LE PARALLELISME DANS LMDZ

Panorama du monde HPC

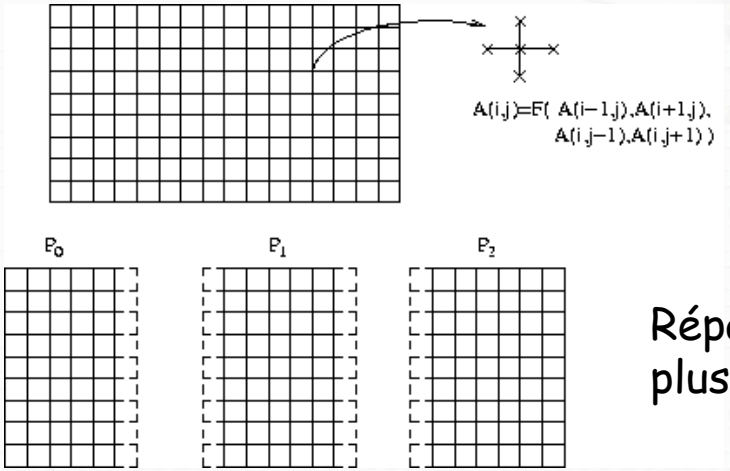
- HPC = High Performance Computing
- Unité de mesure de performance des performance d'un processeur : GFlops
  - Milliard d'opérations de flottant effectuées par seconde.
- Supercalculateur : agrégation de la puissance des processeurs à travers un réseau d'interconnexion => vision d'une machine unique.
  - Puissance crête de la machine = puissance d'un proc. * nombre de proc.
- Les supercalculateurs d'aujourd'hui sont Pétaflopiques : un million de milliard d'opérations /sec : classement au top 500 (http://www.top500.org) :
  - 1er : Los Alamos (US) : IBM Cell – Roadrunner : 1,105 PétaFlops
  - 2ème : OakRidge (US) : Cray XT5 – Jaguar : 1,059 Pétaflops
- Différentes architectures de calcul
  - Architectures vectorielles : NEC SX, CRAY X1 : ~ 100 Gflops/proc.
  - Cluster scalaire : Intel Itanium et Xéon, AMD Opteron, IBM Power6,… (bicoeur ou quadricoeur) : 5~20 Gflops/cœurs.
  - Architectures MPP : IBM BlueGene, Cray XT
  - Architectures exotiques :
    - Processeur IBM Cell : 256 Gflops
    - Processeur de cartes graphiques (GPU) : Nvidia ou AMD/ATI : 1500 Glops/GPU.

Les moyens de calcul en France

- IDRIS (GENCI/CNRS) :
  - **Brodie : 80 processeurs NEC SX8, 1.3 TFlops.**
  - **Vargas : IBM, 3584 cœurs Power6, 67.3 TFlops.**
  - Babel : IBM, 40960 cœurs Blue Gene/P : 139 TFlops.
- CCRT (GENCI/CEA)
  - **Mercure : 64 proc. NEC SX8-R, 2.2 TFlops.**
  - **Platine : Bull : 7456 cœurs Itanium 2 : 47.7 TFlops.**
  - Extension prévu en 2009
    - mai 2009 : **48 proc. NEC SX9 : 4.8 TFlops.**
    - mi 2009 : **8544 cœur Intel Nehalem : 103 TFlops.**
    - mi 2009 : 48 Serveur Tesla Nvidia (GPU) : 192 TFlops.
- CINES (GENCI/Enseignement Supérieur)
  - **Jade : SGI : 12 288 cœurs Xéon, 147 TFlops.**

Quelques notions de parallélisme

- Le parallélisme requiert de faire fonctionner de manière synchronisée un ensemble de processeurs sur une même tâche.
  - Permet de répartir des calculs longs sur un ensemble de processeurs disponibles afin de diminuer le temps de restitution.
  - Permet de répartir un volume de données important sur un ensemble de processus, allégeant ainsi les besoins en mémoire propres à chaque processus.
- Ex : Problème de calcul en différences finies



Répartition du domaine de calcul sur plusieurs processeurs

- Chaque processeur peut calculer sur une partie du maillage en ayant la connaissance des mailles frontières des processeurs voisins.
  - Nécessite de distribuer les données en fonction des processeurs
  - Nécessite d'échanger des données entre processeurs.

Les modèles de programmation parallèle

- Il existe deux grands standards normalisés : MPI et Open MP. Les anciens standards (PVM, shmem, HPF, etc..) sont en perte de vitesse.

- Le parallélisme en mémoire distribuée (SPMD) => MPI
  - 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 qui utilise le réseau d'interconnexion entre les nœuds du calculateur.
    - Les performances reposent sur la qualité du réseau d'interconnexion.
  - Modèle de programmation adapté à tous les calculateurs du marché.

- Le parallélisme à mémoire partagée (SMP) => OpenMP
  - Le parallélisme repose sur le principe du multithreading. Plusieurs thread s'exécutent concurremment au sein d'un processus.
  - Chaque thread a un accès partagé à la mémoire globale du processus.
  - Les boucles sont parallélisées à l'aide de directives.
  - Modèle de programmation limité aux machines SMP, uniquement à l'intérieur d'un nœud.

Le modèle de programmation MPI

- MPI = Message Passing Interface.
  - Les processus se synchronisent et communiquent par passage de messages.
  - Ces communications s'effectuent à travers des appels de fonctions incluses dans la librairie MPI fournis par le constructeur.
    - Nécessité d'inclure la librairie lors de l'édition de lien (-lmpi).
  - Les messages transitent à travers le réseau d'interconnexion. Dans le cas d'une machine SMP, les messages s'échangent à travers la mémoire partagée.
  - Chaque processus exécute le même code.
  - L'exécution du code parallèle demande un utilitaire fournis par le constructeur qui coordonne le lancement sur chacun des nœuds.
    - Ex : `mpirun -np 8 toto.exe` : lance 8 processus du code toto.exe sur chacun des processeurs disponibles.
- Chaque processus MPI est identifié par son rang au sein d'un communicateur.
  - Permet de distribuer les donnée et d'effectuer les calculs en fonction du rang du processus
- Les processus ont la possibilité de communiquer entre eux par envoi de messages au sein d'un même communicateur.

Le modèle de programmation MPI (suite)

Programme MPI minimaliste

```
program WhoAmI
   implicit none
   include 'mpif.h'
   integer  :: rank, size, ierr

   call MPI_Init(ierr)
   call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
   call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
   print '("Salut, je suis le processus ",i2,"/",i2)', rank, size
   call MPI_Finalize(ierr)

end program WhoAmI
```

```
> sxmpif90 -o whoami whoami.f90
> mpirun -np 3 whoami
Salut, je suis le processus 0/3
Salut, je suis le processus 1/3
Salut, je suis le processus 2/3
>
```

Le modèle de programmation MPI (suite)

- Les communications point à point.
  - Permet l'envoi d'un message du processus de rang A vers un processus de rang B.
  - Le message peut-être un scalaire ou un tableau de n'importe quel rang, des types de base fortran : **integer -> MPI_INTEGER**, **real -> MPI_REAL**, etc. …
  - A chaque requête d'envoi du processus A (Send) doit correspondre une requête de réception du processus B.
- Synopsis des appels :
  - Envoi : **MPI_Send(Buff, count, datatype, dest, tag, comm, ierr)**
  - Réception : **MPI_Recv(Buff, count, datatype, source, tag, comm, status, ierr)**
  - Exemple : envoi d'un tableau de 100 entiers de A à B

```
Proc A :    CALL MPI_Send(Tab,100,MPI_INTEGER,rank_b,0,MPI_COMM_WORD,ierr)
                                          ⇩
Proc B :   CALL MPI_Recv(Tab,100,MPI_INTEGER,rank_a,0,MPI_COMM_WORD,status,ierr)
```

- Les requêtes peuvent être bloquante/non bloquante, synchrone ou bufférisée, en fonction du coût calcul / communication / mémoire :
  - **MPI_Isend/MPI_Irecv** : non-bloquante
    - Permet le recouvrement calcul-communication
  - **MPI_Bsend** : envoi bufférisé, rend la main dès que la donnée est dans le buffeur.
  - **MPI_Ssend** : envoi synchrone : évite la consommation de mémoire supplémentaire.

