Le parallélisme dans LMDZ

Petit guide de survie dans un monde parallèle

Formation LMDZ, 27 novembre 2012

NB: Pour quelques détails techniques complémentaires, consulter, sur le site LMDZ:
http://lmdz.lmd.jussieu.fr/developpeurs/notes-techniques/ressources/parallelisme-LMDZ.pdf

Pourquoi paralléliser ?

• Pour diminuer le temps de calcul d'une simulation en répartissant le travail sur plusieurs cœurs travaillant chacun de façon « la plus indépendante possible » (c.-à-d. avec le minimum possible de communication/dépendance d'informations entre cœurs) .

• Pour tirer parti des architectures actuelles (du PC portable aux calculateurs des grands centres de calcul).

Déclinaison du parallélisme dans LMDZ

• LMDZ est conçu pour pouvoir être compilé et exécuté en mode séquentiel ou parallèle (sous plusieurs formes, MPI, OpenMP, mixte MPI/OpenMP, voir plus loin).

• Implémentation pensée pour non seulement pouvoir s'adapter au mieux aux contraintes matérielles, mais aussi développée comme une surcouche la moins intrusive possible pour les utilisateurs/développeurs.

Parallélismes MPI et OpenMP

MPI : Le parallélisme en mémoire distribuée

• Le code à exécuter est répliqué sur l'ensemble des CPUs au sein d'un processus.

• Chaque processus s'exécute indépendamment et n'a pas accès à la mémoire des autres processus.

• Les échanges de données se font à travers **une librairie d'échange de message** (MPICH, OpenMPI, ...) qui utilise le réseau d'interconnexion entre les nœuds du calculateur. Les performances reposent sur la qualité du réseau d'interconnexion.

• On décide du nombre de processus à l'exécution: mpirun -n 8 gcm.e

OpenMP : Le parallélisme en mémoire partagée

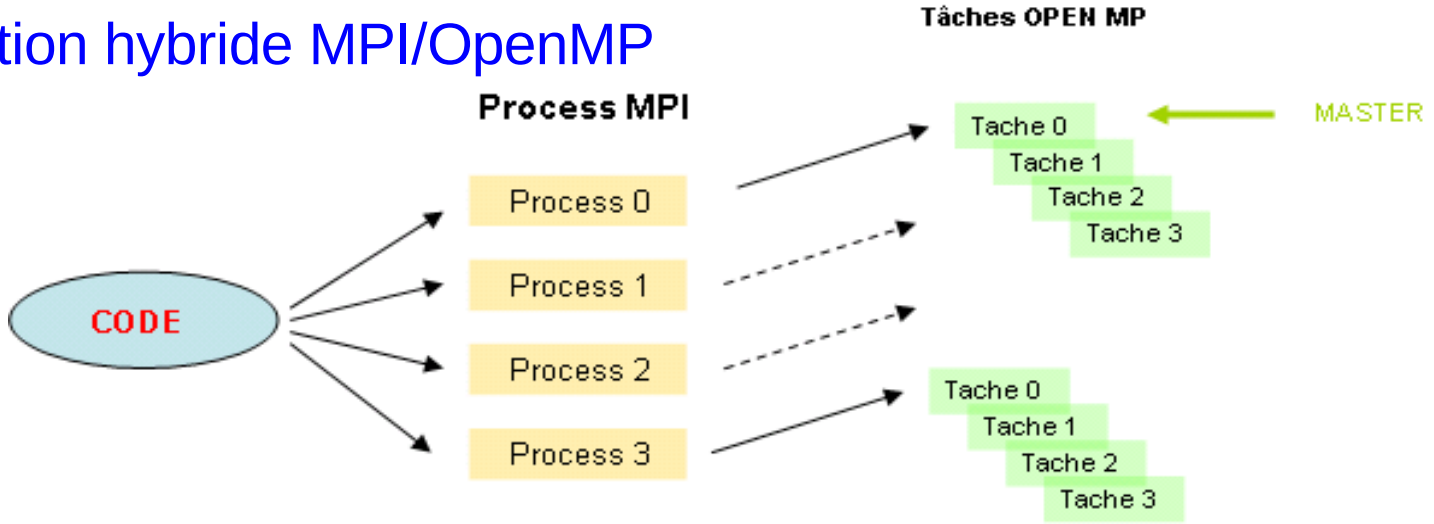
• Ce parallélisme repose sur le principe du multithreading. Plusieurs tâches (threads) s'exécutent concurremment au sein d'un processus.

