La concurrence en Fortran moderne

de do concurrent aux coarray

Vincent LAFAGE

IJCLab, Laboratoire de Physique des 2 Infinis Irène Joliot-Curie
CNRS/IN2P3 & Université Paris-Saclay, Orsay, France

Journée Fortran 2026



Plan

1. Remerciements et sources
2. Quelques réflexions sur la concurrence
3. Les syntaxes de la concurrence en Fortran
4. De MPI aux coarray
5. Les collectives
6. Pièges
7. `teams`
8. Limites



Plan

1. Remerciements et sources
2. Quelques réflexions sur la concurrence
3. Les syntaxes de la concurrence en Fortran
4. De MPI aux coarray
5. Les collectives
6. Pièges
7. teams
8. Limites



Remerciements et sources

- Les programmes MPI de cette présentation proviennent de la formation « MPI » de l'**IDRIS** (CNRS) :
  `http://www.idris.fr/docs/formations/hpc/formation-mpi`
  `https://raw.githubusercontent.com/idriscnrs/formations/refs/heads/main/mpi/idrismpif.pdf`
  Un grand merci à l'IDRIS et à ses formateurs pour ce support de référence.
- Ils ont été retouchés (`use mpi_f08, only:...`, `implicit none (type, external)`) ; certains ont été traduits en coarray.
- Tous les exemples coarray ont été compilés et exécutés avec `gfortran` 13 et OpenCoarrays 2.10 (`caf`, `cafrun`).
- Idée directrice : MPI = envoyer des messages ; coarray = lire et écrire directement chez les autres images.

CC BY SA



Plan

1 Remerciements et sources

2 Quelques réflexions sur la concurrence

3 Les syntaxes de la concurrence en Fortran

4 De MPI aux coarray

5 Les collectives

6 Pièges

7 teams

8 Limites



Séquentialité

PROGRAMMATION IMPÉRATIVE = programmer une séquence d'instructions/sous-tâches pour le processeur
programme comme une liste de courses ordonnée, comme une recette ordonnée
LA SÉQUENTIALITÉ est essentielle à la programmation

CC BY SA



Concurrence

Avec un seul processeur, les **tâches** seront exécutées les unes après les autres.
Souvent cet ordre est obligatoire : permuter les tâches changerait le résultat
...parfois cet ordre est contingent : permuter les tâches ne changerait pas le résultat

Si l'on peut identifier toutes ces tâches permutables,

- on pourrait les exécuter **HORS SÉQUENCE**
- on pourrait les exécuter **CONCURREMMENT** sur plusieurs processeurs, ou unités d'exécution

(exhiber la concurrence dans un programme est un processus d'industrialisation).

CC BY SA



Tâche & Fil d'exécution (*Thread*)

Niveau logique : on veut identifier les **TÂCHES** et parmi elles, les tâches indépendantes de l'ordre.
Niveau physique : on veut assigner les tâches à des **FILS D'EXÉCUTION** (threads).
Le multitâche peut se produire sur un seul processeur :

- **partage de temps** de la ressource de traitement entre les fils
- **changement de contexte** entre les fils

Si l'on a un multiprocesseur, certains/chaque processeur peut se voir assigner un ou plusieurs fils

PROGRAMMATION PARALLÈLE (aussi appelée multiprocessus) = **PROGRAMMATION CONCURRENTE** sur un **MULTIPROCESSEUR** (aussi appelée multiprogrammation)

deux types de boucles :

— les itérations dépendent de la/des précédente(s)
⇒ ce que l'on appelle habituellement un processus itératif

— les itérations sont indépendantes des précédentes
⇒ plus une duplication (ou N-uplication) qu'une itération

⇒ embarrassingly parallel = concurrence la plus faible possible = aussi découplé que possible
⇒ delightfully parallel !
très courant en physique des particules : chaque événement est indépendant et peut être traité sur un processeur séparé / dans un processus séparé
⇒ **traitement DISTRIBUÉ**

CC BY SA



Pureté

Lorsque l'on applique la même fonction à une collection d'objets, la collection des résultats est indépendante de l'ordre d'application de la fonction.
Pour garantir que cela soit vrai, nous avons besoin de fonctions **PURES** :
⇒ des fonctions informatiques aussi proches que possible des fonctions mathématiques

- les valeurs de retour de la fonction sont identiques pour des arguments identiques (pas de variation avec des variables statiques locales, des variables non locales, des arguments de référence mutables ou des flux d'entrée.)
  c'est-à-dire que son évaluation repose sur un **ALGORITHME DÉTERMINISTE** : pour une entrée particulière, elle produira toujours la même sortie, la machine sous-jacente passant toujours par la même séquence d'états
  ⇒ les fonctions sont *référentiellement transparentes* (voir ci-dessous)
- l'application de la fonction n'a pas d'**EFFETS DE BORD** : pas de mutation de variables statiques locales, de variables non locales, d'arguments de référence mutables ou de flux d'entrée/sortie

Pur = déterministe + sans effets de bord

CC BY SA



Pureté paradigme fonctionnel pragmatique...

- les arguments d'entrée doivent être *immutables* : `C++ const`, `Fortran intent (in)`,...
- l'évaluation ne doit pas reposer sur des variables globales (mutables)
  (p.ex. en `Fortran`, elle ne doit pas reposer sur des variables `COMMON`, mais elle peut reposer sur des `parameters` de module ou des variables `protected`.
  En `C++`, on peut utiliser des globales `const / constexpr / static constexpr`)
- une fonction pure ne peut appeler que des fonctions pures
- pas d'exceptions

TRANSPARENCE RÉFÉRENTIELLE :
⇒ l'expression peut être remplacée par sa valeur correspondante (et vice-versa) sans changer le comportement du programme.
⇒ permet la **MÉMOÏSATION** :
technique d'optimisation utilisée principalement pour accélérer les programmes informatiques en stockant les résultats d'appels de fonctions coûteux et en renvoyant le résultat en cache
un type spécifique de **TABLE DE CONSULTATION (*LUT : LookUp Table*)** :
⇒ une collection / un tableau de résultats précalculés que l'on réutilise au lieu de recalculer.
Les tables de consultation sont généralement initialisées au démarrage, tandis que la mémoïsation les remplit à la volée.