Le modèle de programmation MPI (suite)

- Les communications collectives : effectue en un appel une synchronisation ou un transfert sur l'ensemble des processus d'un communicateur.
  - Plus simple et plus efficace qu'un grand nombre de communications point à point.
  - **`MPI_Barrier`** : synchronise tous les processus d'un communicateur.
  - **`MPI_Broadcast`** : Un processus duplique ces données sur tous les autre processus.
  - **`MPI_Scatter`** : Un processus distribue ces données aux autres processus.
  - **`MPI_Gather`** : Les données des processus sont rassemblées sur un seul processus.
  - **`MPI_Reduce`** : effectue une réduction (somme, produit, min., max.,…) sur les données de chacun des processus. Le résultat est récupéré sur l'un des processus.
- Les avantages / inconvénients de la programmation MPI
  - Existe sur tout type de machine, permet du parallélisme massif jusqu'à un grand nombre de processeurs.
  - Maîtrise complète de la parallélisation.
  - Mais nécessite de prendre en compte la globalité du code pour le paralléliser.
- En savoir plus…
  - Cours de l'IDRIS : http://www.idris.fr/data/cours/parallel/mpi/choix_doc.html
  - Document de la norme MPI : http://www-unix.mcs.anl.gov/mpi

Le modèle de programmation OpenMP

- OpenMP utilise un parallélisme multi-threads : un processus en mémoire partagée peut créer plusieurs processus légers appelés threads.
  - L'exécution de ces processus légers ou threads, ou encore tâches, se fait dans l'espace mémoire du processus d'origine.
  - Le lancement du code se fait classiquement en ligne de commande : `./myexe`
  - Le code débute en séquentiel, puis, lors de l'ouverture d'une section parallèle, les threads sont lancés (fork) jusqu'à la fin de la section (join).
    - Plusieurs sections parallèles peuvent exister dans un même code.
  - Le nombre de threads est piloté par la variable d'environnement `OMP_NUM_THREADS`

- L'exécution concurrente des threads va permettre une exécution parallèle du programme d'origine.
  - Ce sont les boucles qui vont être partagées entre les différents threads. La philosophie est essentiellement la même que pour la vectorisation.
  - Le partage des boucles est activé à l'aide de directives OpenMP ajoutées dans le code.
  - Toutes les variables avec l'attribut `SAVE`, sont partagées en mémoire par l'ensemble des threads (attribut public).
    - Les threads ont une vision globale du domaine de calcul mais ne travaillent que sur une partie de celui-ci.

Exemple de code OpenMP

```
program Hello
   integer A(100), B(100), C(100)
   integer tid, i
!  Zone sequentielle
   ...
!  Lancemement zone parallele
!$OMP PARALLEL PRIVATE(size,num)
   num  = omp_get_thread_num()
   size = omp_get_num_thread()
   print *,'Hello, je suis le thread No',num

! Boucle parallélisée
!$OMP DO
   do i=1,100
      C(i) = A(i) + B(i)
   enddo

! Appel d'une subroutine par chacun des threads
  call mysubroutine(size, num,… )
   ...
!$OMP END PARALLEL
```

Les principales directives…

- **!$OMP PARALLEL/END PARALLEL** : ouvre une section parallèle

- **!$OMP BARRIER** : synchronise les threads et met à jour les variables en mémoire (FLUSH).

- **!$OMP MASTER/END MASTER** : seul le thread maître exécute la section.

- **!$OMP SINGLE/END SINGLE** : un seul thread exécute la section.

- **!$OMP CRITICAL/END CRITICAL** : un seul thread à la fois peut rentrer dans la section, les autres attendent leur tour.

- **!$OMP THREADPRIVATE**([variable ou common]) : permet à une variable **SAVE** d'obtenir le statut privé. Chaque thread possède sa propre copie de la variable en mémoire.

Le modèle de programmation OpenMP (suite)

Les avantages de la programmation OpenMP

- Parallélisation facile à mettre en œuvre : compilation avec une option du compilateur : ex. sur NEC SX : **`-P openmp`**.
  - Suporté par la plupart des compilateurs Fortran (y compris open-source).
- Exécution simple comme un code séquentiel.
  - Intéressant sur des processeurs multicoeurs.
- Parallélisation simple et intuitive suivant les mêmes principes que la vectorisation.
- Parallélisation progressive du code en ajoutant des directives au fur et à mesure des boucles.
- Sur un compilateur non open-MP les directives sont comprises comme des commentaires.
  - le code est compilé en séquentiel de manière transparente.

Les inconvénients…

- Parallélisation limitée au nombre de processeurs sur le nœud.
- Les coûts de synchronisation cachés rendent peu performant la parallélisation sur un grand nombre de processeurs.
- Mêmes inconvénients que la vectorisation : pas de dépendance ni de récursion dans les boucles.

Principe d'efficacité dans le parallélisme

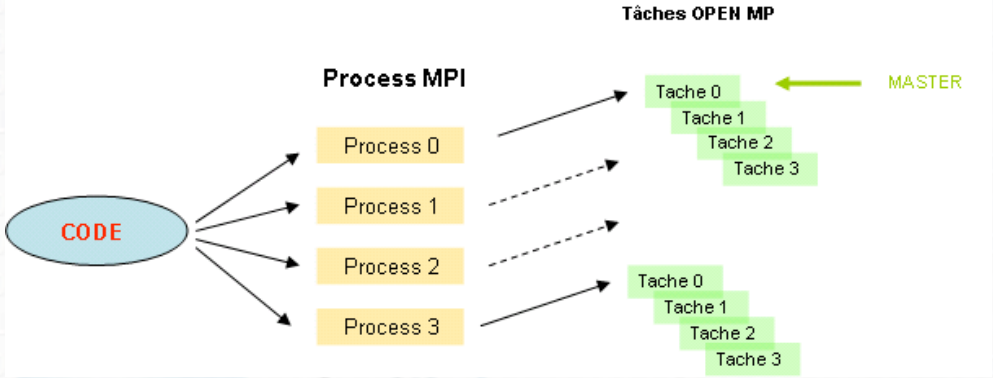
- Principes généraux à avoir en tête :
  - La loi d'Ahmdahl : $$S = \frac{1}{(1-p)+p/N} \Rightarrow p = 0.99, N \longrightarrow \infty, S = 100$$

    S : Accélération (speedup), p : portion du code parallélisé, N : nombre de processeurs
  - L'équilibrage de charge.
    - Les processus les plus rapides attendent les plus lents…
- En MPI
  - Minimiser le coût des transferts.
    - Frontière des domaines minimum.
  - Minimisation des coûts de latence.
    - Minimiser le nombre d'appel, regrouper les appels.
  - Recouvrement des communications par du calcul.
    - utiliser les communications asynchrones non bloquantes : `MPI_Issend`, `MPI_Irecv`.
- En OpenMP
  - Limiter les régions parallèles.
    - Lancement d'une unique section parallèle en début de code.
  - Paralléliser sur les boucles les plus externes.
    - boucles plus longues, meilleure efficacité, n'entre pas en conflit avec la vectorisation.
  - Diminuer le coût des synchronisations
    - Utilisation de la clause `NOWAIT` après une direction `!$OMP DO`
    - Attention, on perd la cohérence de la mémoire entre les threads.

La parallélisation de LMDZ

La parallélisation de LMDZ

- La parallélisation de LMDZ repose sur les deux standards MPI et OpenMP.
- Programmation hybride MPI/OpenMP
  - Chaque processus MPI lance des threads OpenMP dans son espace mémoire



  - Dans LMDZ, les niveaux de parallélisme en MPI et OpenMP ne sont pas les mêmes.
- Le coeur dynamique de LMDZ
  - Pas de temps très courts, beaucoup d'interaction entre les mailles voisines.
    - exige de nombreux échanges et synchronisations.
    - partie délicate de la parallélisation.
- La partie physique
  - Pas de temps long (~15 à 30 min), aucune interaction entre les colonnes d'atmosphères.
- Stratégie de parallélisation différente entre la dynamique et la physique.