• Chaque tâche a un **accès partagé à la mémoire globale** du processus.

• Les boucles sont parallélisées à l'aide de directives (!OMP ... , qui apparaissent comme des commentaires au sein du code) interprétées par le compilateur.

• Le nombres de tâches OpenMP à utiliser est fixé par la variable d'environnement OMP_NUM_THREADS (ex: OMP_NUM_THREADS=4)

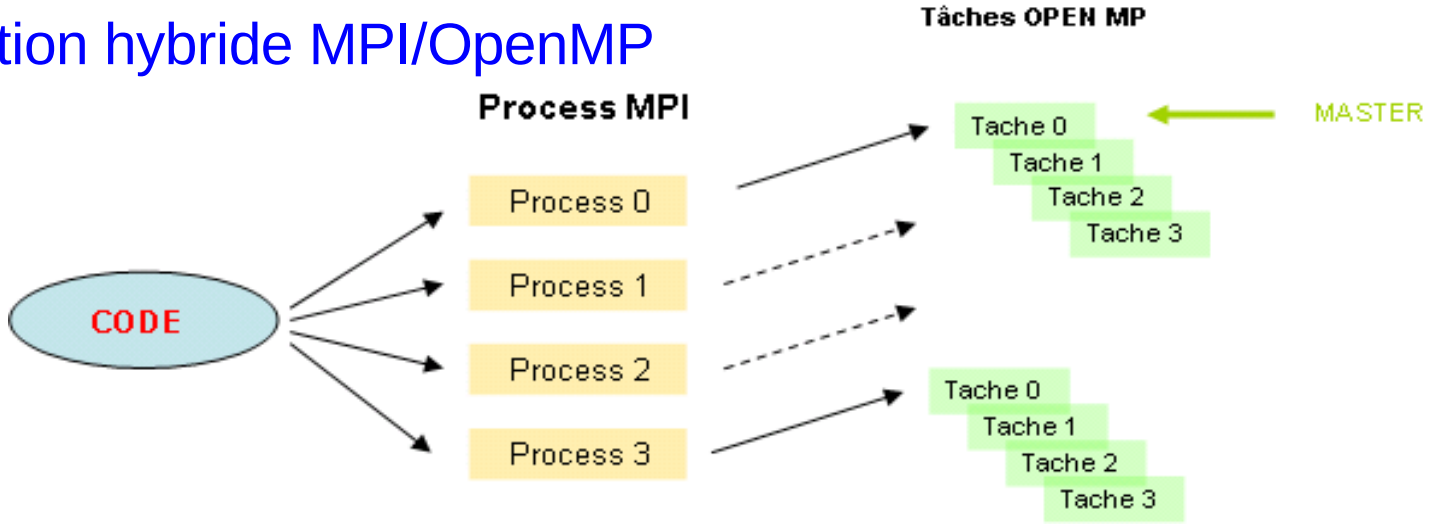
Programmation hybride MPI/OpenMP



- Chaque processus MPI lance des tâches OpenMP dans son espace mémoire

- Dans LMDZ, les implémentations des parallélismes MPI et OpenMP sont distinctinctes.

