Calcul parallèle avec MPI

ANGD Plasmas froids

Guy Moebs

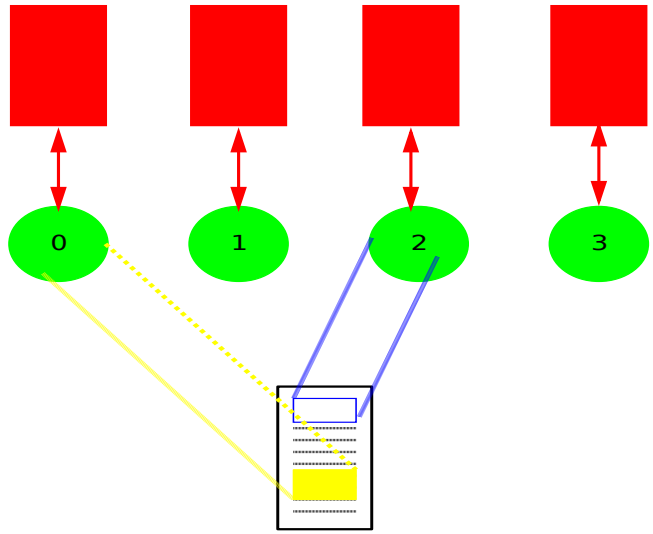
Laboratoire de Mathématiques Jean Leray,
CNRS, Université de Nantes, École Centrale de Nantes

Octobre 2011

Plan de l'exposé

Passage de messages : qu'est-ce que c'est ?

- Plusieurs processus exécutent le même programme (mais pas forcément les mêmes parties) ;



- Chaque processus dispose de ses propres données et n'a pas d'accès direct aux données des autres processus ;
- Les données du programme sont stockées dans la mémoire du processeur sur lequel s'exécute le processus ;
- Une donnée est échangée entre deux ou plusieurs processus via un appel à des routines particulières et spécialisées

MPI : qu'est-ce que c'est ?

- Message Passing Interface : bibliothèque de passage de messages
- Début des années 1990 : chaque constructeur dispose de sa propre implémentation de l'échange de messages
- Besoin d'unifier ces modèles pour parvenir à un ensemble de routines couvrant un large spectre de machines et efficacement implémentable
- "Brouillon" présenté en Novembre 1993 (Supercomputing '93)
- ...

MPI : qu'est-ce que c'est ?

L'interface avec ces routines définit un standard pratique, portable, efficace, et flexible :

- utilisable par des programmes écrits en C et Fortran,
- gestion efficace : évite les recopies en mémoire et permet le recouvrement communications / calcul,
- interface supportée par les machines de nombreux constructeurs,
- interface proche de l'existant (PVM, ...),
- sémantique indépendante du langage,
- interface conçue pour être thread-safe

MPI : historique et évolutions

- Novembre 92 (Supercomputing '92) : formalisation d'un groupe de travail créé en avril 92
- "Brouillon" présenté en Novembre 1993 (Supercomputing '93)
- MPI 1.1 publié en 1995, 1.2 en 1997 et 1.3 en 2008, avec seulement des clarifications et des changements mineurs
- MPI 2 publié en juillet 97, après deux ans de travaux
- Nouveaux groupes de travail constitués en novembre 2007 (Supercomputing '07) pour travailler sur l'évolution de MPI
- MPI 2.1 : uniquement pour des clarifications ; fusion des versions 1.3 et 2.0 ; publié en juin 2008
- MPI 2.2 : corrections jugées nécessaires au standard 2.1 ; publié en septembre 2009
- MPI 3.0 : Changements et ajouts importants par rapport à la version 2.2 ; pour un meilleur support des applications actuelles et futures, notamment sur les machines massivement parallèles et many cores ; attendu fin 2012

MPI : quelques pointeurs

- La norme MPI : `http://www.mpi-forum.org`
- Cours MPI de l'IDRIS : `http://www.idris.fr`
  $=>$ support de cours