Mélanger le paradigme fonctionnel (pureté) avec le paradigme orienté objet changera **fortement** votre style orienté objet

CC BY SA



Effets de bord

ce qui se passe lorsque la fonction n'est pas pure...

- **Entrée/Sortie** : afficher quelque chose se produit dans un ordre donné, stocker des données sur disque (peut être vu comme un objet global)
- **comportement lié au matériel** : dépend de l'interaction avec l'environnement, qui est une variable globale
- **dépendance au temps** : le temps est une variable globale
- **exceptions** : votre fonction ne renvoie pas une valeur du type attendu, probablement à cause d'un domaine de définition limité pour les arguments.
  Une fonction mathématique n'est pas seulement pure, elle vise aussi la *totalité* (expansion maximale du domaine de définition)
- la plupart des générateurs de nombres aléatoires reposent sur un état caché qui change à chaque appel.

$\Rightarrow$ à long terme, aucune fonction informatique ne peut jamais être appelée pure :
faire tourner un ordinateur nécessite de l'énergie et augmente l'entropie de l'Univers, qui est une variable globale
$\Rightarrow$ effet de bord...

CC BY SA



court avertissement

ATTENTION !!!
Les évaluations en virgule flottante dépendent généralement de l'ordre d'évaluation :
les opérations en virgule flottante ne sont PAS associatives, contrairement à l'opération correspondante sur les nombres réels :
$\forall(a,b,c),(a+b)+c=a+(b+c)$ **MAIS** $\exists(a,b,c),(a\oplus b)\oplus c\neq a\oplus(b\oplus c)$
$\Rightarrow$ une opération hors-ordre peut changer, même très légèrement, le résultat
Effets de bord subtils introduits par les langages, les compilateurs et les options d'optimisation...

- `C` se conforme strictement à votre ordre de calcul
- `Fortran`, *i.e.* FORmula TRANslator, essaie d'optimiser quelque peu votre calcul : mathématiquement équivalent, numériquement pas strictement équivalent

Certaines vérifications de pureté par le compilateur sont plutôt formelles (particulièrement sur les architectures hétérogènes)...

CC BY SA



Qu'est-ce qui pourrait mal tourner ?

RÉENTRANCE

une sous-routine est dite réentrante si

- plusieurs invocations peuvent s'exécuter en toute sécurité de manière concurrente sur plusieurs processeurs,
- ou sur un système monoprocesseur, où une procédure réentrante peut être interrompue au milieu de son exécution puis être appelée à nouveau en toute sécurité (« réentrée ») avant que ses invocations précédentes n'aient terminé leur exécution.
- Le code réentrant ne doit détenir aucune donnée non constante statique ou globale sans synchronisation.
- Le code réentrant ne doit pas se modifier lui-même sans synchronisation.
- Le code réentrant ne doit pas appeler de programmes ou routines non réentrants.

SÛRETÉ DES FILS (THREAD SAFETY)

Le code sûr pour les fils ne manipule les structures de données partagées que d'une manière qui garantit que tous les fils se comportent correctement et remplissent leurs spécifications de conception sans interaction involontaire.
(*pas de course de données (data race)*)

réentrant $\not\Rightarrow$ sûr pour les fils
sûr pour les fils $\not\Rightarrow$ réentrant

`https://en.wikipedia.org/wiki/Reentrancy_(computing)`

`https://stackoverflow.com/questions/856823/threadsafe-vs-re-entrant`

CC BY SA



Quand une tâche prend le dessus

Une **SECTION CRITIQUE** est une partie de code où des accès concurrents à des ressources partagées conduiraient à un comportement erroné.
$\Rightarrow$ nous devons protéger ces accès
Verrou / mutex (exclusion mutuelle), objet protégé
(instruction atomique)

Pendant une section critique, on perd tous les bénéfices du multiprocesseur.

!!!Attention !!! : interblocage (dead lock)

point de synchronisation, ou rendez-vous :

parfois une tâche doit attendre l'achèvement d'une autre

CC BY SA



Plan

1 Remerciements et sources

2 Quelques réflexions sur la concurrence

3 Les syntaxes de la concurrence en Fortran

4 De MPI aux coarray

5 Les collectives

6 Pièges

7 teams

8 Limites



Maîtriser les tableaux la chasse aux boucles est ouverte

- opérations sur tableaux : $+$, $-$, $*$ (HADAMARD), / (*element-wise*)
- applications de fonctions sur tableaux (mapping)
  - les fonctions intrinsèques
  - les fonctions `ELEMENTAL` : `PURE` + automap
- réduction de tableaux : `SUM`, `PRODUCT(ARRAY, DIM, MASK)`
- `DOT_PRODUCT(VECTOR1, VECTOR2)`
- `MATMUL`
- `RESHAPE`
- `WHERE (mask) … [ELSEWHERE (mask) …] [ELSEWHERE …] END WHERE`
- …
- `FORALL (variable = from:to{:step})`
- `DO CONCURRENT (variable = from:to{:step})`

CC BY SA



Maîtriser les tableaux la chasse aux boucles est ouverte

- **Expressions de tableaux**
  - Une façon concise d'effectuer des opérations élément par élément sur des tableaux.
  - Permet le parallélisme implicite et améliore la lisibilité.
  - Généralise les opérations scalaires à des tableaux entiers.
  - Fait disparaître les itérations.
- **Boucle** `Forall`
  - Une construction pour les affectations de tableaux avec itération explicite.
  - Offre plus de contrôle que les expressions de tableaux, avec un potentiel d'exécution parallèle.
  - Fait le pont entre les boucles scalaires et les opérations sur tableaux.
- **Boucle** `Do Concurrent`
  - Une construction de boucle moderne conçue pour le parallélisme.
  - Permet des itérations indépendantes, optimisées par les compilateurs pour le multi-threading ou l'exécution sur GPU.
  - La plus puissante, avec une flexibilité pour des calculs complexes.
  - Les instructions à l'intérieur du `DO CONCURRENT` sont exécutées dans l'ordre, dans chaque « itération », mais il n'y a aucune dépendance aux autres itérations. `FORALL` ne pouvait contenir que des affectations de tableaux, mais chaque affectation devait être terminée par toutes les itérations avant que la suivante ne s'exécute, créant effectivement un « attendre tout le monde » après chaque affectation. Oui, l'idée de `FORALL` était d'aider à la parallélisation, mais comme une grande partie de High-Performance Fortran, la variante obsolète dont `FORALL` est issu, elle n'était pas bien pensée et la parallélisation était bien plus difficile qu'il n'y paraissait.