Programmation hybride MPI/OpenMP



➢ Chaque processus MPI lance des tâches OpenMP dans son espace mémoire

Stratégie de parallélisation différente entre la dynamique et la physique

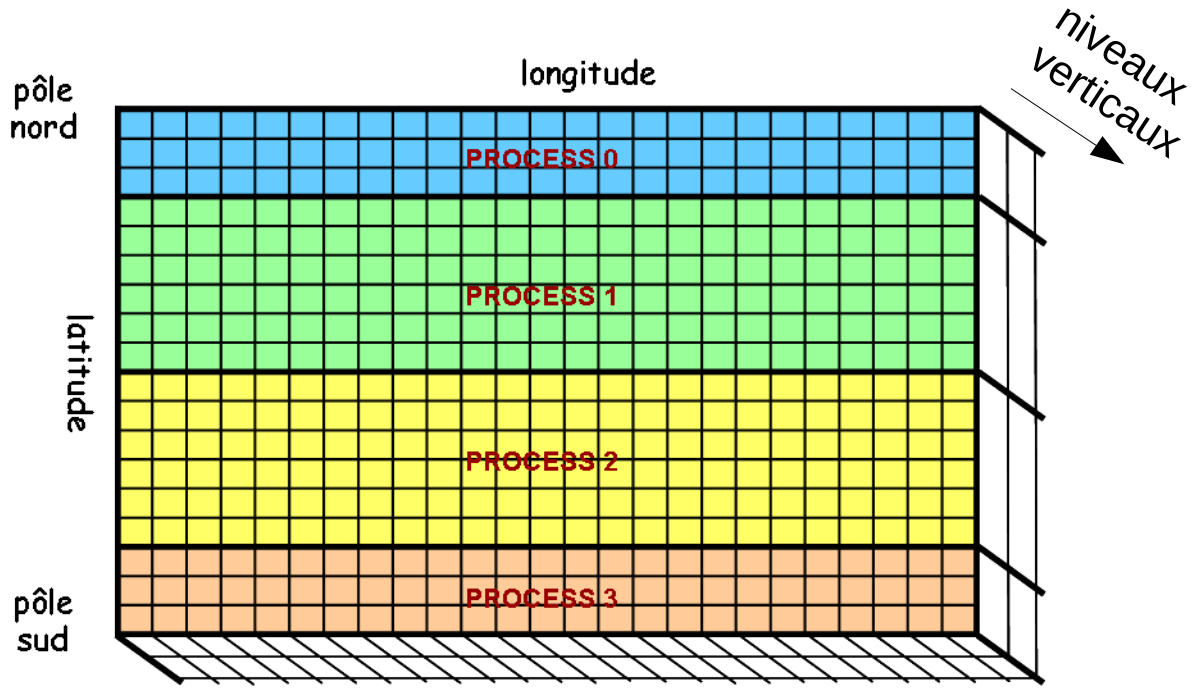
➢ La partie dynamique de LMDZ

Pas de temps très courts; beaucoup d'interactions entre les mailles voisines. Exige de nombreux échanges et synchronisations. Partie délicate de la parallélisation.

➢ La partie physique

Pas de temps plus long; pas d'interactions entre les colonnes d'atmosphères.

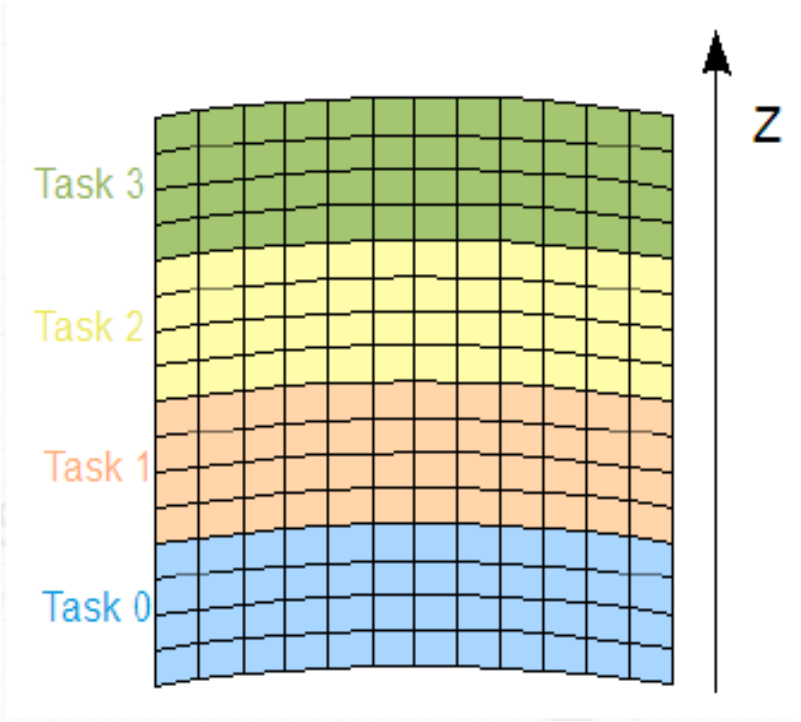
Parallélisme dans la dynamique



=> Découpage MPI

- Découpage par bandes de latitude.
- Il faut **au minimum** 3 bandes de latitude par processus MPI.
- Mais la charge de travail n'est pas la même à toutes les latitudes (principalement à cause du filtre près des pôles).
- Utiliser l'option *adjust=y* (dans *gcm.def*) pour optimiser dynamiquement (en cours d'exécution) la distribution des bandes de latitude par processus.
  - ➔ Exécuter le GCM (en mode MPI seul!) sur au moins plusieurs milliers de pas dynamiques pour obtenir un fichier Bands_**x**x**_*prc.dat
  - ➔ Relancer la simulation avec l'option *adjust=n* (en présence du fichier Bands_*)
    NB: en l'absence d'un fichier Bands_*, le GCM en crée un avec une répartition uniforme