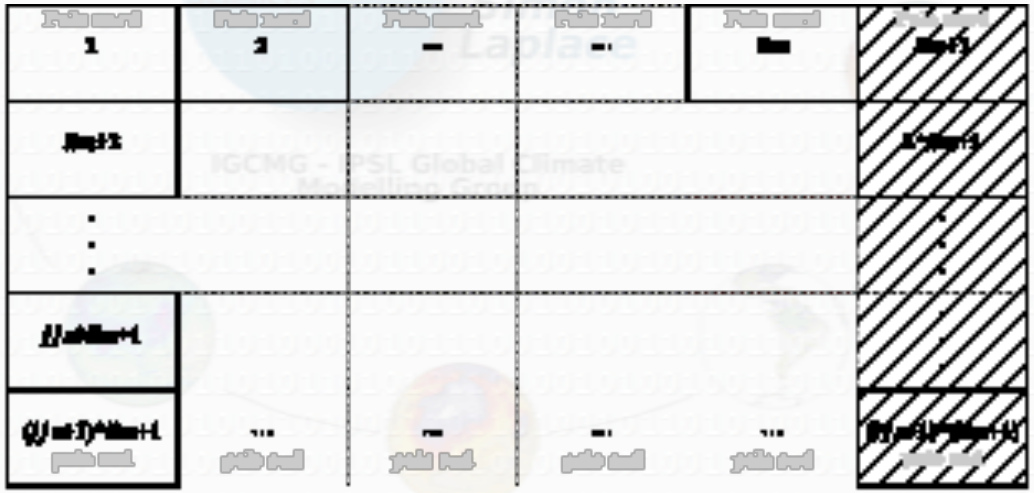
La Parallélisation de la dynamique

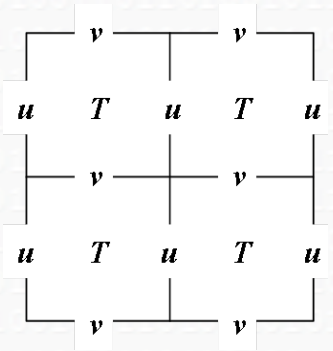
Structure du code

Principales routines

- leapfrog : intégration du schéma temporel (saute-mouton + matsuno).
- caldyn : calcule les termes et tendances des équations primitives (à chaque pas de temps).
- caladvtrac : effectue le transport des traceurs (tous les 5 pas de temps).
- dissip : effectue la dissipation (Laplacien itéré) (tous les 15~30 pas de temps).
- calfis : appel vers la physique (tous les 10~20 pas de temps).
- integre : intègre les tendances de caldyn.

Grille (longitude, latitude, niveaux verticaux) : iim x jjm x llm





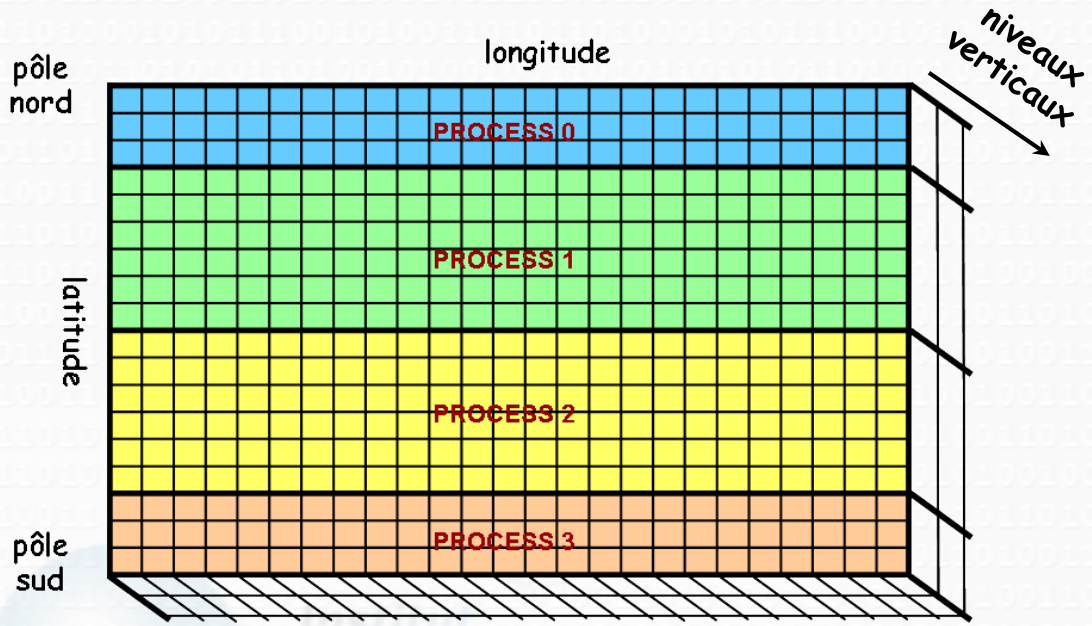
Parallélisation MPI : découpage en domaine

- Le découpage choisi est un compromis entre plusieurs impératifs :
  - Granularité la plus fine, de façon à maximiser le nombre de processus tournant sur le domaine global.
  - Frontière minimum entre les domaines, de façon à limiter les besoins en bande passante lors des communications.
  - Minimisation du nombre de communications MPI, pour limiter les coût de latence.
  - Équilibrage de charge entre les processus.
- Le découpage sur la verticale n'a pas été retenu en premier lieu :
  - Granularité trop faible (19 niveaux), non extensif à un grand nombre de processus.
  - La partie physique implique une multitude d'interactions entre les mailles d'une même colonne d'atmosphère.
    - Plus judicieux d'envisager un découpage du domaine horizontal.
- Découpage du domaine horizontal en blocs (longitude x latitude)
  - Minimise la frontière des domaines.
    - minimise le coût des transferts.
  - Pavage du plan optimum en nombre de processus.
    - optimise la granularité.

MPI : Stratégie de parallélisation

- Oui mais,
  - Perte de la vectorisation (à moins de relinéariser le domaine).
  - Problème du découpage du filtre.
    - Nécessaire pour stabiliser le modèle qui ne respecte pas les conditions CFL aux pôles dû aux resserrement des méridiens.
    - Appliqué sur 1/3 de la surface du globe.
    - filtre uniquement les points d'une même latitude (filtre 1D).
    - Appelé à chaque utilisation d'un opérateur différentiel (donc très souvent).
    - Très consommateur en CPU (de 20 à 50% du coût total du code).
    - Opération matrice/vecteur ou FFT 1D, impossible à paralléliser efficacement (des tentatives ont déjà été effectuées).
  - Problème dans l'advection : un traceur peut être advecté en longitude sur plusieurs mailles voisines.
    - Dépend des champs de vent, non déterminé à l'avance.
    - Complique le schéma de communication.
    - Ajoute des communications inutiles.
- Finalement, on retient un découpage du domaine en bande de latitudes.

MPI : Stratégie de parallélisation



- Avantage
  - Règle implicitement le découpage du filtre puisqu'il ne s'applique que sur les mailles de même latitude.
  - Règle également le problème d'advection suivant la longitude.
  - La structure du code reste la même, les boucles sont toujours linéarisées, on garde de bonnes performances sur la vectorisation.
  - Au plus 2 domaines voisins (minimise le nombre d'envois).
- Inconvénient
  - Limite le nombre maximum de processeurs calculant sur le domaine global.
    - Limitation levée par le niveau supplémentaire de parallélisme en OpenMP.

- Définition des paramètres décrivant le domaine local :
  - Le module **parallel**
  - **integer :: jj_nb_para(mpi_size)** : nb de bandes de latitude pour chaque processus
    - Permet de définir tous les autres paramètres du domaine.
  - **integer :: jj_begin** : indice de début du domaine en latitude.
  - **integer :: jj_end** : indice de fin de domaine en latitude.
  - **integer :: jj_nb** : nombre de bandes de latitude.
  - **integer :: ij_begin** : indice de début de domaine pour les boucles linéarisées en 1D.
  - **integer :: ij_end** : indice de fin de domaine (linéarisé en 1D).
  - **logical :: pole_nord** : .TRUE. si le processeur traite le pole nord.
  - **logical :: pole_sud** : .TRUE. Si le processeur traite le pole sud.
    - Initialisé au début du code
- Chaque processus possède en mémoire l'ensemble du domaine global.
  - Liés aux difficultés d'équilibrage de charge dues à l'application du filtre.
  - Les champs sont toujours déclarés en taille globale.
  - Pas de distribution de la mémoire, mais la priorité première était sur le speed-up.
  - Pour les hautes résolutions (1/2°) et sur Blue Gene, le problème commence à se poser.
  - Amené à changer à l'avenir.
- Chaque processus ne calcule que sur la partie des champs définie par les indices de début et de fin de domaine.

