C (*)
  - Coordonnée verticale géneralisée (*)

# Modèles de programation en parallèle

- Parallèlisme par directives.
- Message passing.
- Multi-threading.

# Parallèlisme par directives

```
!  DOALL private(j)
do j = 1,n
   call ocean(j)
   call atmos(j)
enddo
```

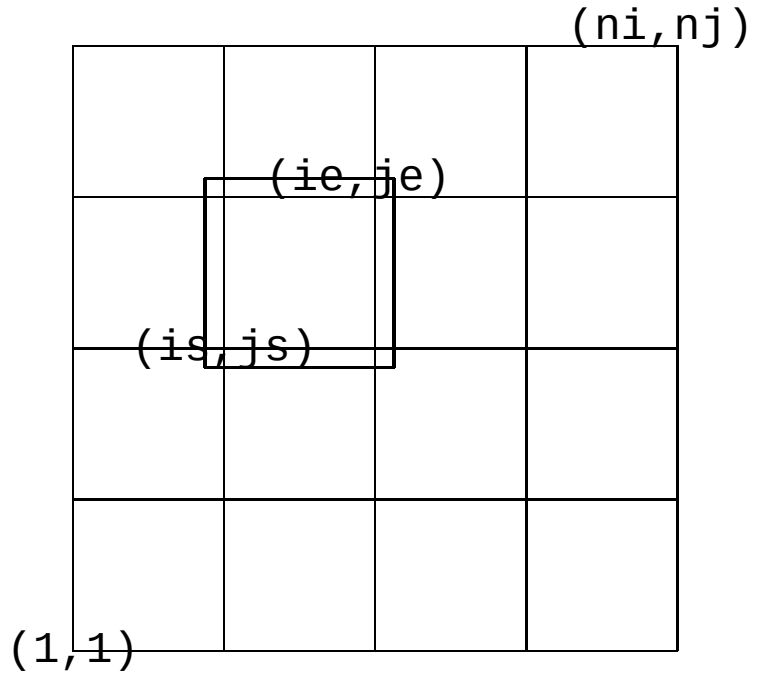
- Architecture type: mémoire unique.
- Tri de variables entre ceux qui sont partagées entre processus et ceux qui sont privées.
- Modifications aux variables partagées nécessitent une *région critique* où le travail se sérialise.

# Message passing

```
call domain_decomp(1,J,js,je,npes)
do j = js,je
   call ocean(j)
   call atmos(j)
enddo
call halo_update()
```

- Architecture type: mémoire distribué.
- Domaine global `(1:J)` est décomposé en `npes` sous-domaines. `(js:je)` sont les indices définissant le début et la fin du sous-domaine.
- Le halo doit être explicitement mis au point à la fin du chaque pas de temps.

```
do j = js,je
   do i = is,ie
        a(i,j) = ...  + a(i-1,j+1) + ...
   enddo
enddo
```



# Multi-threading

```
call domain_decomp(1,J,js(1:nthreads),je(1:nthreads),npes,nthreads)
do n = 1,nthreads
   js=js(n)
   je=je(n)
   do j = js,je
     call ocean(j)
     call atmos(j)
   enddo
enddo
call halo_update()
```

- Architecture type: “cluster of SMPs”.
- Domaine global (`1:J`) est décomposé en `npes*nthreads` fils sur `npes` processeurs. Le “processeur” peut aussi bien être un noeud du SMP.

- Il y a plusieures façons d'exprimer le parallélisme, chacune ayant un lien avec une couche sousjacente d'architecture parallèle.

- Chaque architecture valorise une de ses expressions du parallèlisme. Les institutions de recherche sont obligées par contre d'avoir des codes performants sur plusieures architectures. Malheureusement les "standards" tel MPI ne sont pas au point de livrer une performance satisfaisante sur toute architecture.

- La démarche proposée est de développer une interface qui protége le code d'utilisateurs des details d'outils du parallèlisme. Cette interface permettra une ensemble minimale d'opérations qui est suffisante pour le genre de codes qui nous interressent, et capable d'être implementée dans plusieures expressions du parallélisme. Une implementation au moins sera toujours conformante aux "standards".