Passage de messages avec MPI

- Il repose sur l'échange de messages entre les processus pour le transfert de données, les synchronisations, les opérations globales
- La gestion de ces échanges est réalisée par MPI (Message Passing Interface)
- Cet ensemble repose sur le principe du SPMD (*Single Program Multiple Data*)
- Chaque processus dispose de ses propres données, sans accès direct à celles des autres
- Explicite, cette technique est entièrement à la charge du développeur
- Ces échanges qui impliquent deux ou plusieurs processus se font dans un communicateur
- Chaque processus est identifié par son rang, au sein du groupe

Passage de messages avec MPI

On classe les routines de la bibliothèque en plusieurs catégories, décrites par la suite ; celles qui traitent de

1. l'environnement MPI ;
2. des communications point à point ;
3. des communications collectives ;
4. des types de données dérivés ;
5. des communicateurs ;
6. des entrées / sorties.

Passage de messages avec MPI

On classe les routines de la bibliothèque en plusieurs catégories, décrites par la suite ; celles qui traitent de

1. l'environnement MPI ;
2. des communications point à point ;
3. des communications collectives ;
4. des types de données dérivés ;
5. des communicateurs ;
6. des entrées / sorties.

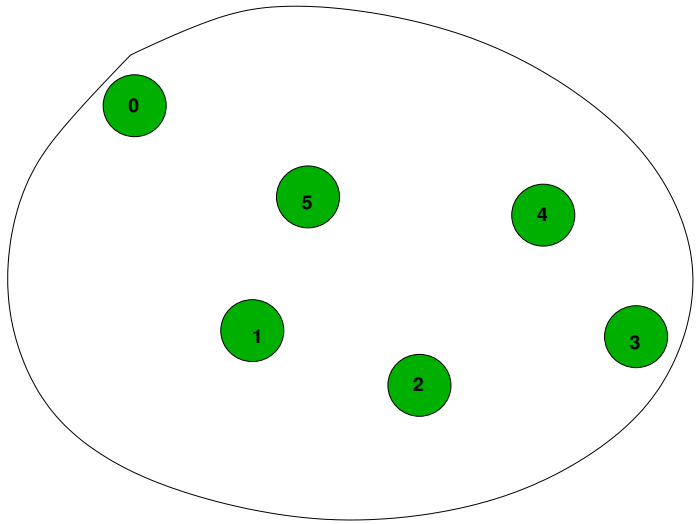
Voyons cela en détails ...

Environnement MPI

- Initialisation en début de programme (`MPI_INIT`)

```
INTEGER :: ierr = 0
CALL MPI_INIT (ierr)
```

- MPI crée alors un communicateur qui regroupe tous les processus actifs et va gérer leurs échanges de données.
  Le communicateur par défaut s'appelle `MPI_COMM_WORLD`



- Finalisation en fin de programmme (`MPI_FINALIZE`)

```
INTEGER :: ierr = 0
CALL MPI_FINALIZE (ierr)
```

Environnement MPI

- Le nombre de processus gérés par un communicateur est connu avec la routine MPI_COMM_SIZE :

```
INTEGER :: ierr = 0
INTEGER :: nbproc
CALL MPI_COMM_SIZE (MPI_COMM_WORLD, nbproc, ierr)
```

- Chaque processus est identifié par son rang, un entier entre 0 et la valeur retournée par MPI_COMM_SIZE -1, fourni par la routine MPI_COMM_RANK :

```
INTEGER :: ierr = 0
INTEGER :: myrank
CALL MPI_COMM_RANK (MPI_COMM_WORLD, myrank, ierr)
```

- les routines suivantes sont ainsi présentes dans tous les programmes MPI :

- MPI_INIT ;
- MPI_FINALIZE ;
- MPI_COMM_SIZE ;
- MPI_COMM_RANK

Fichiers d'en-tête

- Un fichier d'en-tête est toujours nécessaire (`mpif.h`, `mpi.h`)
- La norme MPI 2 crée des interfaces pour le Fortran 95, le C/C++.
- En Fortran, on dispose dorénavant d'un module qui encapsule :
  - la déclaration des constantes,
  - la définition des sous-programmes MPI.
- Ce module s'utilise de manière analogue à tout module, avec l'instruction `USE` :

```
USE mpi
```