CC BY SA



Maîtriser les tableaux la chasse aux boucles est ouverte

- **Expressions de tableaux**
  - `real, dimension(10) :: a, b, c, d`
  - `c = a * b + 2.0` (Opérations combinées)
  - `d = sqrt(c)` (Fonctions élémentales ⇒ pures)
  - `real, dimension(5,5) :: m ; m = m + transpose(m)` (Symétrisation de matrice)
- **Boucle** `Forall`
  - `forall (i=1:10) a(i,i) = 0.0` (Mettre les éléments diagonaux à zéro)
  - `integer, dimension(100) :: idx ; forall (i=1:100, idx(i) > 0) idx(i) = idx(i) - 1` (Décrément conditionnel)
- **Boucle** `Do Concurrent`

```
real, dimension(100) :: a, x
do concurrent (i=1:100)
    a(i) = sin(x(i)) * cos(x(i))
    if (a(i) < 0) a(i) = 0.0
end do
```

```
a = sin (x) * cos (x)
where (a < 0.0) a = 0.0
! ou, sans where :
a = max (sin (x) * cos (x), 0.0)
```

```
real, dimension(1000) :: c
total = 0.0
do concurrent (i=1:1000) reduce(+:total)
   total = total + exp(c(i))
end do
```

```
total = sum (exp (c))
```

```
real, dimension(10, 10) :: b
do concurrent (i=1:10, j=1:10) local(tmp)
   tmp = 0.0
   if (i /= j) tmp = 1.0 / real(i-j)
   b(i,j) = tmp
end do
```

```
integer :: ii(10,10), jj(10,10), k
ii = spread ([(k, k = 1, 10)], dim=2, ncopies=10)
jj = spread ([(k, k = 1, 10)], dim=1, ncopies=10)
where (ii /= jj)
   b = 1.0 / real (ii - jj)
elsewhere
   b = 0.0
```


CC BY SA



Maîtriser les tableaux la chasse aux boucles est ouverte

- **Expressions de tableaux**
  - `real, dimension(10) :: a, b, c, d`
  - `c = a * b + 2.0` (Opérations combinées)
  - `d = sqrt(c)` (Fonctions élémentales ⇒ pures)
  - `real, dimension(5,5) :: m ; m = m + transpose(m)` (Symétrisation de matrice)
- **Boucle** `Forall`
  - `forall (i=1:10) a(i,i) = 0.0` (Mettre les éléments diagonaux à zéro)
  - `integer, dimension(100) :: idx ; forall (i=1:100, idx(i) > 0) idx(i) = idx(i) - 1` (Décrément conditionnel)
- **Boucle** `Do Concurrent`

```
real, dimension(100) :: a, x
do concurrent (i=1:100)
    a(i) = sin(x(i)) * cos(x(i))
    if (a(i) < 0) a(i) = 0.0
end do
```

```
a = sin (x) * cos (x)
where (a < 0.0) a = 0.0
! ou, sans where :
a = max (sin (x) * cos (x), 0.0)
```

```
real, dimension(1000) :: c
total = 0.0
do concurrent (i=1:1000) reduce(+:total)
   total = total + exp(c(i))
end do
```

```
total = sum (exp (c))
```

```
real, dimension(10, 10) :: b
do concurrent (i=1:10, j=1:10) local(tmp)
   tmp = 0.0
   if (i /= j) tmp = 1.0 / real(i-j)
   b(i,j) = tmp
end do
```

```
integer :: ii(10,10), jj(10,10), k
ii = spread ([(k, k = 1, 10)], dim=2, ncopies=10)
jj = spread ([(k, k = 1, 10)], dim=1, ncopies=10)
where (ii /= jj)
   b = 1.0 / real (ii - jj)
elsewhere
   b = 0.0
```


CC BY SA



Vraie do concurrent

Ensemble de Mandelbrot

```
integer, parameter :: nx = 800, ny = 600, maxit = 1000
integer :: iter(nx, ny), ix, iy, k
complex :: c, z

do concurrent (ix = 1:nx, iy = 1:ny) local (c, z, k)
   c = cmplx (xmin + (ix - 1) * dx, ymin + (iy - 1) * dy)
   z = (0.0, 0.0)
   do k = 1, maxit
      if (abs (z) > 2.0) exit      ! sortie de la boucle interne : autorisée
      z = z * z + c
   end do
   iter(ix, iy) = k
end do
```

- Concurrence entre pixels, séquentialité à l'intérieur
- `local` prend tout son sens
- Seule la sortie de la construction `do concurrent` est interdite

Résolution de **systèmes tridiagonaux indépendants** (algorithme de Thomas)

```
do concurrent (j = 1:m)
   call thomas (a(:,j), b(:,j), c(:,j), d(:,j), x(:,j))   ! thomas est pure
end do
```


CC BY SA

Localité (Fortran 2018)

```
do concurrent (i = 1:n) &
     default (none) shared (x, y) local
      ↪  (tmp)
   tmp  = sin (x(i))
   y(i) = tmp * tmp
end do
```

Réduction (Fortran 2023)

```
total = 0.0
xmax  = -huge (xmax)
do concurrent (i = 1:n) reduce (+:total)
   total = total + y(i)
end do
do concurrent (i = 1:n) reduce (max:xmax)
   xmax = max (xmax, y(i))
end do
```

- Une variable écrite par plusieurs itérations est une course de données *(data race)* : il faut dire *qui la possède*.
- `local(v)` : une copie privée par itération, indéfinie à l'entrée et après la boucle.
- `local_init(v)` : idem, initialisée avec la valeur d'avant la boucle.
- `shared(v)`, `default(none)` : rendre la propriété **explicite**, le compilateur vérifie.
- `reduce(op:v)` : chaque itération contribue, le résultat est combiné avec la valeur d'avant.
  op : `+`, `*`, `max`, `min`, `iand`, `ior`, `ieor`, `.and.`, `.or.`…
- Attention : l'ordre de combinaison n'est pas spécifié, et l'addition flottante n'est pas associative (diapo 12) ⇒ le résultat peut varier.
- Support : vérifier compilateur + version