Parallélisme dans la dynamique

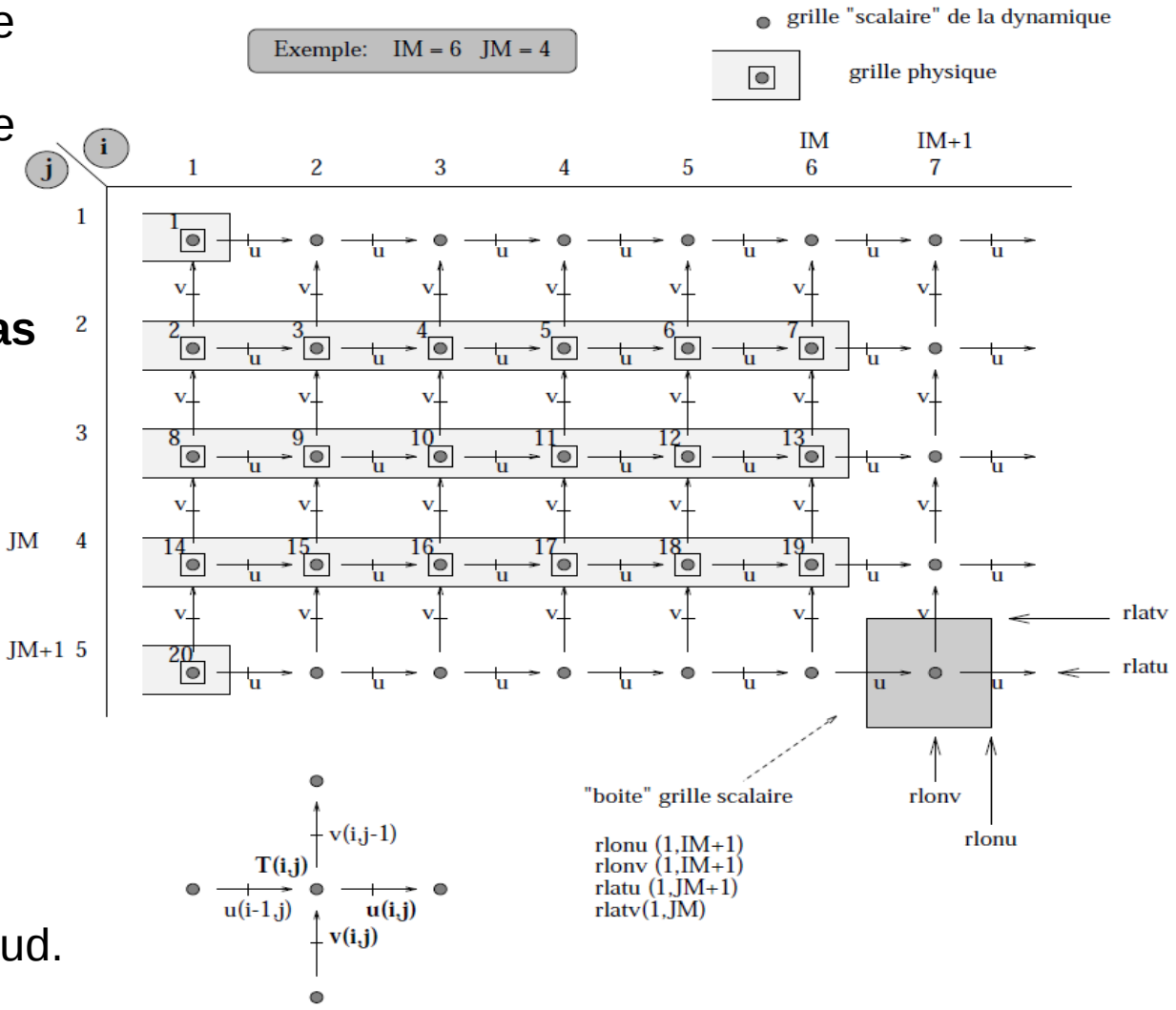


=> Découpage OpenMP

- Découpage suivant les niveaux verticaux (boucle la plus externe dans la grande majorité des calculs).
- Une taille indicative des blocs à attribuer à chacune des tâches peut être précisée via l'option *omp_chunck*=... dans gcm.def.
- En pratique, prévoir une tâche OpenMP pour 4 ou 5 niveaux verticaux (compromis optimal, mais qui peut dépendre de la machine sur laquelle on travaille).