Environnement MPI : exemple

```
PROGRAM hello
USE mpi
IMPLICIT NONE
INTEGER :: nbproc, myrank, ierr = 0

CALL MPI_INIT (ierr)
CALL MPI_COMM_SIZE (MPI_COMM_WORLD, nbproc, ierr)
CALL MPI_COMM_RANK (MPI_COMM_WORLD, myrank, ierr)

WRITE (6,*) 'Bonjour, je suis le processus de rang ', &
myrank, ' parmi ', nbproc, ' processus.'

CALL MPI_FINALIZE (ierr)
END PROGRAM hello
```

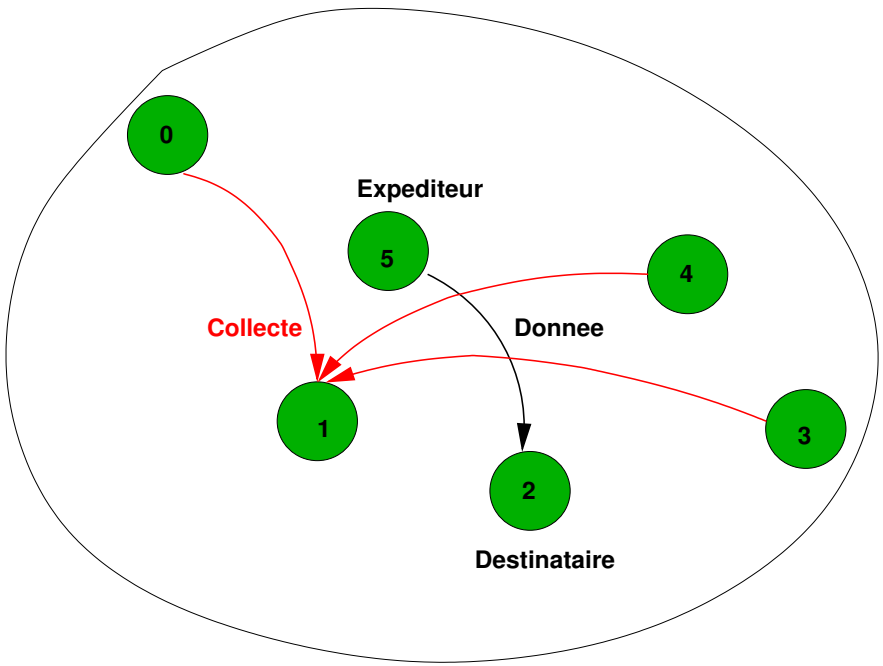
Compilation : (plate-forme !) `mpif90 -O3 tp1.f90 -o tp1.out`

Exécution : `mpirun -np 2 ./tp1.out`

```
Bonjour, je suis le processus de rang 0 parmi 2 processus.
Bonjour, je suis le processus de rang 1 parmi 2 processus.
```

Communications

- Pour réaliser des opérations impliquant des données d'autres processus, il est nécessaire d'échanger ces informations aux travers de messages
- Ces messages se font sous la forme de communications impliquant au moins deux processus
- On peut faire une analogie avec le courrier électronique



Communications point à point

- La communication point à point est une communication entre deux processus :
  $\Rightarrow$ expéditeur et destinataire
- Elle comprend deux opérations élémentaires : l'envoi et la réception
- Différents ingrédients sont nécessaires pour composer un message
  - le communicateur
  - l'identifiant du processus expéditeur
  - l'identifiant du processus destinataire
  - une étiquette (tag) qui permet au programme de distinguer différents messages
  $\Rightarrow$ ils forment l'**enveloppe du message**
- Les données envoyées sont typées, le message contient aussi :
  - la donnée à transmettre,
  - son type, intrinsèque ou dérivé,
  - sa taille