CC BY SA



Do Concurrent et OpenMP la même localité, deux vocabulaires

DO CONCURRENT	OpenMP	Remarque
`local (v)`	`private (v)`	copie non initialisée
`local_init (v)`	`firstprivate (v)`	copie initialisée
`shared (v)`	`shared (v)`	
`default (none)`	`default (none)`	même **discipline**
`reduce (op:v)`	`reduction (op:v)`	même sémantique, orthographe différente
`(i=…, j=…)`	`collapse (2)`	indices multiples
(rien)	`lastprivate (v)`	`local` est indéfinie après la boucle
(rien)	`schedule`, `num_threads`, `nowait`…	le compilateur décide

- Affirmer ou prescrire : `do concurrent` *affirme* que les itérations sont indépendantes, le compilateur choisit l'exécution (séquentielle, SIMD, fils, GPU). `!$omp parallel do` *prescrit* l'exécution en fils. Le plus proche en OpenMP est `!$omp loop` (5.0), qui est descriptif.
- Par itération ou par fil : `local` et `local_init` donnent une copie *par itération*, `private` et `firstprivate` une copie *par fil*. Avec `firstprivate`, la copie d'un fil peut garder la valeur de son itération précédente ; avec `local_init`, elle repart de la valeur d'avant la boucle.
- Écrire `default(none)` des deux côtés oblige à expliciter le même inventaire de variables : la traduction clause à clause devient presque mécanique. Restent à traiter la copie par itération ou par fil, les indices et les variables de bornes.

CC BY SA



Do Concurrent et OpenMP un même calcul

Fortran (do concurrent)

```
total = 0.0
do concurrent (i = 1:n, j = 1:m) &
     default (none) shared (x, y) &
     local (tmp) reduce (+:total)
   tmp   = x(i,j) * y(i,j)
   total = total + tmp
end do
```

OpenMP (parallel do)

```
total = 0.0
!$omp parallel do collapse(2) &
!$omp default(none) shared(x, y, n, m) &
!$omp private(tmp) reduction(+:total)
do j = 1, m
   do i = 1, n
      tmp   = x(i,j) * y(i,j)
      total = total + tmp
   end do
end do
!$omp end parallel do
```

- Mêmes variables, mêmes rôles : seuls changent le mot-clé et l'écriture de la boucle (un seul `do concurrent` contre deux `do` imbriqués).
- Les indices `i` et `j` sont implicitement locaux dans les deux cas.
- La somme flottante dépend de l'ordre de combinaison, dans les deux versions (diapo 12).

CC BY SA



Plan

1 Remerciements et sources

2 Quelques réflexions sur la concurrence

3 Les syntaxes de la concurrence en Fortran

4 De MPI aux coarray

5 Les collectives

6 Pièges

7 teams

8 Limites



Un modèle SPMD, deux vocabulaires

MPI

```
program qui_je_suis
  use mpi_f08, only: MPI_COMM_SIZE, MPI_COMM_WORLD, MPI_INIT,
  ↪  MPI_COMM_RANK, MPI_FINALIZE
  implicit none (type, external)
  integer :: nb_procs, rang

  call MPI_INIT ()

  call MPI_COMM_SIZE (MPI_COMM_WORLD, nb_procs)
  call MPI_COMM_RANK (MPI_COMM_WORLD, rang)

  print *, 'Je suis le processus ', rang, ' parmi ', nb_procs

  call MPI_FINALIZE ()
end program qui_je_suis
```

Coarray

```
program qui_je_suis
   implicit none (type, external)
   integer :: nb_images, this_image_num

   nb_images = num_images ()
   this_image_num = this_image ()

   print *, 'Je suis l'image ', this_image_num, ' parmi
   ↪  ', nb_images
end program qui_je_suis
```

- Ni initialisation ni finalisation ;
  rangs MPI de 0 à $N-1$, images de 1 à N : décalage de un.
- `mpif90 -g -Wall -Wextra -Wpedantic -Werror -fcheck=all -Og qui_je_suis.f90 -o qui_je_suis && \`
  `mpiexec -n 7 qui_je_suis`
- `caf -g -Wall -Wextra -Wpedantic -Werror -fcheck=all -Og qui_je_suis_coarray.f90 -o qui_je_suis_caf && \`
  `cafrun -n 7 qui_je_suis_caf`

CC BY SA



Anatomie d'un coarray

```
integer, parameter :: nx = 2
integer :: valeur[*]            ! 1 entier par image
real    :: a(100)[*]            ! 100 réels par image
real    :: grille(10,10)[nx,*] ! cobornes multiples
real, allocatable :: b(:)[:]   ! corang différé

moi = this_image ()
valeur = 10 * moi               ! sans crochets : LOCAL
a(moi)[1] = real (moi)          ! écriture chez l'image 1
sync all                        ! tout le monde a écrit
dernier = valeur[num_images()] ! lecture chez la dernière
allocate (b(100)[*])            ! COLLECTIF
```

- Un coarray est une variable présente sur *chaque* image, avec la même forme partout.
- Sans crochets : accès local. Avec crochets : accès à une autre image. La communication est *visible* dans le source.
- `[*]` : dernière coborne ouverte, déduite de `num_images()`.
  `this_image(grille)`, `image_index` : coindices de l'image / image d'un coindice.
- `allocate`/`deallocate` d'un coarray : collectifs et synchronisants.
- Aucun ordre garanti entre images sans `sync`.