Exemple : en séquentiel

- Cas 2D : `real :: ucov(iip1,jmp1,llm), vcov(iip1,jjm,llm),rot(iip1,jjm,llm)`

```
DO l=1,llm
  DO j=1,jjm
    DO i=1,iip1
      rot( i,j,l ) =  vcov(i+1,j,l)-vcov(i,j,l) + ucov(i,j+1,l)-ucov(i,j,l)
    ENDDO
  ENDDO
ENDDO
```

- Cas linéarisé 1D : `real :: ucov(ip1jmp1,llm), vcov(ip1jm,llm),rot(ip1jm,llm)`

```
DO l=1,llm
  DO ij=1,ip1jm
    rot( ij,l ) =  vcov(ij+1,l)-vcov(ij,l) + ucov(ij+iip1,l)-ucov(ij,l)
  ENDDO
ENDDO
```

MPI : Mise en œuvre (suite)

- En parallèle :
  - Cas 2D

```
DO l=1,llm
  DO j=jj_begin,jj_end
    DO i=1,iip1
      rot( i,j,l ) =  vcov(i+1,j,l)-vcov(i,j,l) + ucov(i,j+1,l)-ucov(i,j,l)
    ENDDO
  ENDDO
ENDDO
```

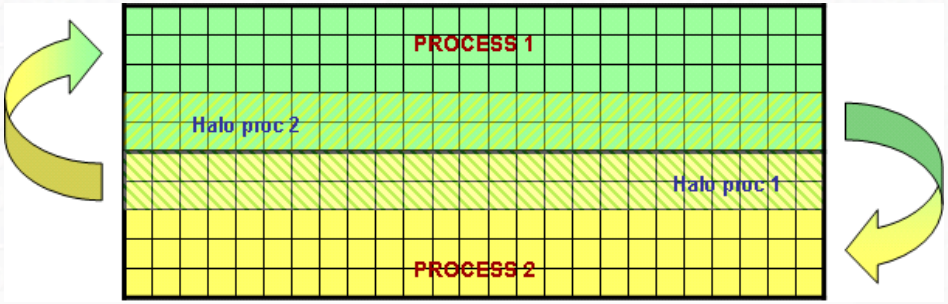
  - Cas linéarisé

```
DO l=1,llm
  DO ij=ij_begin,ij_end
    rot( ij,l ) =  vcov(ij+1,l) - vcov(ij,l) + ucov(ij+iip1,l) - ucov(ij,l)
  ENDDO
ENDDO
```

- Cas plus complexe lorsque :
  - Traitement particulier aux pôles nord et sud.
    - utilisation des booléens : pole_nord et pole_sud.
  - Les résultats de calcul sur le domaine demande des valeurs de champs en dehors du domaine
    - Nécessite de jouer sur les indices de début et de fin de boucle.
    - Nécessite le transfert de halos aux frontières du domaine.

MPI : échange de halos de données

- Nécessité d'échanger des halos de données entre processus voisins.



  - caldyn : échange de halos à chaque pas de temps sur : **`ucov, vcov, teta,ps, masse, pk, pks, pkf, phis`** et **`q`**.
  - Advection : échange nécessaire avant chaque calcul de l'advection en latitude (**`vly`**).
  - Dissipation : de nombreux échanges de halos en interne dus au calcul du Laplacien itératif (halo de 2 sur chaque application du Laplacien).
- Les appels à la bibliothèque MPI pour les échanges de halos sont encapsulés dans les routines du module : **`mod_halo`**.
  - On ne manipule pas directement les appels MPI.
  - Simplifie la procédure d'appel car le halo complet d'un champ n'est pas continu en mémoire (à cause des niveaux verticaux).
  - Elles permettent de cumuler les échanges de halos de plusieurs champs en un seul appel MPI.
  - Elles permettent de facilement recouvrir les communications par du calcul.

MPI : échange de halos de données (suite)

Synopsis

- Définir le halo d'un champ au sein d'une requête. Uniquement descriptif.

```
subroutine Register_Hallo(Field,ij,ll,RUp,Rdown,SUp,SDown,a_request)
```

`integer :: ij` : taille du champ sur le domaine horizontal

`integer :: ll` : nb de niveaux verticaux du champ

`real :: field(ij,ll)` : champ à échanger

`integer :: RUp` : taille du halo reçu par le proc. voisin du haut (`mpi_rank-1`)

`integer :: Rdown` : taille du halo reçu par le proc. voisin du bas (`mpi_rank+1`)

`integer :: RUp` : taille du halo reçu par le voisin du haut

`integer :: Rdown` : taille du halo reçu par le voisin du bas

`type(request) :: a_request` : objet contenant les informations sur la requête en cours.

- Envoyer la requête : bufférise les données puis effectue l'appel MPI.

```
subroutine SendRequest(a_Request)
  type(request) :: a_request
```

- Attendre la complétion de la requête : les données ont bien été envoyées, les halos ont bien été reçus.

```
subroutine WaitRequest(a_Request)
  type(request) :: a_request
```

MPI : échange de halos de données (suite)

- Exemple (presque) tiré du code : les échanges de halos pour caldyn :

```
! On defini les hallos dans la requète : Caldyn_Request
  CALL Register_Hallo(ucov,ip1jmp1,llm,1,1,1,1, Caldyn_Request)
  CALL Register_Hallo(vcov,ip1jm,llm,1,1,1,1, Caldyn_Request)
  CALL Register_Hallo(teta,ip1jmp1,llm,1,1,1,1, Caldyn_Request)
  CALL Register_Hallo(ps,ip1jmp1,1,1,2,2,1, Caldyn_Request)
  CALL Register_Hallo(pkf,ip1jmp1,llm,1,1,1,1, Caldyn_Request)
  CALL Register_Hallo(pk,ip1jmp1,llm,1,1,1,1, Caldyn_Request)
  CALL Register_Hallo(pks,ip1jmp1,1,1,1,1,1, Caldyn_Request)
  CALL Register_Hallo(p,ip1jmp1,llmp1,1,1,1,1, Caldyn_Request)

! Fin de la définition des halos : on envoie la requête.
  CALL SendRequest(Caldyn_Request)

! Je ferai bien un petit calcul en attendant que les données soient là…
  CALL Faire_un petit_calcul

! Les données sont arrivées, on met à jour les champs.
  CALL WaitRequest(Caldyn_Request)
! Les halos sont à jours, on peut effectuer le pas de temps.
```

L'équilibrage de charge

- Le filtre est appliqué sur 1/3 de la surface du globe, 1/6 près de chaque pôle.
  - Le filtre est très coûteux en temps de calcul (jusqu'à 50% du coût total) pour les hautes résolutions.
  - Les processus près des pôles travaillent 10 fois moins vite que ceux près de l'équateur.
  - Les plus rapides attendent les plus lents lors des synchronisations (échange de halos).
  - Facteur 3 en pertes d'efficacité.
- Solution : modifier la taille de chaque domaine de façon à ce chaque processus ait globalement la même charge de calcul.
  - Modifier les valeurs jj_nb_para de manière adéquate, puis calculer les autres paramètres du domaine en fonction.
- Oui mais… Les différentes routines utilisent le filtre de manière plus ou moins intensive :
  - Caldyn : assez intensément
  - Advection : pas du tout
  - Dissipation : très intensément
  - Calfis+physique : pas du tout
- Or toutes ces routines nécessitent des synchronisations !
  - Impossibilité de définir efficacement domaine unique.