Communications point à point

```
PROGRAM msg
USE mpi
INTEGER :: myrank, ierr = 0
CHARACTER(10) :: message
INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status

CALL MPI_INIT (ierr)
CALL MPI_COMM_RANK (MPI_COMM_WORLD, myrank, ierr)

IF (myrank == 0) THEN
   message = "Salut"
   CALL MPI_SEND (message, len(message), MPI_CHARACTER, &
                  1, 99, MPI_COMM_WORLD, ierr)
ELSE
   CALL MPI_RECV (message, len(message), MPI_CHARACTER, &
                  0, 99, MPI_COMM_WORLD, status, ierr)
   WRITE (6,'(A)') message
END IF

CALL MPI_FINALIZE (ierr)
END PROGRAM msg
```

Communications point à point

- Le processus 0 (`myrank = 0`) envoie un message au processus de rang 1 (`myrank = 1`) avec l'opération d'envoi :

```
CALL MPI_SEND (buf, count, datatype, dest, tag, &
               comm, ierr)
```

- buf, count et datatype constituent le message
- L'enveloppe spécifie le destinataire et inclut des informations utilisables par le destinataire pour sélectionner un message particulier
- Ainsi c'est l'envoi :
  - d'un message identifié par tag,
  - de longueur count,
  - de type datatype,
  - à partir de l'adresse buf,
  - au processus dest,
  - dans le communicateur comm.

Communications point à point

- Le processus 1 (`myrank = 1`) réceptionne un message avec l'opération de réception :

```
CALL MPI_RECV (buf, count, datatype, src, tag, &
               comm, status, ierr)
```

- Les trois premiers arguments décrivent le buffer de réception
- Les trois suivants permettent de choisir le message voulu
- Le dernier (hors code d'erreur) contient des informations sur le message juste reçu
- Important : l'étiquette doit être la même pour les deux processus
- Ainsi c'est la réception :
  - d'un message identifié par tag,
  - de longueur count,
  - de type datatype,
  - à partir de l'adresse buf,
  - en provenance du processus src,
  - dans le communicateur comm

- `CALL MPI_SEND (buf, count, datatype, dest, tag, comm, ierr)`

buf	IN	au choix
count	IN	entier non négatif
datatype	IN	objet MPI
dest	IN	entier
tag	IN	entier
comm	IN	objet MPI
ierr	OUT	entier

- `CALL MPI_RECV (buf, count, datatype, src, tag, comm, status, ierr)`

buf	OUT	au choix
count	IN	entier non négatif
datatype	IN	objet MPI
src	IN	entier
tag	IN	entier
comm	IN	objet MPI
status	OUT	tableau d'entiers
ierr	OUT	entier

Communications point à point : premier bilan

- `MPI_SEND (buf, count, datatype, dest, tag, comm, ierr)`

- ```
  INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status
  MPI_RECV (buf, count, datatype, src, tag, comm,
            status, ierr)
  ```