# Les modules MPP

Pour GFDL, SGI a développé une interface parallèle qui consiste de trois modules f90 rangées en couches.

- `mpp_mod` est une ensemble d'appels simples pour le partage de données entre processeurs en parallèle.
- `mpp_domains_mod` est une couche supérieure donnant accés aux outils pour la décomposition en domaines des grilles rectilinéaires et la communication entre eux.
- `mpp_io_mod` est une couche pour les entrées/sorties en parallèle.
- Disponibilité libre:

  `http://www.gfdl.gov/~vb`

# mpp_mod

mpp_mod est une ensemble d'appels simples expressibles dans les appels de plusieures bibliothèques parallèles (MPI, SHMEM, Co-Array Fortran). Il est utilisé actuellement par tous les modèles de GFDL (MOM3, FMS). Les subroutines de mpp_mod fournissent des opérations de communication de données, d'opérateurs de réduction, et de synchronisation.

# Caractéristiques de `mpp_mod`

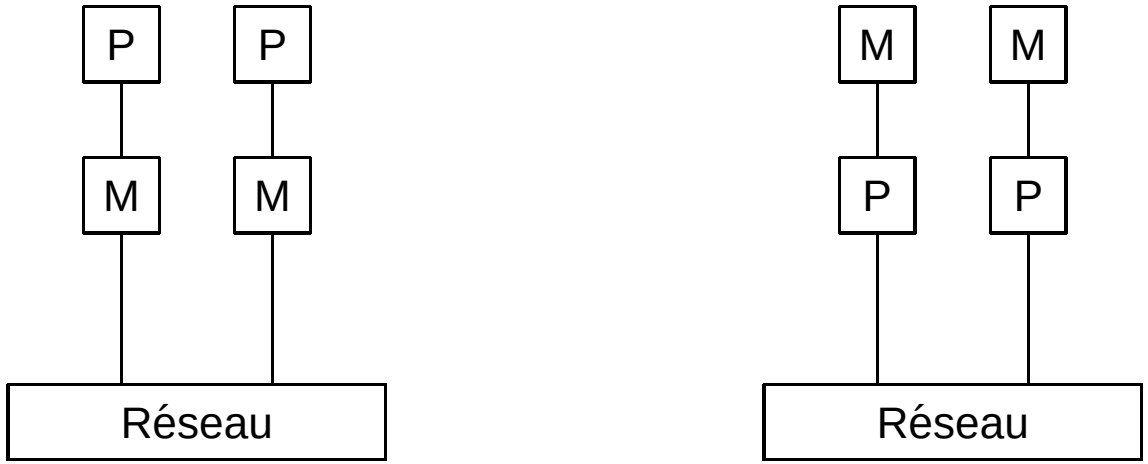
- Minimalisme.
- Accés direct aux bibliothèques sousjacentes si désiré.
- Surcoût minimal par rapport aux bibliothèques inférieures.

# mpp_mod API

- Appels de `mpp_mod` :
  - `mpp_init()`
  - `mpp_exit()`
  - `mpp_transmit()`: routine de base pour la transmission de données. Usage typique présume deux transmissions du processeur par appel, e.g halo.
  - `mpp_sync()`
- Réduction:
  - `mpp_max()`
  - `mpp_sum()`: option d'avoir résultat exactement réproduisible.

# Implementation de `mpp_mod`

- MPI: `MPI_Isend()` et `MPI_Recv()`.
- SHMEM: `shmem_get`.
- sur Origin: envoi d'adresse suivi par une copie directe.




Couplage fort: mémoire plus proche au réseau.

Couplage faible: processeur plus proche au réseau.

# Co-Array Fortran

A mon avis, l’expression du parallélisme la plus naturelle!

```
real :: a(:)    !array (local array)
real :: b(:)[:] !co-array (distributed array)

!on processor 1
a(:) = b(:)[2] !copy b on processor 2 to a on processor 1
```

Développé par SGI, mais (malheureusement) pas encore standardisé.