L'équilibrage de charge (suite)

- Solution : définir un domaine de calcul optimum pour chaque classe de routine.
- Un domaine est complètement défini par le nombre de bandes de latitude par processus <==> `jj_nb_para(0:mpi_size-1)` .
- On défini plusieurs distributions sous forme de tableaux (module **`bands`** fichier **`bands.F90`**) :
  - **`jj_nb_caldyn`** : pour caldyn
  - **`jj_nb_vanleer`** : pour l'advection
  - **`jj_nb_physiq`** : pour l'appel de la physique
  - **`jj_nb_dissip`** : pour la dissipation
- Une distribution équilibrée dépend de la machine, de la résolution et du nombre de processus utilisés.
- Les valeurs des distributions sont lues à partir du fichier : **`Bands_resolution_nbprc.dat`**.
  - ex : résolution 96x72x19 sur 8 proc => **`Bands_96x72x19_8prc.dat`**
- Pour passer d'une distribution à l'autre et redéfinir les paramètres du domaine :

```
subroutine set_distrib(new_distrib)
  integer :: new_distrib(0:mpi_size-1)
```

L'équilibrage de charge (suite)

- Nécessite le transfert entre processus des valeurs disjointes des champs des 2 distributions :

```
subroutine RegisterSwapField(field, ij, ll, new_distrib,a_request)
  integer :: ij
  integer :: ll
  real :: field(ij,ll)
  integer :: new_distrib(0:mpi_size-1)
  type(request) :: a_request
```

- Même utilisation que pour les halos.
- Exemple passage de la distribution caldyn à dissip avec tranfert des champs ucov et vcov.

```
! On est dans la distribution caldyn
! On enregistre les champs que l'on souhaite dans la nouvelle distrib
CALL RegisterSwapField(ucov,ip1jmp1,llm,jj_nb_dissip,request_dissip)
CALL RegisterSwapField(vcov,ip1jm,llm,jj_nb_dissip,request_dissip)
! On envoie la requète
CALL SendRequest(request_dissip)
! On attend que les champs soient reçus
CALL WaitRequest(request_dissip)
! On passe dans la nouvelle distribution)
CALL set_distrib(jj_nb_dissip)
! On peut effectuer les calculs sur le nouveau domaine
```

L'équilibrage de charge (suite)

- Routine permettant de combiner la distribution des champs avec le transfert d'un halo :

  subroutine RegisterSwapFieldHalo

- L'ajustement automatique.
  - La détermination de l'équilibrage de charge optimum est vite fastidieux.
    - Procédure automatique d'ajustement.
      - activée avec le paramètre `adjust=y` dans le `run.def`

    1. On démarre avec une distribution uniforme (même nb de bandes par processus) ou du fichier bands_res_nproc.dat s'il existe.
    2. On mesure avec un timer le temps passé dans une distribution pour chacun des processus.
    3. On enlève une bande à la distribution du processus le plus lents et on en ajoute une au processus le plus rapide.
    4. On se place dans la nouvelle distribution.
    5. On réitère la mesure de temps pour la nouvelle distribution, jusqu'à ce que l'échange de bandes entre processus n'amène plus d'amélioration.
    6. On écrit le fichier de Bands pour la prochaine exécution.

  - En général, pour un bon ajustement, il faut entre 1000 et 2000 pas de temps => ~ 5 jours.

La gestion des IO dans la dynamique

- L'initialisation
  - Tous les processus effectuent la lecture des paramètres (**getin**) à partir des fichiers ***.def**.
  - Tous les processus lisent à l'identique le fichier de démarrage (start.nc).
  - A l'entrée de **leapfrog**, l'ensemble des processus ont leurs champs initialisés sur le domaine global.
- L'écriture du fichier de redémarrage (restart.nc).
  - Les champs à écrire sont rassemblés sur le proc. de rang 0.
  - Seul le processus de rang 0 écrit le fichier de restart.
- Le guidage.
  - Seul le proc. de rang 0 lit et interpole les champs de guidage.
  - Les champs sont ensuite distribués aux autres processus.
- Les fichiers histoires : dyn_hist.nc, dyn_histv.nc, dyn_histave.nc
  - Chaque processus écrit son fichier local (sous la distribution caldyn), indexé par le rang du processus.
  - Les fichiers sont ensuite reconstruits à l'aide de l'utilitaire rebuild (IOIPSL) (voir la reconstruction des fichiers histoire de la physique).
- Les fichiers de bilan zonaux : dyn_zon.nc
  - Chaque processus écrit son fichier local qui doit être reconstruit.

La gestion des IO dans la dynamique

- Les fichiers de stockage des flux :
  - **defstock.nc** : écrit par le proc. maître.
  - **fluxstoke.nc** et **fluxstokev.nc** : chaque proc. écrit son fichier.

- Quelques routines utiles dans la gestion des IOs en parallèle:

  - Rassembler des champs sur le processus maître : **Gather_Field**

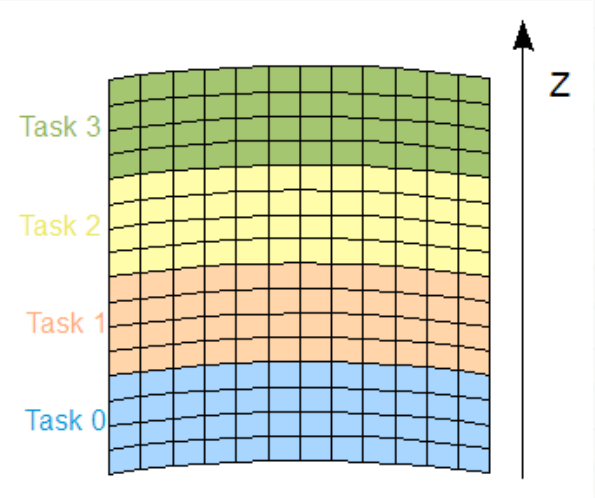
  Ex : **CALL Gather_Field(ucov,ip1jmp1,llm,0)**

  - Rassembler les champs sur tous les processus : AllGather_Field

  - Dupliquer un champ sur tous les processus à partir du processus maître : **Broadcast_Field**

  Ex: **CALL Broadcast_field(vcov,ip1jm,llm,0)**

- Ajout d'un niveau de parallélisation sur la verticale.
  - On n'entre pas en concurrence avec la parallélisation MPI.
  - Comme on parallélise sur les boucles les plus externes, on n'entre pas en conflit avec la vectorisation.
  - La parallélisation en OpenMP sur les niveaux verticaux est beaucoup plus simple qu'en MPI.



- L'initialisation est faite en séquentielle.
- Ouverture de la section parallèle juste avant l'appel à leapfrog.
  - Le nombre de tâches lancées est fixé à l'aide de la variable d'environnement OMP_NUM_THREADS.
  - Une seule section parallèle pour tout le code.

La parallélisation OpenMP (suite)

- Principe de la parallélisation : seules les boucles sur la verticale sont parallélisées par l'ajout de directives.
- Exemple :

```
!$OMP DO SCHEDULE(STATIC,omp_chunk)
  DO l=1,llm
    DO ij=ij_begin,ij_end
      rot( ij,l ) =  vcov(ij+1,l) - vcov(ij,l) + ucov(ij+iip1,l) - ucov(ij,l)
    ENDDO
  ENDDO
!$OMP END DO NOWAIT
```

- La clause **SCHEDULE(STATIC,omp_chunk)**
  - Le découpage des boucles par tâche est déterminée à la compilation
  - Se fait par paquet de taille fixé par la valeur omp_chunk, à la manière round-robin.
  - La valeur omp_chunk est définie dans le fichier run.def.
    - Choisir de préférence la valeur : (llm+1)/nb_task.
  - Exemple : pour 4 tâches OpenMP et **llm=19**, répartition des itérations :
    - tâche 0 : itérations 1-5 (5)
    - tâche 1 : itérations 6-10 (5)
    - tâche 2 : itérations 11-15 (5)
    - tâche 3 : itérations 16-19 (4)

La parallélisation OpenMP (suite)

- La clause **NOWAIT** ➔ le début des problèmes…
  - Ne synchronise pas les tâches à la fin de la boucle.
    - Gain important sur les performances.
  - Attention !! La mémoire n'est plus cohérente entre les différentes tâches car les valeurs modifiées dans la boucle sont encore dans les registres ou en cache !
  - Implique que chaque tâches doit toujours travailler sur le même domaine d'itération de façon à garder la cohérence localement.
  - Imposer une synchronisation et rétablir la cohérence mémoire : la directive **barrier** :

```
!$OMP DO BARRIER
```

    - Très coûteux, à faire le moins possible.
- Statut des variables :
  - Les principaux champs sur la grille (ucov, vcov, etc…) sont déclarés avec l'attribut **SAVE** et sont donc partagés par tous les threads.
    - Gain en espace mémoire.
    - Permet d'assurer la cohérence mémoire pour ces champs.
  - L'ensemble des variables de contrôle sont déclarées automatiquement sur la pile (stack) et sont donc privées à chaque threads.
- L'ensemble du code est exécuté à l'identique par toutes les tâches, seules les itérations sont distribuées.