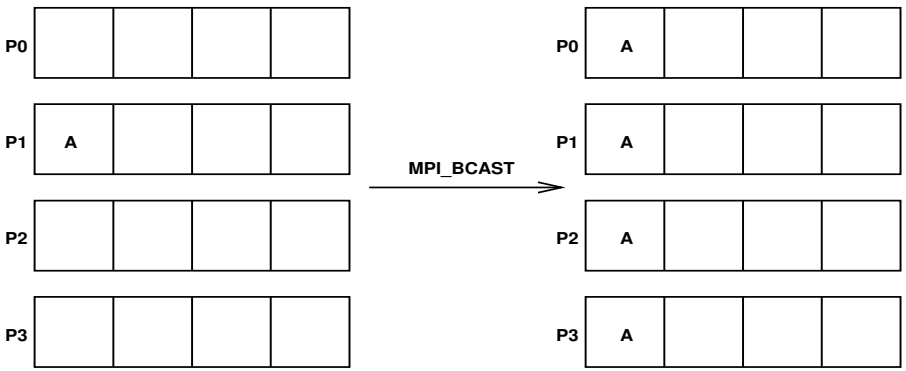
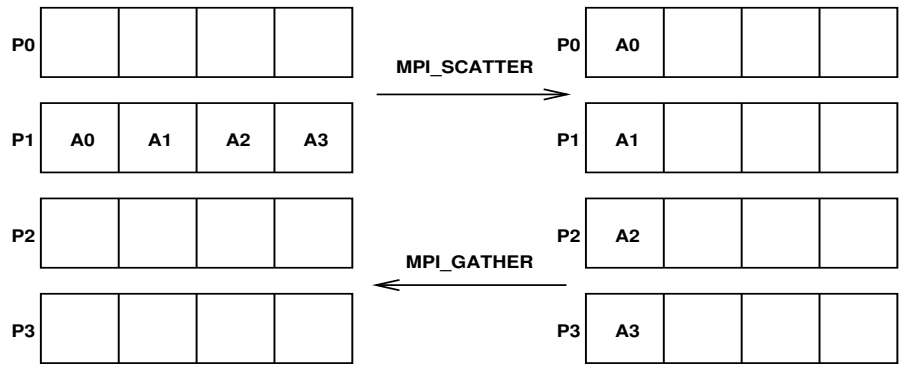
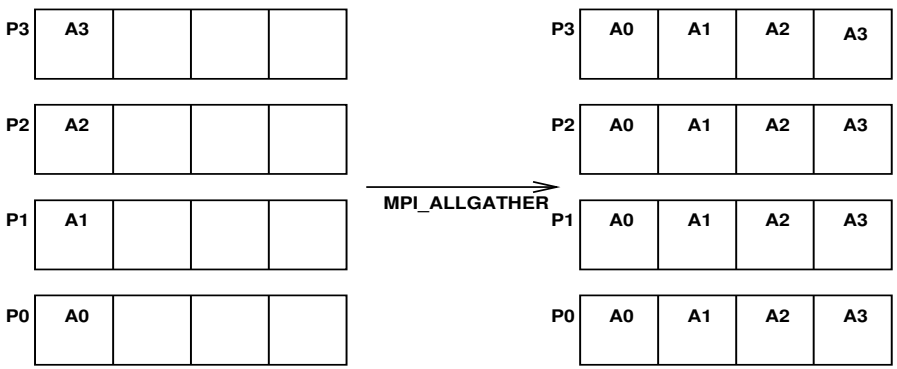
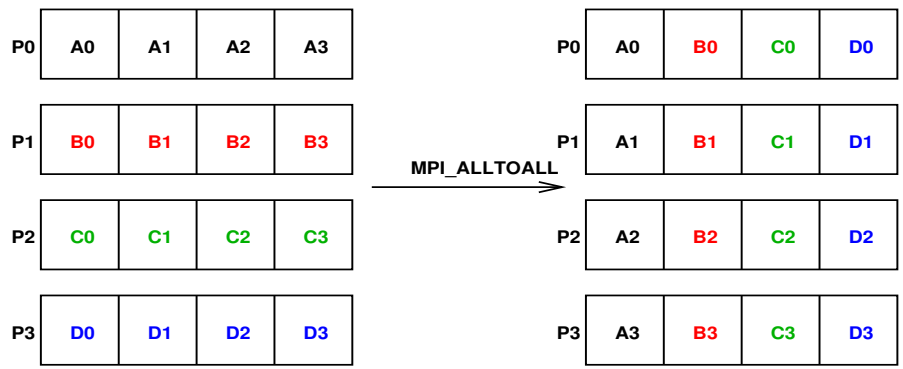
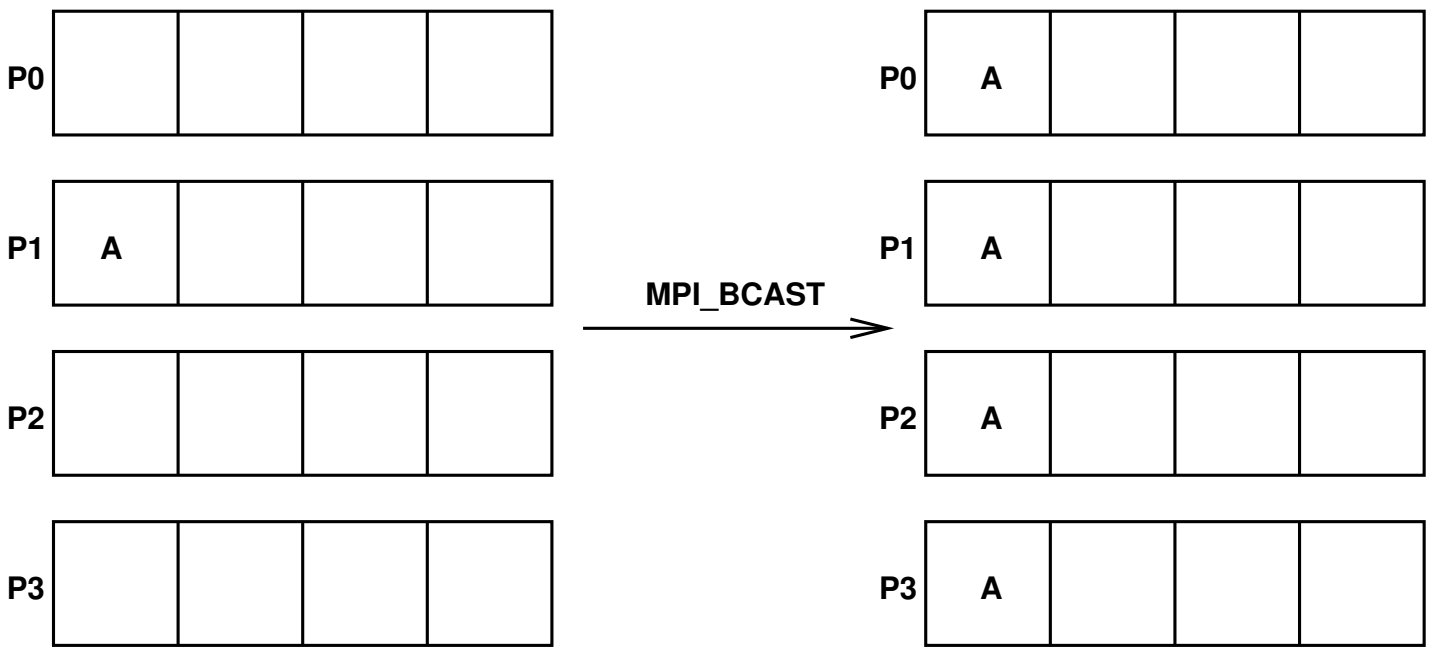
- Envoi et réception d'une donnée vers un destinataire avec attente de la fin de la communication avant de continuer l'exécution
- Un message peut être reçu si on a la correspondance sur l'enveloppe (source, tag, comm)