Parallélisme dans la physique

- La partie physique est chargée du calcul de phénomènes physiques interagissant au sein d'une même colonne d'atmosphère : rayonnement, convection, couche limite, etc. …
- Les différentes colonnes d'atmosphère **n'interagissent pas** entre-elles.
- La stratégie de parallélisation consiste donc à distribuer les colonnes d'atmosphère sur les différents cœurs.
- Grille de la physique : klon_glo points géographiques sur klev niveaux verticaux.
  premier point (1) => pôle nord,
  dernier point (klon_glo) => pôle sud.

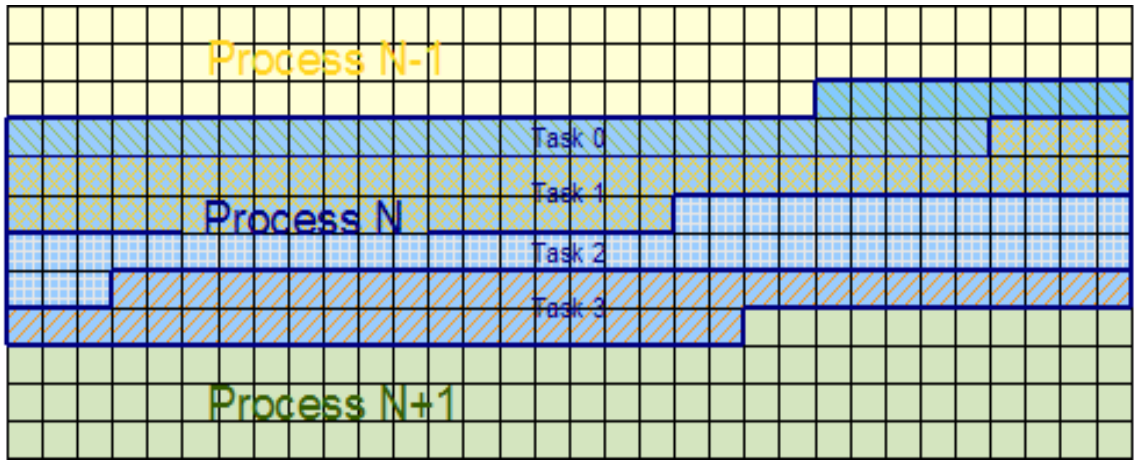


Parallélisme dans la physique

- Les colonnes du domaine global sont d'abord distribuées sur les différents processus MPI.
  - Le domaine global : klon_glo colonnes d'atmosphere

- Les colonnes de chaque domaine MPI sont distribuées sur les tâches OpenMP associées à ce processus :
  - Chaque domaine MPI : klon_mpi colonnes : Σ klon_mpi = klon_glo
  - Chaque domaine OpenMP : klon_omp colonnes : Σ klon_omp = klon_mpi

- En pratique, la taille du domaine local klon **est un alias** de klon_omp (pour ne pas changer les habitudes par rapport au fonctionnement en séquentiel).
  ➔ Ne pas oublier que klon diffère d'un cœur à l'autre.



Quelques définitions des paramètres liés au parallélisme

Grille globale : module mod_grid_phy_lmdz

- klon_glo : nombre de mailles sur l'horizontale du domaine globale (grille 1D)
- nbp_lon : nombre de points en longitude (grille 2D) = iim
- nbp_lat : nombre de points en latitude (grille 2D) = jjm+1
- nbp_lev : nombre de niveaux verticaux = klev ou llm

Grille MPI : module mod_phys_lmdz_mpi_data

- klon_mpi : nombre de points sur le domaine mpi local.
- klon_mpi_begin : indice de départ du domaine sur la grille 1D globale.
- klon_mpi_end : indice de fin du domaine sur la grille 1D globale.
- ii_begin : indice en longitude du début du domaine (grille 2D globale).
- ii_end : indice en longitude de la fin du domaine (grille 2D globale).
- jj_begin : indice en latitude de début de domaine (grille 2D globale).
- jj_end : indice en latitude de fin de domaine (grille 2D globale).
- jj_nb : nombre de bandes de latitude = jj_end-jj_begin+1
- is_north_pole : .true. si le processus inclue le pôle nord.
- is_south_pole : .true. si le processus inclue le pôle sud.
- is_mpi_root : .true. si processus MPI maître.
- mpi_rank : rang du processus MPI.
- mpi_size : nombre de processus MPI.