CC BY SA



Dictionnaire MPI → coarray

MPI	**Fortran coarray**
`MPI_INIT`, `MPI_FINALIZE`	implicite
`MPI_COMM_RANK`, `MPI_COMM_SIZE`	`this_image()`, `num_images()`
`MPI_SEND` + `MPI_RECV`	affectation `a[j] = x` + `sync images`
`MPI_BARRIER`	`sync all`
`MPI_BCAST`, `MPI_REDUCE`, `MPI_ALLREDUCE`	`co_broadcast`, `co_sum`, `co_min`, `co_max`, `co_reduce` (Fortran 2018)
`MPI_GATHER(V)`, `MPI_SCATTER`, `MPI_ALLGATHER`	affectations de sections de coarray
`MPI_PUT`, `MPI_GET`, `MPI_WIN_FENCE`	accès natif `a(:)[j]` + `sync memory` / `sync all`
`MPI_TYPE_*` (types dérivés)	sections de tableaux, types dérivés Fortran



Synchronisations et exclusion mutuelle

Fortran	**Rôle**	**MPI (≈)**
`sync all`	barrière + ordre mémoire	`MPI_BARRIER`
`sync images (liste)`	rendez-vous avec des images ciblées	`Send/Recv`
`sync memory`	ordre mémoire, sans attente	`MPI_WIN_SYNC`
`critical…end critical`	une image à la fois	—
`lock/unlock` (`lock_type`)	verrou	`MPI_WIN_LOCK`
`event post/event wait`	signal sans barrière	`MPI_ISEND+WAIT`
`atomic_define/atomic_ref`	accès atomique distant	`MPI_ACCUMULATE`

CC BY SA



Point à point : Send/Recv devient une affectation

MPI (rang 2 → rang 5)

```
if (rang == 2) then
   valeur = 1000
   call MPI_SEND (valeur, 1, MPI_INTEGER,
    ↪  5, etiquette, MPI_COMM_WORLD)
elseif (rang == 5) then
   call MPI_RECV (valeur, 1, MPI_INTEGER,
    ↪  2, etiquette, MPI_COMM_WORLD,
    ↪  statut)
   print *, 'Moi, processus 5, ai reçu ',
    ↪  valeur, ' du processus 2.'
end if
```

Coarray (image 3 → image 6)

```
if (moi == 3) then                    ! rang 2 en
 ↪  MPI
   valeur [6] = 1000                   ! écriture
    ↪  directe dans l'image 6
   sync images (6)                    ! « je l'ai
    ↪  déposé »
else if (moi == 6) then               ! rang 5
 ↪  en MPI
   sync images (3)                    ! «
    ↪  j'attends que l'image 3 ait fini
    ↪  »
   print *, 'Moi, image 6, ai reçu ',
    ↪  valeur, ' de l'image 3.'
end if
```

- Plus de tampon, de type, d'étiquette ni de statut : la donnée est déposée dans la variable de l'image cible.
- La synchronisation devient explicite : sans `sync images`, l'image 6 peut lire avant l'écriture.



Plan

1 Remerciements et sources

2 Quelques réflexions sur la concurrence

3 Les syntaxes de la concurrence en Fortran

4 De MPI aux coarray

5 Les collectives

6 Pièges

7 teams

8 Limites



Broadcast

MPI

```
program bcast
  use mpi_f08, only: MPI_COMM_WORLD, MPI_INTEGER, &
       MPI_INIT, MPI_COMM_RANK, MPI_BCAST, MPI_FINALIZE
  implicit none (type, external)
  integer :: rang, valeur

  call MPI_INIT ()
  call MPI_COMM_RANK (MPI_COMM_WORLD, rang)

  if (rang == 2) valeur = rang + 1000

  call MPI_BCAST (valeur, 1, MPI_INTEGER, 2, MPI_COMM_WORLD)

  print *, 'Moi, processus ', rang, ', ai reçu ', valeur, ' du
  ↪  processus 2'

  call MPI_FINALIZE ()
end program bcast
```

Coarray

```
program bcast
  implicit none (type, external)
  integer :: valeur[*]

  if (this_image () == 3) valeur = this_image () + 1000

  call co_broadcast (valeur, source_image=3)

  print *, 'Moi, image ', this_image (), ', ai reçu ',
  ↪  valeur, ' de l'image 3'
end program bcast
```


CC BY SA



Collectives : co_broadcast, co_sum, co_reduce

```
! --- MPI_BCAST (valeur, 1, MPI_INTEGER, 2, MPI_COMM_WORLD) ---
if (this_image () == 3) valeur = this_image () + 1000
call co_broadcast (valeur, source_image=3)
print *, 'Moi, image ', this_image (), ' : valeur = ', valeur

! --- MPI_REDUCE (..., MPI_SUM, 0, ...) : résultat sur l'image 1 seulement ---
somme = merge (1000, this_image () - 1, this_image () == 1)
call co_sum (somme, result_image=1)
if (this_image () == 1) print *, 'Somme globale : ', somme

! --- MPI_ALLREDUCE (..., MPI_PROD, ...) : résultat sur toutes les images ---
produit = merge (10, this_image () - 1, this_image () == 1)
call co_reduce (produit, mul)
print *, 'Image ', this_image (), ' : produit global = ', produit
```

- Ce sont les vrais équivalents des collectives MPI : une seule ligne, sans coarray déclaré ni synchronisation à écrire.
- `co_reduce` attend une fonction `pure` (ici le produit).

CC BY SA



MPI_GATHERV : plus de comptes, ni de déplacements

MPI

```
allocate (nb_elements_recus (nb_procs), deplacements
↪  (nb_procs)) ! allocate for every image
if (rang == 2) then
   nb_elements_recus (1) = nb_valeurs / nb_procs
   if (reste > 0) nb_elements_recus (1) = nb_elements_recus
   ↪  (1) + 1
   deplacements (1) = 0
   do i = 2, nb_procs
      deplacements (i) = deplacements (i - 1) +
      ↪  nb_elements_recus (i - 1)
      nb_elements_recus (i) = nb_valeurs / nb_procs
      if (i-1 < reste) nb_elements_recus (i) =
      ↪  nb_elements_recus (i) + 1
   end do
end if
call MPI_GATHERV (valeurs, longueur_tranche, MPI_REAL,
↪  donnees, nb_elements_recus, &
   deplacements, MPI_REAL , 2, MPI_COMM_WORLD)
```

Coarray

```
! Chaque image calcule SA tranche : plus de
↪  recvcounts ni de displs à construire
longueur_tranche = nb_valeurs / nb_images + merge (1,
↪  0, moi <= reste)
debut = (moi - 1) * (nb_valeurs / nb_images) + min
↪  (moi - 1, reste) + 1
fin   = debut + longueur_tranche - 1

allocate (valeurs (longueur_tranche))
valeurs = [(1000. + debut - 1 + i, i = 1,
↪  longueur_tranche)]

donnees (debut:fin)[3] = valeurs   ! dépôt direct
↪  dans la section voulue de l'image 3
sync all
```

- Chaque image calcule sa propre tranche et la dépose directement dans `donnees(debut:fin)[3]`.