- **Sécurité** : on a la donnée envoyée ou reçue (resp.) avant de la modifier / l'employer (resp.)
- **Inconvénient** : attente de la fin de l'opération, voire de l'opération elle-même pour continuer les calculs
- Autres possibilités .... à suivre
  ```

Principaux types de données intrinsèques : Fortran

- Les types des données sont transmis à MPI au travers de constantes

Type de données MPI	Type de données Fortran
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)

Principaux types de données intrinsèques : C

- Les types des données sont transmis à MPI au travers de constantes

Type de données MPI	Type de données C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Communications point à point : autres possibilités

- On peut recevoir un message de n'importe quelle source : `MPI_ANY_SOURCE`
  $\Rightarrow$ On ne connait pas forcément à la compilation le processus qui va envoyer le message

- On peut recevoir un message muni de n'importe quelle étiquette : `MPI_ANY_TAG`
  $\Rightarrow$ On ne connait pas forcément à la compilation l'ordre d'arrivée des messages

- L'argument `status` des réceptions contient la bonne étiquette et le rang de l'expéditeur

Communications point à point : autres possibilités

- On peut envoyer un message à un processus n'existant pas : `MPI_PROC_NULL`
  $\Rightarrow$ Cela permet d'éviter des tests fastidieux (conditions aux limites, ...)

- On peut envoyer des messages dont le contenu n'est pas une donnée de type intrinsèque, on utilise les types dérivés (voir plus loin)

- On peut effectuer un envoi et une réception en une seule communication : `MPI_SENDRECV`

- On peut échanger des données en une seule communication : `MPI_SENDRECV_REPLACE`

Optimisation des communications point à point

- Important d'optimiser les communications : gain en perf.
  $\Rightarrow$ Minimiser le temps passé à faire autre chose que des calculs (i.e. l'*overhead*)

- Différentes possibilités :
  - recouvrir les communications par des calculs
  - éviter la recopie du message dans une zone mémoire temporaire
  - minimiser les surcoûts dûs aux appels répétés aux routines de communications

- Différents types de communication :
  - communications standards
  - communications synchrones
  - communications bufférisées

Communications standards bloquantes