Quelques définitions des paramètres liés au parallélisme

Grille OpenMP : module mod_phys_lmdz_mpi_data

- klon_omp : nombre de points sur le domaine OpenMP.
- klon_omp_begin: indice de début de domaine OpenMP par rapport au domaine MPI.
- klon_omp_end : indice de fin de domaine OpenMP.
- is_omp_root : .true. si tâche maîtresse OpenMP.
- omp_size : nombre de tâches OpenMP à l'intérieur du processus.
- omp_rank : rang de la tâche OpenMP.

Comment coder dans la partie physique ?

• Rien ne change (par rapport au séquentiel) si :
  - Il n'y a pas d'interaction entre les colonnes d'atmosphère
  - Il n'y a pas de calcul de moyenne globale ou zonale
  - Il n'y a pas de lecture ou d'écriture de fichiers

• Seul impératif pour l'OpenMP : toutes les variables en SAVE ou les « common » doivent se trouver dans une clause !$OMP THREADPRIVATE, par exemple :

```
REAL, SAVE :: save_var
!$OMP THREADPRIVATE(save_var)
```

• Les tableaux sont alloués en taille locale klon, sachant que klon peut varier d'un coeur à l'autre. Par exemple :

```
ALLOCATE (myvar(klon))
```

Attention! En général, myvar(1) ou myvar(klon) ne représente ni le pôle sud, ni le pôle nord.

• Faire appel aux variables booléens, is_north_pole et is_south_pole si on a besoin d'un traitement spécifique sur les pôles

Comment coder dans la partie physique ?

- Attention si :
  - interaction entre les colonnes d'atmosphère
  - calcul de moyenne globale ou zonale
  - lecture ou écriture de fichiers

=> Dans ces cas, il faut faire des transferts entre les différents processus MPI et les tâches OpenMP:

- Toutes **les routines de transfert sont encapsulées** et gèrent de façon transparente les transferts entre les processus MPI et entre les tâches OpenMP.

- Toutes sont dans le module : mod_phys_lmdz_transfert_para

Les interfaces de transfert acceptent indifféremment les principaux types de base :
REAL
INTEGER
LOGICAL
CHARACTER : uniquement le broadcast

Les interfaces acceptent indifféremment les champs de 1 à 4 dimensions.

Les transfert de données dans la physique (suite)

Broadcast : le processeur maître duplique ses données sur les autres processus/tâches.

Indépendant des dimensions de la variable

CALL bcast(var)

Scatter : la tâche maîtresse possède un champ sur la grille globale (klon_glo) qu'elle distribue sur la grille locale (klon).

La 1ère dimension du champs global doit être klon_glo, et celle du champ local klon

CALL scatter(field_glo,field_loc)

Gather : un champ défini sur la grille locale (klon) est rassemblé sur la grille globale de la tâche maîtresse (klon_glo).

La 1ère dimension du champ global doit être klon_glo, et celle du champ local klon

CALL gather(field_loc,field_glo)

Scatter2D : même chose que Scatter sauf que le champ global est défini sur la grille 2D : nbp_lon x nbp_lat.

La 1ère et 2ème dimension du champs global doit être (nbp_lon,nbp_lat), et celle du champ local klon

CALL scatter2D(field2D_glo,field1D_loc)

Gather2D : rassemble les données sur la grille 2D de la tâche maîtresse.

CALL gather2D(Field1D_loc,Field2D_glo)

Quelques exemples

La lecture des fichiers de paramètres de la physique : physiq.def et config.def

- Effectuée dans conf_phys.F90
- La tâche maître lit la valeur (call getin('VEGET',ok_veget_omp) dans une région !$OMP MASTER) puis la duplique sur les autres tâches (après le !$OMP END MASTER , on a un ok_veget = ok_veget_omp)

La lecture des champs du fichier de démarrage : startphy.nc

- Effectuée dans phyetat0.F
- La tâche maîtresse sur le processus maître (if(is_mpi_root.and.is_omp_root)) lit le champ sur la grille globale puis le redistribue sur la grille locale à l'aide d'un scatter
- Encapsulé dans la routine get_field du module iostart

L'écriture des champs dans le fichier de démarrage : restartphy.nc

- Effectuée dans phyredem.F
- Encapsulé dans la routine put_field du module iostart. put_field effectuera d'abord un gather pour rassembler les champs locaux dans un champ global. Pui la tâche maîtresse sur le processus maître écrit le champ dans le fichier de redémarrage restartphy.nc