La parallélisation OpenMP (suite)

- En résumé :
  - Les calculs sur les variables privées (non partagées) sont effectués par tous les threads, elles ont donc la même valeur sur chaque tâches.
  - Les calculs sur les champs partagés (**SAVE**) sont distribués, mais après une synchronisation (barrière), les champs lus par chacune des tâches sont identiques.
- Lorsqu'une tâche a besoin d'une valeur de champ calculé par une autre tâche
  - **!$OMP BARRIER** : rétabli la cohérence.
- Utilisation de la directive : **!$OMP MASTER/!$OMP END MASTER**
  - Lorsque seule la tâche maître doit effectuer une opération.
  - Exemple : la gestion des Entrées/Sorties.
  - Synchronisation avant de façon a obtenir un accès cohérent à la mémoire.
- Cohabitation MPI/OpenMP
  - Les bibliothèques MPI « thread-safe » sont quasi-inexistante.
  - En général les bibliothèques MPI permettent le multi-threading à condition qu'une seule tâche ne fasse appel à la bibliothèque à la fois.
    - Utilisation de sections critiques

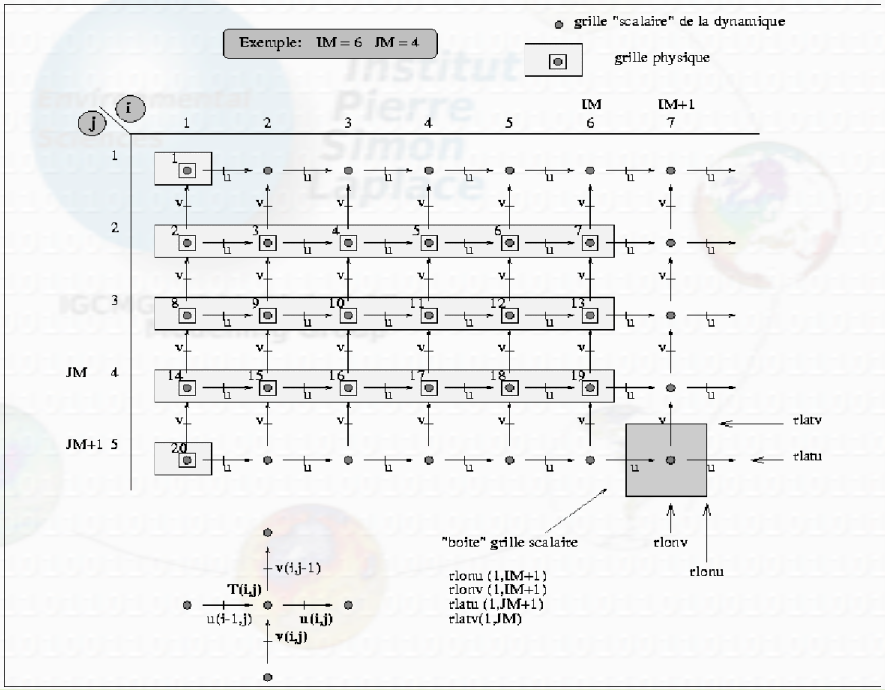
```
!$OMP CRITICAL(section_mpi)
  CALL MPI_ISSEND(…)
!$OMP END CRITICAL
```

  - Pour les échanges de halos, chaque tâche envoie sa portion de halo à la tâche correspondante du processus MPI voisin.

La parallélisation de la physique

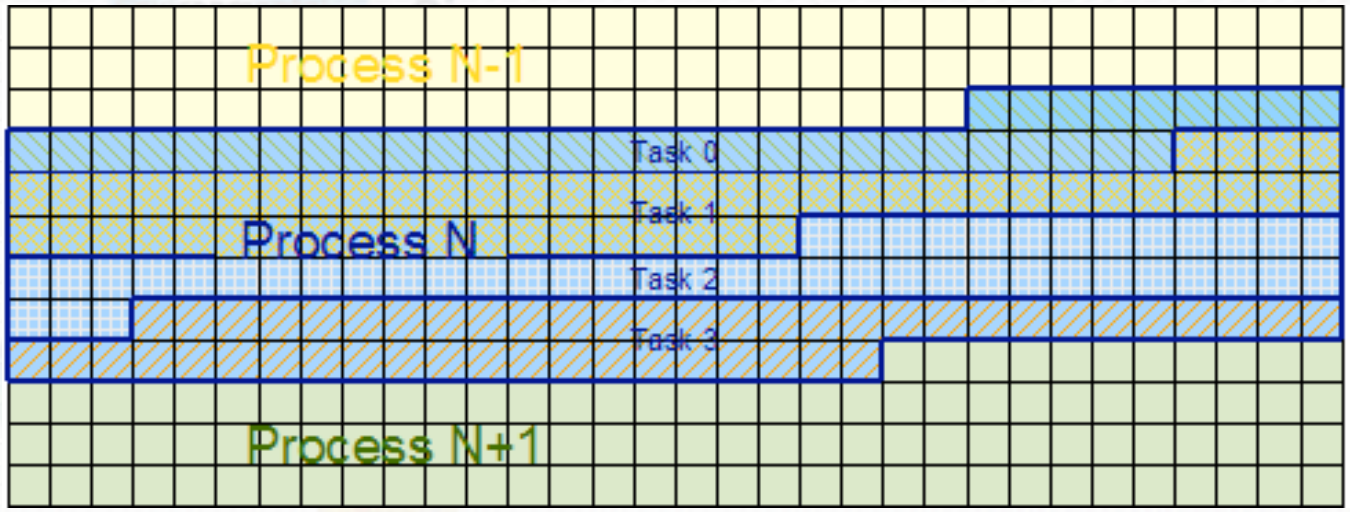
La partie physique de LMDZ

- La partie physique est chargée du calcul des phénomènes atmosphériques interagissant au sein d'une même colonne d'atmosphère : rayonnement, convection, interaction des couches limites,etc. …
- Les différentes colonnes d'atmosphère n'interagissent pas entre-elles.
- La stratégie de parallélisation consiste donc à distribuer le calcul des colonne d'atmosphère sur les différents processeurs.
- Grille de la physique : **`klon`** point géographiques sur **`klev`** niveaux verticaux.
  - 1er point => pôle nord, dernier point (**`klon`**) => pôle sud.



Découpage du domaine

- Les points du domaine global sont distribués aux différents processus MPI.
- Les points d'un domaine MPI sont distribués à chacun des threads OpenMP tournant à l'intérieur du processus :
  - Le domaine global : `klon_glo` x `klev` mailles.
  - Le domaine MPI : `klon_mpi x klev` maille : Σ `klon_mpi = klon_glo`
  - Le domaine OpenMP : `klon_omp x klev` mailles : Σ `klon_omp = klon_mpi`



Définition des paramètres du domaine

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` : nb points sur le domaine mpi local.
- `klon_mpi_begin` : indice de départ du domaine sur la grille 1D globale.
- `klon_mpi_end` : indice de départ 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 bande de latitude == `jj_end-jj_begin+1`
- `is_north_pole` : `.true.` si le processus possède le pôle nord.
- `is_south_pole` : `.true.` si le processus possède le pôle sud.
- `is_mpi_root` : `.true.` si processus MPI maître.
- `mpi_rank` : rang du processus MPI.
- `mpi_size` : nombre de processus MPI.

Définition des paramètres du domaine

- Grille OpenMP : sous-grille du domaine MPI
  - **klon_omp** : nombre de point 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. tâche maîtresse OpenMP.
  - **omp_size** : nb tâche OpenMp à l'intérieur du processus.
  - **omp_rank** : rang de la tâche OpenMP.