CC BY SA



MPI_SCATTER : chacun vient chercher sa tranche

MPI

```
longueur_tranche = nb_valeurs / nb_procs
allocate (donnees (longueur_tranche))

if (rang == 2) then
   allocate (valeurs (nb_valeurs))
   valeurs (:) = [(1000. + i, i = 1, nb_valeurs)]
   print *, 'Moi, processus ', rang, ' envoie mon tableau
   ↪  valeurs : ', &
        valeurs (1:nb_valeurs)
else
   allocate (valeurs (0)) ! prevent execution error on
   ↪  non-master images
end if

call MPI_SCATTER (valeurs, longueur_tranche, MPI_REAL,
↪  donnees, longueur_tranche, &
     MPI_REAL, 2, MPI_COMM_WORLD)
```

Coarray

```
allocate (valeurs (nb_valeurs)[*])    ! allocation
↪  COLLECTIVE : toutes les images, mêmes bornes
allocate (donnees (longueur_tranche))

if (moi == 3) valeurs = [(1000. + i, i = 1,
↪  nb_valeurs)]
sync all                                  ! les données de
↪  l'image 3 sont prêtes

! Chaque image vient chercher sa tranche chez l'image
↪  3
donnees = valeurs ((moi - 1) * longueur_tranche + 1 :
↪  moi * longueur_tranche)[3]
print *, 'Moi, image ', moi, ', ai reçu ', donnees, '
↪  de l'image 3'
```

- L'allocation d'un coarray est collective : toutes les images allouent le tableau complet (pas de `allocate (valeurs (0))` sur les non-racines).



Plan

1 Remerciements et sources

2 Quelques réflexions sur la concurrence

3 Les syntaxes de la concurrence en Fortran

4 De MPI aux coarray

5 Les collectives

6 Pièges

7 teams

8 Limites



Les pièges de la traduction

- **Courses de données** : aucune synchronisation implicite. Une première traduction de `point_a_point` sans `sync images` a affiché une valeur aléatoire (par exemple -2128261501) au lieu de 1000.
- **Ne pas lire son voisin trop tôt** : même type de défaut dans la traduction de `joker` (l'image 2 affichait des zéros).
- **Données locales ou distantes** : dans la traduction de `gatherv`, les tableaux de déplacements n'étaient calculés que par l'image 3 ; les autres images utilisaient des zéros.
- **Allocation collective** : oublier l'`allocate` d'un coarray allouable provoque une erreur de segmentation.
- **Réflexe** : compiler avec `-fcheck=all`, tester avec plusieurs nombres d'images et relancer plusieurs fois.

CC BY SA



Plan

1. Remerciements et sources
2. Quelques réflexions sur la concurrence
3. Les syntaxes de la concurrence en Fortran
4. De MPI aux coarray
5. Les collectives
6. Pièges
7. `teams`
8. Limites



Les équipes (*teams*) l'équivalent de `MPI_COMM_SPLIT`

MPI	**Fortran 2018**
`MPI_COMM_SPLIT (color)`	`form team (numero, equipe)`
`key`	`new_index = ...`
`MPI_COMM_RANK/SIZE` **sur le**	`this_image()/num_images()` **dans** `change`
sous-communicateur	`team`
collectives sur le sous-communicateur	`co_sum`, `co_max`…**dans** `change team`
`MPI_BARRIER (comm)`	`sync all` **dans** `change team`, **ou** `sync team`
	`(equipe)`
`MPI_COMM_FREE`	`end team`

- Contexte implicite : en MPI, on passe le communicateur à chaque appel. Ici l'*équipe courante* change le sens de `this_image`, `num_images`, `sync all` et des `co_*`, sans argument.
- Les équipes sont hiérarchiques : `change team` imbrique des équipes (parente, ancêtres), alors que les communicateurs MPI sont plus libres.

CC BY SA



Équipes exemple 1 : deux groupes, une collective par groupe

```
use, intrinsic :: iso_fortran_env, only: team_type
type(team_type) :: equipe
integer :: moi, somme

moi = this_image ()                    ! numéro GLOBAL
form team (2 - mod (moi, 2), equipe)
   ! équipe 1 : impaires ; équipe 2 : paires

change team (equipe)
   somme = moi
   call co_sum (somme)                 ! limitée à
    ↪ l'équipe
   if (this_image () == 1) &
      print *, 'équipe', team_number (), &
               ' : somme =', somme
end team
```

- Toutes les images de l'équipe courante exécutent `form team`.
- Dans le `change team`, `this_image()` est le numéro *dans l'équipe* : on sauvegarde donc le numéro global `moi` avant.
- Avec 6 images : somme 9 pour l'équipe 1 (1+3+5), 12 pour l'équipe 2 (2+4+6).
- Sans les équipes, il faudrait une réduction à la main avec des `sync images`.

CC BY SA



Équipes exemple 2 : une équipe par ligne d'une grille

```
integer, parameter :: nx = 3, ny = 2
type(team_type) :: equipe_ligne
integer :: moi, ligne, colonne
real    :: v

if (num_images () /= nx * ny) &
   error stop 'lancer avec nx*ny images'
moi     = this_image ()
ligne   = 1 + (moi - 1) / nx
colonne = 1 + mod (moi - 1, nx)

form team (ligne, equipe_ligne, new_index =
 ↪  colonne)

change team (equipe_ligne)
   v = real (moi)
   call co_max (v)                ! max sur la ligne
   if (this_image () == 1) &
      print *, 'ligne', team_number (), ' : max
       ↪  =', v
end team
```

- `ligne` = couleur (`color`) ; `colonne` = clé (`key`).
- `new_index` impose le numéro de chaque image dans la nouvelle équipe : ici, sa colonne.
- Avec 3×2 images : max 3 sur la ligne 1, max 6 sur la ligne 2.
- Les cobornes multiples (`a[nx,*]`) donnent la grille, mais pas ce découpage en sous-groupes.