Exemple de lecture d'un fichier, extrait (simplifié) de phylmd/read_map2D.F90

```
USE dimphy
USE netcdf
USE mod_grid_phy_lmdz
USE mod_phys_lmdz_para
...

REAL, DIMENSION(nbp_lon,nbp_lat) :: var_glo2D
REAL, DIMENSION(klon_glo)        :: var_glo1D
REAL, DIMENSION(klon)            :: varout

! Read variable from file. Done by master process MPI and master thread OpenMP
  IF (is_mpi_root .AND. is_omp_root) THEN
     NF90_OPEN(filename, NF90_NOWRITE, nid)
     NF90_INQ_VARID(nid, varname, nvarid)

     start=(/1,1,timestep/)
     count=(/nbp_lon,nbp_lat,1/)
     NF90_GET_VAR(nid, nvarid, var_glo2D,start,count)
     NF90_CLOSE(nid)

     ! Transform the global field from 2D to 1D
     CALL grid2Dto1D_glo(var_glo2D,var_glo1D)
  ENDIF

! Scatter gloabl 1D variable to all processes
  CALL scatter(var_glo1D, varout)
```

L'écriture des fichiers de sortie d'IOIPSL et reconstruction

- Chaque processus MPI écrit la partie de son domaine dans un fichier distinct. On obtient donc autant de fichiers histmth_00XX.nc que de processus MPI on été utilisés pour réaliser la simulation.

- Le domaine du fichier IOIPSL est défini au moment de l'appel à histbeg, encapsulée dans histbeg_phy (module iophy.F).

- Les données sont rassemblées sur la tâche maîtresse OpenMP (de rang 0) de chaque processus. Puis chaque processus MPI appelle la routine histwrite d'IOIPSL, encapsulée dans histwrite_phy (module iophy.F).

- **Attention**: ce qui est dit ci-dessus est valable pour **les sorties dans la physique**; il est également possible de faire des sorties dans la dynamique (actionnées via ok_dyn_ins et ok_dyn_ave dans *run.def*), mais pour ces dernières, les données sont en plus rassemblées sur le processus maître pour sortir un seul fichier (ce qui plombe les perfs.) => à n'utiliser que pour du debug.

L'écriture des fichiers histoires d'IOIPSL et reconstruction

Une fois la simulation terminée, on doit recombiner les données dans un fichier global. Pour ceci, on utilise l'utilitaire rebuild:

```
rebuild -o histmth.nc histmth_00*.nc
```

- rebuild est un programme distribué avec IOIPSL
  Voir « Comment installer IOIPSL et l'outil rebuild » sur la FAQ du site LMDZ (http://lmdz.lmd.jussieu.fr/utilisateurs/faq)

- Sur les centres de calculs IDRIS et CCRT, rebuild est disponible déjà compilé dans un répertoire avec des outils communs :

IDRIS

Ulam	/home/rech/psl/rpsl035/bin/rebuild
Vargas	/homegpfs/rech/psl/rpsl035/bin/rebuild

CCRT

Titane	/home/cont003/p86ipsl/X64/bin/rebuild
Mercure	/home/cont003/p86ipsl/SX8/bin/rebuild
Curie	/home/cont003/p86ipsl/X64/bin/rebuild

Quelques réflexions et aspects pratiques en vrac

- Pour tourner en « local » (sur un PC/portable multicœur):
  - Avoir une librairie MPI installée; avoir adapté les fichiers « arch » pour la compilation du modèle: 'makelmdz_fcm -arch arch-local ...'
  - Penser à se donner les coudées franches côté utilisation de la mémoire: ulimit -s unlimited
  - Penser également à réserver de la mémoire pour les tâches OpenMP: export OMP_STACKSIZE=200M
  - Utiliser 'mpiexec -np n ...' pour exécuter n processus et 'export OMP_NUM_THREADS=m' pour utiliser m tâches OpenMP
  - Quelques conseils et infos sont données ici: http://lmdz.lmd.jussieu.fr/utilisateurs/guides/lmdz-parallele-sur-pc-linux