- La partie physique est activement développée en permanence.
- L'objectif de la parallélisation est d'être (relativement) transparente pour les développeurs.
- La taille du domaine local **klon** est un alias de **klon_omp**.
  - Inclusion du module : **dimphy**
  - Les boucles sur le domaine horizontal vont toujours de 1 à **klon**.
- Ce qui change : (presque) rien dans la majorité du code (95%), si :
  - pas d'interaction entre les colonnes d'atmosphère.
  - pas de moyenne ou de calcul zonal.
  - 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**.

```
real, save :: save_var
!$OMP THREADPRIVATE(save_var)
```

Le transfert des données entre tâches/processus

- Nécessité de transférer des données entre processus et/ou entre tâches pour les Entrées/Sorties ou les interfaces vers les autres codes.

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

- Inclusion du module : **`mod_phys_lmdz_transfert_para`**

- Les interfaces de transfert acceptent indifféremment les principaux types de base :
  - **`REAL`**
  - **`INTEGER`**
  - **`LOGICAL`**
  - **`CHARACTER`** : juste le broadcast

- Les interfaces acceptent indifféremment les champs jusqu'à 4 dimensions.

Le transfert des données (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 : `nb_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)
```

Les transfert de grille 1D/2D

- Routines définies pour tous types et jusqu'à 4 dimensions.

- Grille globale : module : **mod_grid_phy_lmdz.**
  - 1D -> 2D

```
CALL grid1Dto2D_glo(field1d_glo,field2d_glo)
```

  - 2D -> 1D

```
CALL grid2Dto1D_glo(field2d_glo,field1D_glo)
```

- Pour la grille MPI : **mod_phys_lmdz_mpi_transfert.**
  - Le champs local sur la grille 2D doit avoir les dimensions **nbp_lon x jj_nb** et **klon_mpi** sur la grille 1D.
  - 1D -> 2D

```
CALL grid1Dto2D_mpi(field1D_mpi,field2D_mpi)
```

  - 2D -> 1D

```
CALL grid2Dto1D_mpi(field2D_mpi,field1D_mpi)
```

La gestion des entrées/sorties

- Cas de figure qui demande des transferts tâche/processus.

- La lecture du fichier de paramètres de la physique : **physiq.def**
  - Effectué dans le fichier **conf_phys.F90**
  - La tâche maître lit la valeur puis la duplique (bcast) sur les autres tâches.
  - Exemple :

```
logical :: ok_strato
IF (is_mpi_root .AND. is_omp_root) CALL getin('ok_strato',ok_strato)
CALL bcast(ok_strato)
```

- La lecture des champs du fichier de démarrage : **startphy.nc**
  - Effectué dans **phyetat0**.
  - La tâche maître 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**.
  - Exemple : lecture des fractions de point de terre.

```
CALL get_field('FTER',pctsrf(:,ister),found)
IF (.NOT. Found) PRINT *, 'Le champ <FTER> n'existe pas'
```

La gestion des entrées/sorties (suite)

- L'écriture du fichier de redémarrage : **restartphy.nc**
  - Effectué dans **phyredem.F90**
  - Les données sont rassemblées sur la grille globale de la tâche maîtresse à l'aide d'un gather puis sont ensuite écrites.
  - Encapsulé dans la routine **put_field**
  - Exemple :

```
CALL put_field('FTER','fraction de continent',pctsrf(:,is_ter))
```

- L'écriture des fichiers histoires d'IOIPSL.
  - Chaque processus MPI écrit la partie de son domaine.
  - Le domaine du fichier IOIPSL est défini au moment de l'appel à **histbeg**
    - encapsulé dans la routine **histbeg_phy**
  - Les données sont rassemblées sur la tâche de rang 0 de chaque processus

```
CALL gather_omp(field,field_mpi)
```

  - Chaque processus MPI appelle la routine **histwrite** d'IOIPSL.
  - Les appels à l'écriture vers IOIPSL sont encapsulés dans la fonction **histwrite_phy.**
  - Exemple : écriture de la température au sol mensuelle.

```
CALL histwrite_phy(nid_mth,"tsol",itau_w,zxtsol)
```

  - Recontruction : **rebuild -o histmth.nc histmth_000[0-N].nc**

La gestion des entrées/sorties (suite)

- De manière générale : comment lire un champ dans un fichier netcdf (2D).
  - Toutes les ouvertures de fichiers et les accès en lecture doivent être faite par la tâche maîtresse.
    - Utiliser les booléans :

```
IF (is_mpi_root .AND. is_omp_root) THEN
```

  - Le champ doit être lu en global 2D par la tâche maîtresse.

```
REAL :: Field2D_glo(nbp_lon,nbp_lat,nbp_lev)
```

  - Le champ est transféré en 1D aux autres tâches/processus.

```
REAL :: Field1D(klon,klev)
CALL scatter2D(Field2D_glo,field1D)
```

  - En résumé :

```
IF (is_mpi_root .AND. is_omp_root) THEN
  CALL Ouverture_fichier_netcdf)
  CALL lecture_variable(Field2d_glo)
ENDIF
CALL scatter2D(field2d_glo,field1d)
```

- Pour l'écriture : c'est l'inverse…
  - `CALL gather2D(field1d,Field2d_glo)`

L'équilibrage de charge

- Si le nombre de point par tâche est distribué de façon uniforme, l'équilibrage de charge n'est pas optimum sur la physique.
  - Certains calculs ne sont pas effectué de la même manière si l'on se trouve près des pôles ou près de l'équateur.

- Calcul de l'équilibrage optimum :
  - Lors d'un run précédent, on mesure le temps passé dans la physique par chaque processus.
  - Par une règle de 3, on en déduit la nouvelle répartition optimum.
  - Cette nouvelle répartition est stockée dans le fichier bands_res_nbproc.dat

- Lors de l'initialisation :
  - Si le fichier de Bands existe, on en utilise les valeurs pour fixer le nombre de mailles par processus MPI (**`klon_mpi`**)
  - Sinon, le nombre de mailles par processus est fixée de façon uniforme : **`klon_mpi=klon_glo/mpi_size.`**
  - Le nombre de mailles par tâche OpenMP est toujours fixé uniformément au sein d'un même processus MPI : **`klon == klon_omp = klon_mpi/omp_size`**

Les interfaces vers les autres modèles

L'interface dynamique/physique

- Le passage des champs de la dynamique vers la physique est effectué dans la routine **calfis**.

- La distribution MPI de la dynamique est alors : **jj_nb_physique**.

- Pour transformer un champs, on utilise le module **mod_interface_dyn_phys**
  - **integer :: index_i(klon_mpi)** : correspondance indice de longitude (global) de la dynamique en fonction de l'indice de maille de la physique (local).
  - **integer :: index_j(klon_mpi)** : correspondance longitude/indice de maille physique.

- On procède en 2 étapes :
  - Recopie du champ de la grille dynamique (globale) vers la grille MPI (klon_mpi). Le tableau intermédiaire doit avoir l'attribut **SAVE** pour être lu par toutes les tâches.
  - Recopie vers la grille locale OpenMP (**klon_omp**) à l'aide de l'indice de début de domaine **klon_omp_begin**.

- Exemple : passage du champ de pression de la dynamique vers la physique.

L'interface dynamique/physique

```
REAL :: pp(iim+1,jjm+1,llm+1)
REAL,SAVE :: zplev(klon_mpi,llm+1)
REAL      :: zplev_omp(klon_omp,llm+1)

! On transforme le champs de le dynamique vers la grille MPI locale
!$OMP DO SCHEDULE(STATIC,OMP_CHUNK)
  DO l = 1, llmp1
    DO ig0=1,klon_mpi
      i=index_i(ig0)
      j=index_j(ig0)
      zplev( ig0,l ) = pp(i,j,l)
    ENDDO
  ENDDO
!$OMP END DO NOWAIT

! Une synchronisation est ici nécessaire
!$OMP BARRIER

! Distribution du champ à chaque tâche
 DO l=1,llm+1
    DO i=1,klon_omp
      zplev_omp(i,l)=zplev(klon_omp_begin-1+i,l)
   ENDDO
 ENDDO