CC BY SA



Équipes exemple 3 : rapporter un résultat à l'équipe initiale

```
use, intrinsic :: iso_fortran_env, only: &
     team_type, initial_team
type(team_type) :: equipe, equipe_initiale
integer :: res(2)[*]                    ! coarray de
 ↪  l'équipe initiale
integer :: moi, somme

moi = this_image ()
equipe_initiale = get_team (initial_team)
form team (2 - mod (moi, 2), equipe)

change team (equipe)
   somme = moi
   call co_sum (somme)
   if (this_image () == 1) &            ! un chef par
    ↪  équipe
      res(team_number ())[1, team =
       ↪  equipe_initiale] = somme
end team

sync all                               ! cette fois, tout
 ↪  le monde
if (moi == 1) print *, 'res =', res
```

- `[1, team = equipe_initiale]` : l'image 1 *de l'équipe initiale*, pas de l'équipe courante.
- Chaque chef écrit dans sa case `res(numéro d'équipe)` : pas de course de données.
- `end team` ne synchronise que l'équipe : le `sync all` final est nécessaire pour que l'image 1 voie les deux résultats.

CC BY SA



Équipes : pièges et support

- **Collectif sur l'équipe courante** : toutes ses images doivent exécuter `form team`, `change team` et `end team`, sinon blocage.
- **Deux numérotations** : numéro dans l'équipe courante ou dans une équipe ancêtre. Sauvegardez le numéro global avant de changer d'équipe.
- **Synchronisation limitée** : `change team`, `end team` et `sync all` ne couvrent que l'équipe.
- **Coarrays allouables** : un coarray alloué avant le `change team` ne peut pas y être désalloué.
- **Support** : à tester avec votre compilateur. `ifx` documente les équipes ; pour gfortran/OpenCoarrays, le suivi de bogues signale que les options de `form team` (`new_index`, `stat`, `errmsg`) ne sont pas implémentées, au dernier état que j'ai pu consulter.

CC BY SA



Plan

1 Remerciements et sources

2 Quelques réflexions sur la concurrence

3 Les syntaxes de la concurrence en Fortran

4 De MPI aux coarray

5 Les collectives

6 Pièges

7 teams

8 Limites



Ce que les coarray ne font pas (ou mal)

- **Entrées/sorties parallèles** : pas d'équivalent de MPI-IO (`MPI_FILE_*`, vues de fichier, écritures collectives) ; chaque image gère ses fichiers, ou une image écrit pour les autres.
- **Topologie** : pas de `MPI_CART_CREATE` ; les cobornes multiples (`a[nx,*]`) donnent une grille, sans périodicité ni calcul de voisins.
- **Sous-groupes** : les équipes (*teams*, Fortran 2018) jouent le rôle des communicateurs, mais leur support est inégal selon les compilateurs.
- **Messages à la MPI** : ni étiquettes, ni `MPI_ANY_SOURCE`, ni statut (voir `joker`) ; pas de communications non bloquantes explicites.
- **Autres collectives** : pas de `MPI_ALLTOALL` ni de `MPI_SCAN` natifs : à réécrire avec des sections de coarray.
- **Hybride coarray+OpenMP ?**



Conclusion

- Les coarray sont une écriture plus courte et plus Fortran de nombreux motifs MPI : accès direct, collectives `co_*`.
- Le prix à payer : synchroniser soi-même, et accepter un périmètre plus étroit que MPI (E/S, topologies, sous-groupes).
- Les deux approches coexistent : OpenCoarrays s'appuie d'ailleurs sur MPI.
- Plus d'exemples dans *« Modern Fortran : Building efficient parallel applications »* de Milan Curcic `https://www.manning.com/books/modern-fortran`.

Questions ?

Merci encore à l'**IDRIS** pour ses supports de formation MPI.

CC BY SA



Entrées/sorties parallèles ce qu'il reste à faire à la main

Stratégie	Principe	Atouts	Limites
Image 1 écrit pour les autres	les images déposent leur tranche chez l'image 1 (affectation de section), qui écrit	portable, fichier unique et ordonné	goulot de mémoire et de bande passante sur l'image
Un fichier par image	`write (nom, '(a,i0,a)') 'sortie_', moi, '.bin'`	trivial, passe à l'échelle	beaucoup de fichiers, dépe du nombre d'images, post-traitement
Fichier partagé, accès `stream`	chaque image écrit à sa position (`pos=`) dans le même fichier	un seul fichier, pas de goulot	comportement dépendant c compilateur et du système fichiers, pas d'agrégation
Bibliothèque parallèle (HDF5, NetCDF, ADIOS2…)	s'appuie sur MPI-IO	collectif, agrégation, format auto-décrit	mélange coarray/MPI non normalisé

- Ce qui disparaît avec MPI-IO : vues de fichier, écritures collectives avec agrégation, *hints*, E/S non bloquantes.
- Pour de gros volumes sur un système de fichiers parallèle, plusieurs petites écritures non alignées peuvent être très lentes : c'est précisément ce que l'agrégation collective de MPI-IO évite.

CC BY SA

Image 1 écrit pour les autres

```
real, allocatable :: tout(:)[:]
...
allocate (tout(nb_valeurs)[*])
tout(debut:fin)[1] = valeurs
sync all

if (moi == 1) then
   open (newunit=u, file='sortie.bin', &
         access='stream', form='unformatted', &
         status='replace')
   write (u) tout
   close (u)
end if
```

Fichier partagé en accès `stream`

```
if (moi == 1) then        ! création : UNE seule image
   open (newunit=u, file='sortie.bin', &
         access='stream', form='unformatted', &
         status='replace')
   close (u)
end if
sync all

open (newunit=u, file='sortie.bin', &
      access='stream', form='unformatted', &
      status='old', action='write')
write (u, pos = 1 + (debut - 1) &
              * (storage_size (valeurs) / 8)) valeurs
close (u)
sync all
```

- À gauche, toutes les images allouent le tableau complet (allocation collective) : seule l'image 1 en a besoin. Un type dérivé à composante allouable permet des tailles différentes selon les images.
- À droite, `status='replace'` exécuté par toutes les images pourrait tronquer le fichier en cours d'écriture : une seule image crée, puis `sync all`.
- Écritures *disjointes* seulement, et `close` puis `sync all` avant toute relecture par une autre image.



Coarray + OpenMP : est-ce possible ?