- Pour tourner sur des clusters (Climserv, Ciclad, Gnome, ...) et machines des mésocentres et centres nationaux (IDRIS, CCRT,..):
  - Consulter la documentation relative à chaque centre pour savoir comment préciser le nombre de processus MPI, tâches OpenMP, les limitations (temps et mémoire) des jobs à soumettre en batch, etc.
  - Quelques infos sur les en-têtes de jobs pour les machines « habituelles » utilisées à l'IPSL sont données ici: https://forge.ipsl.jussieu.fr/igcmg/wiki/IntegrationOpenMP

Exemple d'un job (loadleveler) pur MPI (ici sur Vargas)

> cat Job_MPI

```
# ########################
# ##   VARGAS IDRIS   ##
# ########################
# @ job_name = test
# @ job_type = parallel
# @ output = $(job_name).$(jobid)
# @ error =  $(job_name).$(jobid)
# Nombre de processus MPI demandes
# @ total_tasks = 32
# Temps CPU max. par processus MPI hh:mm:ss
# @ wall_clock_limit = 0:30:00
# Memoire max. utilisee par processus (stack+data<=3.7gb)
# @ data_limit = 3.2gb
# Memoire stack demandee
# @ stack_limit = 0.5gb,0.5gb
# Fin de l entete
# @ queue

./gcm.e  > gcm.out  2>&1
```

> llsubmit Job_MPI

Exemple d'un job (loadleveler) mixte MPI/OpenMP (ici sur Vargas)

> cat Job_MPI

```
# ########################
# ##   VARGAS IDRIS   ##
# ########################
# @ job_name = test
# @ job_type = parallel
# @ output = $(job_name).$(jobid)
# @ error =  $(job_name).$(jobid)
# Nombre de processus MPI demandes
# @ total_tasks = 32
# Nombre de tâches OpenMP par processus MPI
# @ parallel_threads = 4
# Temps CPU max. par processus MPI hh:mm:ss
# @ wall_clock_limit = 0:30:00
# Memoire max. utilisee par processus (stack+data<=3.7gb par thread)
# @ data_limit = 12.8gb
# Memoire stack demandee
# @ stack_limit = 2.0gb,2.0gb
# Fin de l entete
# @ queue

# Mémoire stack propre à chaque thread, ici 512 Mo
Export XLSMPOPTS=$XLSMPOPTS:stack=524288000
./gcm.e  > gcm.out  2>&1
```

> llsubmit Job_MPI

libIGCM

• L'environnement modipsl/libIGCM de gestion des modèles IPSL (LMDZOR, LMDZORINCA, IPSLCM5, LMDZREPR, ORCHIDEE_OL) est une surcouche facilitant l'extraction, l'installation, la mise en place et l'exécution de ces modèles sur les machines « usuelles » de l'IPSL.

• L'entête du job, et certain paramètres sont à modifier; par exemple la mémoire et le temps limite d'exécution.

• Les reconstructions des sorties sont lancées automatiquement a posteriori.

• Il y a des formations modipsl/libIGCM régulièrement (organisées par Josefine Ghattas & Anne Cozic) ; la prochaine (en anglais) est le 29 novembre 2012, à Jussieu (LMD, 3eme étage, salle 313, 10h-12h30; après-midi: séances d'exercices et questions-réponses).

En guise de résumé

► Dans la physique, tant qu'il n'y a pas de communications entre colonnes, on peut développer « comme en séquentiel ». Seul Impératif (lié à l'OpenMP): les variables en SAVE sont à déclarer en !$OMP THREADPRIVATE.

➔ Penser à tester l'impact des modifications! On doit retrouver exactement le même résultat, au bit près, lorsqu'on change le nombre de processeurs ou le nombre de tâches OpenMP (au moins en mode 'debug').

► Dans la partie dynamique, le parallélisme est bien plus intrinsèque; là il faut vraiment bien comprendre l'ensemble avant de toucher au code.

► On compile en mode parallèle de son choix: mpi, omp ou mpi_omp
makelmdz_fcm -parallel mpi ..... , makelmdz_fcm -parallel mpi_omp .....

► On lance l'exécution, sur le plus de cœurs disponibles, sans oublier que pour une grille donnée, le nombre maximum de processus MPI = nombre de points en latitude / 3 et qu'on est optimal en utilisant environ 1 tâche OpenMP pour 4 ou 5 points suivant la verticale.

► Pour optimiser le découpage sur les différents processus MPI, on tourne un premier mois avec *adjust=y* dans run.def. On obtient un fichier bands_resol_Xprc.dat qu'on ré-utilisera pour la reste des simulations.

► On reconstruit les fichiers de sorties : rebuild -o histmth.nc histmth_00*.nc