! On peut appeler la physique
CALL physiq(…,zplev_omp,…)
```

L'interface LMDZ (partie physique) / INCA

- INCA est sur la même grille que LMDZ.
- INCA est parallélisé MPI/OpenMP suivant les mêmes principes que la physique de LMDZ.
- Aucune intervention n'est nécessaire pour passer de la physique vers INCA et inversement.

L'interface LMDZ/OASIS/NEMO

- OASIS est conçu pour transférer des champs entre des codes parallèle MPI.

- L'OpenMP n'est pas géré, donc seule la tâche de rang 0 participe au transfert.

- Etapes lors de l'échange d'un champ vers OASIS/NEMO
  - Le champ à échanger est décompressé : **`knon -> klon`**
  - Le champ est ensuite transférer sur la tâche de rang 0 :
    - **`CALL gather_omp(field,field_mpi)`**
  - Le champ est transformé en un champ 2D sur la grille MPI locale
    - **`CALL grid1Dto2D(field_mpi,field2D_mpi)`**
  - Le champ est transmit à OASIS par la tâche de rang 0.

- Pour un champ provenant de NEMO, on effectue les étapes en sens inverse.

L'interface LMDZ/ORCHIDEE

- ORCHIDEE et LMDZ travaillent sur la même grille.
- Mais ORCHIDEE travaille uniquement sur les points de terre : grille compressée : **`ktindex(knon)`** .
- ORCHIDEE est parallèle MPI/OpenMP suivant les mêmes principes que la physique de LMDZ.
- Passage implicite entre les deux codes : module **`surf_land_orchidee`**.
- Difficulté lorsque **`knon = 0`** : pas de point de terre sur le domaine : on ne peut pas rentrer dans ORCHIDEE.

- Si sur toutes les tâches d'un processus **`knon = 0`** : pas de point de terre sur le domaine MPI.
  - Création du communicateur MPI d'ORCHIDEE : COMM_ORCH, qui contient un (ou plusieurs) processus en moins.
  - Seul les processus ayant des points de terre rentrent dans ORCHIDEE.
  - Les processus peuvent communiquer entre eux avec le nouveau communicateur transmit à ORCHIDEE.
  - Les processus sans point de terre continus le calcul dans LMDZ et attendent à la prochaine synchronisation MPI.

- Si sur une tâche : **knon = 0**
  - La tâche ne doit pas rentrer dans orchidée.
  - Elle doit attendre que les autres tâches sortent d'ORCHIDEE pour pouvoir continuer.
  - Une barrière `!$OMP BARRIER` est inefficace car ces synchronisations sont également utilisées dans ORCHIDEE.
    - Pas de notion de communicateur en OpenMP.
  - Implémentation d'une routine permettant de synchroniser les tâches : **mod_synchro_omp : subroutine synchro_omp**.
  - Utilise un mécanisme de mémoire partagée pour synchroniser les tâches.

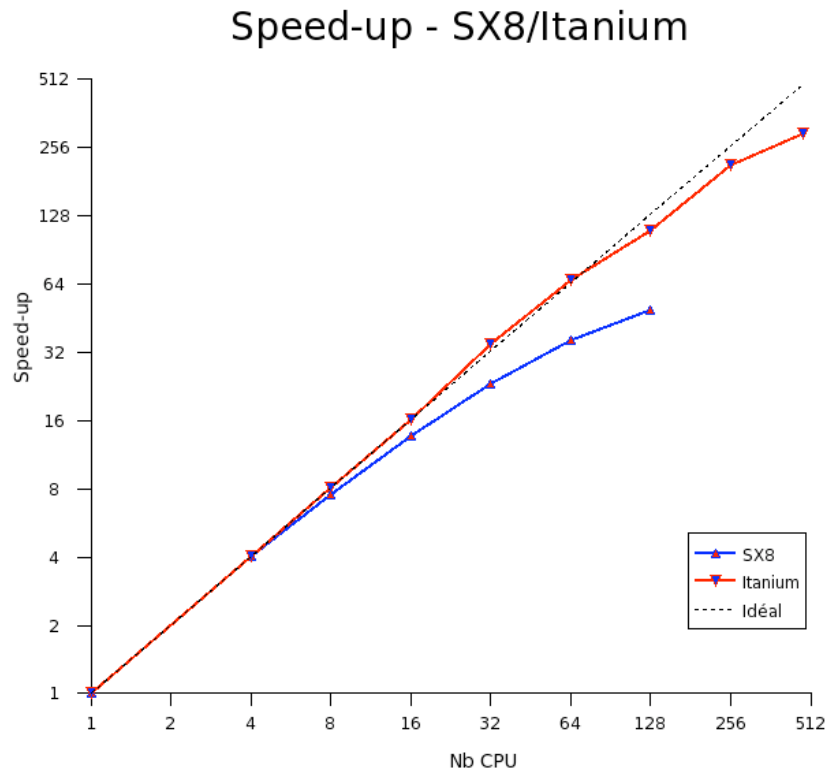
Quelques résultats…

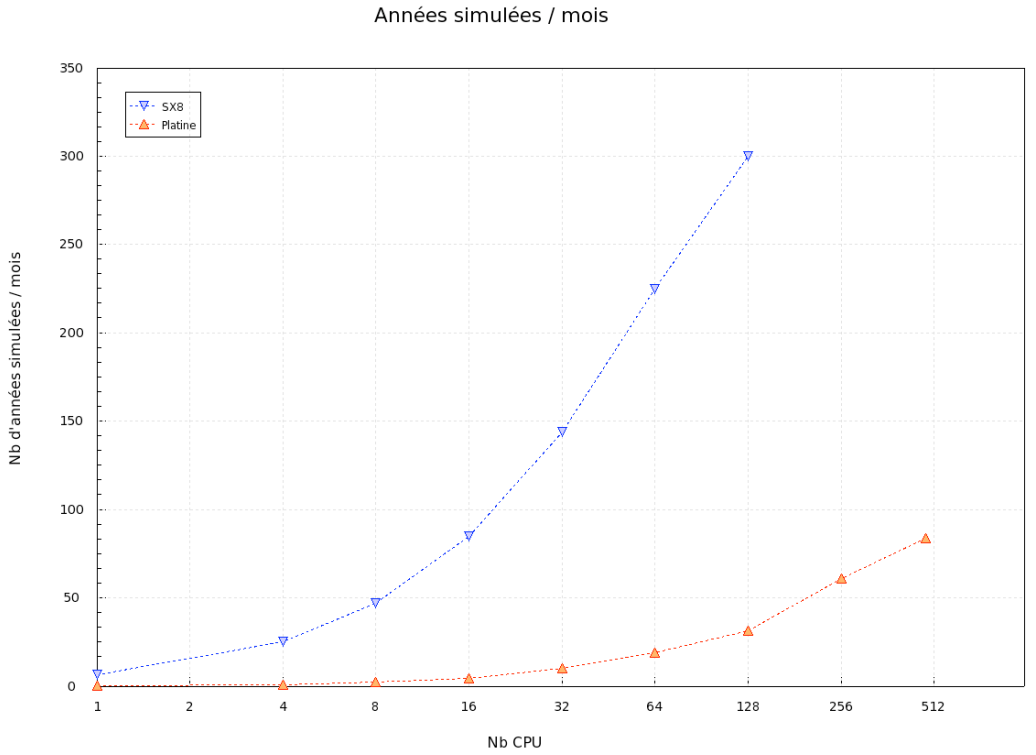
Quelques résultats

- Résolution 360x180X55
  - 10 ans de run version strato effectués au Earth Simulateur sur 128 Proc.

Nb processus	Platine		ES	
	Temps	Speed-up	Temps	Speed-up
1	(25056 s)	1	(2280 s)	1
2	(12528 s)	2	(1140 s)	2
4	(6264 s)	4	570 s	4
8	(3132 s)	8	307 s	7.42
16	1566 s	16	169 s	13.5
32	727 s	34.5	99 s	23
64	381 s	65.8	64 s	35.6
128	229 s	109.4	47 s	48.6
256	118 s	212.3		
480	86 s	291.3		

Quelques résultats (suite)





Quelques résultats (suite)

- Vargas : résolution 96x71x19

Nb CPUs	Nb process MPI	Nb tâche OMP/process	temps (s)	accélération
1	1	1	14640	1
2	2	1	7500	1,95
2	1	2	6930	2,11
4	1	4	3765	3,88
4	2	2	3585	4,08
4	4	1	3885	3,76
8	2	4	1956	7,48
8	4	2	1968	7,43
8	8	1	2250	6,51
16	4	4	1068	13,71
16	8	2	1128	12,98
16	16	1	1269	11,53
24	24	1	816	17,94
32	8	4	633	23,12
32	16	2	636	23
48	24	2	445	32,9
64	16	4	370	39,57
96	24	4	274	53,43