- **Oui** : deux niveaux de parallélisme, comme en MPI + OpenMP. Les images sont des processus ; chaque image peut lancer des *threads* OpenMP.
- Le standard Fortran ne définit pas l'interaction entre images et *threads* : ce qui est permis dépend du compilateur et de son runtime.
- Le vocabulaire du cours IDRIS « Programmation hybride MPI-OpenMP » se transpose :

MPI + OpenMP	**Coarray + OpenMP**
`MPI_THREAD_FUNNELED` : seul le thread maître communique	accès coarray et `co_*` hors région parallèle, ou dans `!$omp master`
`MPI_THREAD_MULTIPLE` : tous les threads communiquent	accès coarray depuis plusieurs threads : non portable (voir plus loin)

Source de la démarche : R. Lacroix et D. Lecas, *Programmation hybride MPI-OpenMP*, IDRIS (CNRS), v4.0, 2024 (CC BY-NC-SA).



Exemple 1 : réduction OpenMP, puis co_sum entre images

```
somme_locale = 0._dp
!$omp parallel do reduction (+: somme_locale)    ! niveau 2 : threads dans l'image
do i = 1, n
   somme_locale = somme_locale + x (i)
end do
!$omp end parallel do

somme_globale = somme_locale
call co_sum (somme_globale)                      ! niveau 1 : entre images, hors
 ↪  région parallèle
```

- Niveau 2 (dans l'image) : `reduction (+: somme_locale)`. Niveau 1 (entre images) : `co_sum`, hors région parallèle.
- Compilation : `caf -fopenmp` ; exécution : `OMP_NUM_THREADS=4 cafrun -n 2`.
- Résultat vérifié : somme exacte sur 2 images $\times$ 4 *threads*.

CC BY SA



Exemple 2 : calcul par tous les threads, communication par le maître

```
!$omp parallel default (shared) private (i)
!$omp do
do i = 1, n
   a (i) = 10. * this_image () + i              ! calcul : tous les threads
end do
!$omp end do                                    ! barrière implicite
!$omp master
recu (:)[voisin] = a                            ! communication : thread maître seul
sync all                                        ! synchronisation entre images
!$omp end master
!$omp barrier                                   ! les autres threads attendent le maître
!$omp do
do i = 1, n
   a (i) = a (i) + recu (i)                     ! utilisation des données reçues
end do
!$omp end do
!$omp end parallel
```

- Équivalent de `MPI_THREAD_FUNNELED` : seul le maître écrit dans l'image voisine et appelle `sync all`.
- Le `!$omp barrier` est indispensable : `master` n'impose aucune attente aux autres *threads*.

CC BY SA



Piège : accès coarray depuis plusieurs *threads*

```
!$omp parallel do num_threads (nt)
do t = 1, nt
   boite (t)[voisin] = 100 * this_image () + t   ! accès coarray DEPUIS une région
    ↪ parallèle
end do
!$omp end parallel do
```

- Avec gfortran 13 et OpenCoarrays 2.10 (OpenMPI) : arrêt immédiat sur `MPI_ERR_RMA_CONFLICT` dans `MPI_Win_lock`.
- Le *runtime* n'accepte pas des accès distants simultanés venant de plusieurs *threads* d'une même image.
- Résultat propre à cette chaîne d'outils : à retester avec le compilateur visé (ifx, Cray, ...).
- Règle prudente : communiquer hors région parallèle ou dans `!$omp master`.

CC BY SA



Hybride : coarray pour le distribué, `do concurrent` pour le local

- **Entre images** : coarray (accès distants, `sync images`, `co_sum`, ...).
- **Dans une image** : `do concurrent` déclare les itérations indépendantes ; c'est au compilateur de décider comment les exécuter.
- Même schéma que MPI + OpenMP, sans directives : tout reste du Fortran standard.

	MPI + OpenMP	**Coarray +** `do concurrent`
Niveau distribué	messages	accès coindicés
Niveau local	`!$omp parallel do`	`do concurrent`
Réduction locale	`reduction (+: s)`	`sum` **sur un tableau partiel**
Réduction globale	`MPI_ALLREDUCE`	`co_sum`

CC BY SA



Exemple 1 : diffusion 1D, halos en coarray, calcul en `do concurrent`

```
do k = 1, nstep
   if (moi > 1)  u (nloc + 1)[gauche] = u (1)       ! échange de halos : accès distants
   if (moi < np) u (0)[droite]        = u (nloc)    ! (hors do concurrent)
   sync images (voisins)                            ! les halos sont arrivés
   do concurrent (i = 1:nloc)                       ! calcul local : sans dépendance
      unew (i) = u (i) + alpha * (u (i - 1) - 2 * u (i) + u (i + 1))
   end do
   sync images (voisins)                            ! plus personne ne lit mes cellules
   u (1:nloc) = unew
end do
```

- Les échanges de halos (`u (0)`, `u (nloc+1)`) sont hors de la boucle `do concurrent`.
- Deux `sync images` par pas : arrivée des halos, puis fin de lecture par les voisins avant d'écraser les cellules.
- Vérifié : écart nul avec le calcul séquentiel (1, 3 et 4 images).

CC BY SA



Exemple 2 : réduction locale puis co_sum

```
do concurrent (i = debut:fin)                    ! niveau local : aucune dépendance
   part (i) = 4._dp / (1._dp + ((i - 0.5_dp) * h)**2)
end do

pi_approx = sum (part) * h                       ! réduction locale (intrinsèque)
call co_sum (pi_approx)                          ! réduction entre images
```

- Calcul de π par $\int_0^1 4/(1+x^2)\,\mathrm{d}x$: chaque image traite sa tranche d'indices.
- Une réduction ne s'écrit pas dans la boucle : on remplit un tableau partiel, puis sum, puis co_sum.
- Vérifié : π à 10^{-13} près sur 1, 2 et 4 images.

CC BY SA



Limites constatées avec gfortran 13 + OpenCoarrays

- `sync all` dans un `do concurrent` : erreur de compilation (instruction de contrôle d'image interdite dans le bloc).
- Un accès coindicé (`b (i)[k]`) dans la boucle est accepté et s'exécute ; toutefois, vaut sortir les communications de la boucle.
- Les clauses `local` et `reduce` sont rejetées par cette version (erreur de syntaxe) : d'où le tableau partiel de l'exemple 2.
- `do concurrent` n'impose aucun *thread* : avec gfortran 13, l'option `-ftree-parallelize-loops=4` n'a pas parallélisé ces boucles (vectorisation seulement). Le gain dépend du compilateur et de ses options.

CC BY SA