# mpp_domains_mod

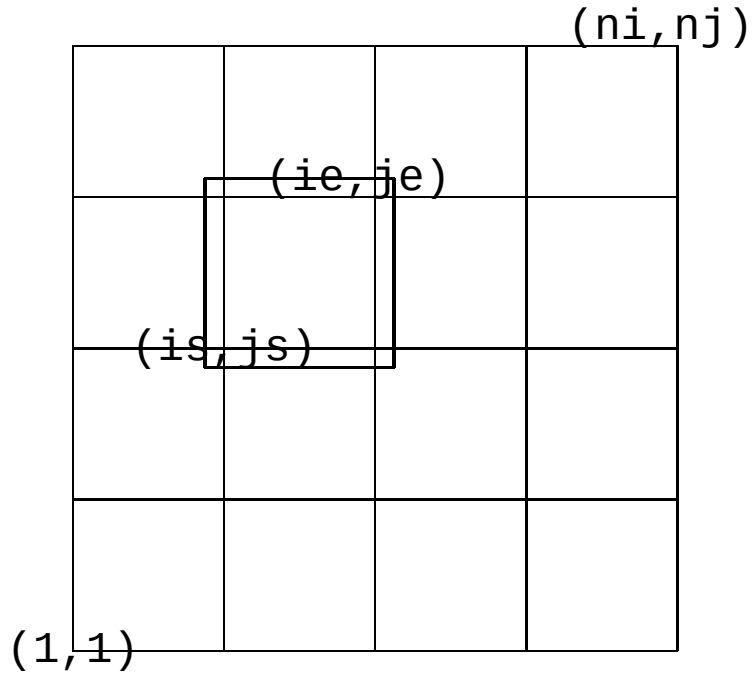
Définition de *domaine*:

- Domaine global: la grille entière du modèle.
- Domaine computationnel: ensemble de points calculé par un processeur.
- Domaine de données: ensemble de points requise par les calculs sur le domaine computationnel.

Toutes ses informations sont maintenues en forme compact dans un variable f90 du type *domaintype*.

```
type, public :: domain_axis_spec
   integer :: start_index, end_index, size, max_size
   logical :: is_global
end type domain_axis_spec
type, public :: domain1D
   type(domain_axis_spec) :: compute, data, global
   integer :: ndomains
   integer :: pe
   integer, dimension(:), pointer :: pelist
   type(domain1D), pointer :: prev, next
end type domain1D
```

```
!domaintypes of higher rank can be constructed from type domain1D
  type, public :: domain2D
     sequence
     type(domain1D) :: x
     type(domain1D) :: y
     integer :: pe
     type(domain2D), pointer :: west, east, south, north
  end type domain2D
```



Le type `domain2D` contient toute information nécessaire pour définir tous les domaines associés avec le processeur. Le processeur qui contient les halos sont disponibles dans une liste enchainée des voisins.

Des grilles assez compliquées (e.g bi-polaire, sphère projeté sur cube) sont représentables dans cette structure s'ils suivent une logique rectilinéaire.

# mpp_domains_mod calls:

- mpp_define_domains()
- mpp_update_domains()

```
type(domain2D) :: domain(0:npes-1)
call mpp_define_domains( (/1,ni,1,nj/), domain, xhalo=2, yhalo=2 )
...
!allocate f(i,j) on data domain
!compute f(i,j) on compute domain
...
call mpp_update_domains( f, domain(pe) )
```

# I/O en parallèle

“I/O certainly has been lagging in the last decade.” – Seymour Cray, Public Lecture (1976).

“Also, I/O needs a lot of work.” – David Kuck, Keynote Address, 15th Annual Symposium on Computer Architecture (1988).

“I/O has been the orphan of computer architecture.” – Hennessy and Patterson, Computer Architecture - A Quantitative Approach. 2nd Ed. (1996).

# I/O en parallèle

- Données distribuées sur plusieurs processeurs, à écrire à un seul fichier.

- Fichier unique distribué sur plusieures disques.

Dans le contexte actuel, on est principalement concerné par le premier, quoique le *disk-striping* est aussi permis.

# I/O Pour les modèles du climat

- Les modèles du climat écrivent typiquement beaucoup plus de données qu'ils lisent. (Lecture à l'entrée, écritures fréquentes en cours de l'integration).

- Fichiers *restart* écrits à la précision de machine (64 bit), fichiers d'analyse géneralement à 32 bit (netCDF utilise IEEE 32 bit).

- Partage de données entre institutions.

# Caractéristiques de `mpp_io_mod`

- Fichier compact (aucun fichier descripteur, etc).
- Fichier ne retient pas ses traces du parallèlisme.
- Accés aux bibliothèques inférieures.
- Valorisation d'écriture par rapport à la lecture.
- Plusieurs formats (principalement netCDF).

# Modes de sortie de `mpp_io_mod`

- Single-threaded I/O: un seul processeur rassemble les données et les écrit.

- Multi-threaded, single-fileset I/O: plusieurs processeurs écrivent à un seul fichier.

- Multi-threaded, multi-fileset I/O: plusieurs processeurs écrivent à des fichiers différents (rassemblés en suite par le *post-processing*).

# mpp_io_mod API

- mpp_io_init()
- mpp_open()
- mpp_close()
- mpp_read()
- mpp_write()
- mpp_write_meta()

# Metadata

Le metadata est l'entete de chaque fichier: Il contient les descriptions assez verboses des grilles, des variables, leurs unités, masques, points de données manquées, packing, etc. L'ensembles de ces attributes sont dans le code en façon compacte dans les types `axistype` et `fieldtype`.

# mpp_open

L'appel principal est `mpp_open()`. Toute information sur le type de I/O à performer vient de là.

```
call mpp_open( unit, file, action=MPP_WRONLY, format=MPP_-
IEEE32, access=MPP_SEQUENTIAL, threading=MPP_SINGLE )
```

Le format est un parmi `MPP_ASCII`, `MPP_IEEE32`, `MPP_NATIVE`, ou `MPP_NETCDF`.

```
call mpp_open( unit, file, action=MPP_WRONLY, format=MPP_-
IEEE32, access=MPP_SEQUENTIAL, threading=MPP_MULTI, fileset=MPP_-
MULTI )
```

# Exemple

```
type(domain2D) :: domain(0:npes-1)
type(axistype) :: x, y, z, t
type(fieldtype) :: field
integer :: unit
character*(*) :: file
real, allocatable :: f(:,:,:)
call mpp_define_domains( (/1,ni,1,nj/), domain )
call mpp_open( unit, file, action=MPP_WRONLY, format=MPP_IEEE32, &
  access=MPP\_SEQUENTIAL, threading=MPP_SINGLE )
call mpp_write_meta( unit, x, 'X', 'km', ... )
...
call mpp_write_meta( unit, field, (/x,y,z,t/), 'Temperature', 'kelvin', ... )
...
call mpp_write( unit, field, domain(pe), f, tstamp )
```

# Un modèle *shallow water*

$$\frac{\eta^{n+1} - \eta^n}{\Delta t} = -H(\nabla \cdot \mathbf{u})^n \quad (1)$$

$$\frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t} = -g(\nabla \eta)^n + f\mathbf{k} \times \left( \frac{\mathbf{u}^{n+1} + \mathbf{u}^n}{2} \right) + \mathbf{F} \quad (2)$$

```
program shallow_water
  type(scalar2D) :: eta(0:1)
  type(hvector2D) :: utmp, u, forcing
  integer tau=0, taup1=1
...
  f2 = 1./(1.+dt*dt*f*f)
  do l = 1,nt
     eta(taup1) = eta(tau) - (dt*h)*div(u)
     utmp = u - (dt*g)*grad(eta(tau)) + (dt*f)*kcross(u) + dt*forcing
     u = f2*( utmp + (dt*f)*kcross(utmp) )
     tau   = 1 - tau
     taup1 = 1 - taup1
  end do
end program shallow_water
```

- Le code ne contient aucune reférence au parallélisme.

- Le code est entièrement écrit en f90.

- Le code permet d'avoir des noyaux computationnels trés optimisés pour l'architecture en question sans sacrifier la portabilité et la lisibilité.

- Le surcoût de langage orienté objet est non-nul mais pas intolérable non plus!

```
type, public :: scalar2D
   real, pointer :: data(:,:)
   integer :: is, ie, js, je
end type scalar2D
```

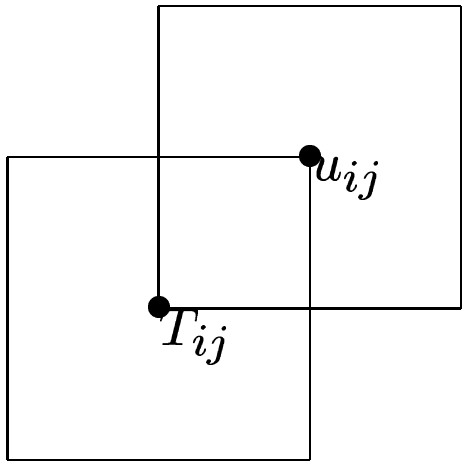
`is, ie, js, je` contient l'information dur le *domaine de validité* de la région de calcul. Les appels du type *mpp_update_domains* se font au fur et à mésure des besoins.

# div and grad

$$\nabla \cdot \mathbf{u} = \delta_x(\overline{u}^y) + \delta_y(\overline{v}^x) \tag{3}$$

$$(\nabla T)_x = \delta_x(\overline{T}^y) \tag{4}$$

$$(\nabla T)_y = \delta_y(\overline{T}^x) \tag{5}$$



```
    function grad_scalar2D(scalar)
      type(hvector2D) :: grad_scalar2D
      type(scalar2D), intent(inout) :: scalar
...
      if( scalar%ie.LE.ie .OR. scalar%je.LE.je )then
          call mpp_update_domains( scalar%data, domain, EUPDATE+NUPDATE )
          scalar%ie = ied
          scalar%je = jed
      end if
      grad_scalar2D%is = scalar%is; grad_scalar2D%ie = scalar%ie - 1
      grad_scalar2D%js = scalar%js; grad_scalar2D%je = scalar%je - 1

!dir$ IVDEP
      do j = grad_scalar2D%js,grad_scalar2D%je
         do i = grad_scalar2D%is,grad_scalar2D%ie
            tmp1 = scalar%data(i+1,j+1) - scalar%data(i,j)
            tmp2 = scalar%data(i+1,j) - scalar%data(i,j+1)
            work2D(i,j,nbuf2) = gradx(i,j)*( tmp1 + tmp2 )
            work2D(i,j,nbufy) = grady(i,j)*( tmp1 - tmp2 )
         end do
      end do
```

# Caractéristiques des opérateurs différentiels

- Les détails numériques sont cachés de la couche supérieure.
- Noyaux optimisés aux besoins.
- Extensible: On peut avoir autant d'algorithmes désirés.
- Plusieures métriques de grilles possibles, fixées à l'initialisation.
- Communication entre processeurs au fur et à mésure des besoins.
- Equilibrage de communication et calcul avec *wide halos*.

# Wide halos

Sur une machine avec un réseau lent, on peut remplacer la communication par des calculs redondants sur plusieurs processeurs:

- Quelques points dans le domaine de validité (défini plus large que nécessaire) sont calculés sur plus qu'un processeur.
- Le domaine de validité est réduit jusqu'au point que le domaine computationnel n'est plus calculable.
- Appel en suite à *mpp_update_domains*. Celui-ci peut arriver qu'une fois sur plusieurs pas de temps.

```
call mpp_define_domains( ..., xhalo=1, yhalo=1 )
call mpp_define_domains( ..., xhalo=6, yhalo=6 )
```

# Conclusions

- GFDL et SGI ont tous les deux profité d'avoir établi un lien étroit entre les développeurs des modèles et les développeurs des bibliothèques.

- Les "standards" tel MPI ne livre pas la performance souhaitée par les chercheurs. C'est devenu un outil qui sert uniquement pour le benchmarking. C'est désirable de converger sur un outil minimaliste spécialisé au domaine météorologique et réalisable dans plusieurs bibliothèques performantes chacune sur sa propre architecture.

- Une convergence de ces outils entre les grandes laboratoires serait une Bonne Chose. Des efforts vers ce but ont commencés.