- ```
  CALL MPI_SEND (buf, count, datatype, dest, tag, &
                 comm, ierr)
  ```
- ```
  CALL MPI_RECV (buf, count, datatype, src,  tag, &
                 comm, status, ierr)
  ```

- Communication telle que la réception n'a lieu que si le message envoyé est arrivé (communication bloquante)

- Ceci est assuré (pour l'expéditeur) soit :
  - par bufférisation : recopie du message dans un buffer ;
  - par synchronisation : avant de continuer le calcul après un envoi, on attend que la réception correspondante soit initialisée

- C'est le cas des routines `MPI_SEND` et `MPI_RECV`
- Le `MPI_SEND` devient bloquant lorsque le message est trop gros pour le bufférisér.

Communications standards non bloquantes

- Le programme continue avant l'initialisation de la réception correspondante
  $\Rightarrow$ Cela permet de calculer pendant le transfert

- ```
  CALL MPI_ISEND (buf, count, datatype, dest, tag, &
                  comm, request, ierr)
  ```

- ```
  CALL MPI_IRECV (buf, count, datatype, src , tag, &
                  comm, request, ierr)
  ```

- L'argument `request` permet d'identifier les opérations de communication impliquées et de les faire correspondre

Tests des communications pour complétion

- Les commandes associées pour tester :
- `CALL MPI_WAIT` (request, status, ierr)
  $\Rightarrow$ Attendre jusqu'à la terminaison d'une requête
- `CALL MPI_TEST` (request, flag, status, ierr)
  $\Rightarrow$ Vérifier si une requête est terminée (flag = `.TRUE.`)
- `CALL MPI_PROBE` (source, tag, status, comm, ierr)
  $\Rightarrow$ Contrôler sans réceptionner si une requête est arrivée
  Version non bloquante : `MPI_IPROBE`

Tests des communications pour complétion

- Il existe des variantes pour tester des groupes de communication :

Test de complétion	WAIT bloquant	TEST interroge
Au moins une renvoie exactement une	MPI_WAITANY	MPI_TESTANY
Toutes	MPI_WAITALL	MPI_TESTALL
Au moins une renvoie toutes celles finies	MPI_WAITSOME	MPI_TESTSOME

- Le test de complétion désalloue la requête d'une communication non bloquante qui est finie

Communications synchrones

- Communication synchrone :
  avant de continuer le calcul après un envoi, on attend que la réception correspondante soit initialisée ; il n'y a pas de "bufférisation"

- ```
  CALL MPI_SSEND (buf, count, datatype, dest, tag, &
                  comm, ierr)
  ```

- La complétion ne peut se faire que si la réception a été postée (i.e. prête)

- La complétion d'un `MPI_SSEND` indique donc que :
  - la donnée envoyée est à nouveau disponible (i.e. modifiable),
  - la réception a commencé
  ```

Communications bufférisées

- Le message est recopié dans un buffer avant d'être envoyé
- Cette opération peut être commencée (et terminée) sans que la réception correspondante ne soit prête
- ```
  CALL MPI_BSEND (buf, count, datatype, dest, tag, &
                  comm, ierr)
  ```
- Les allocations des buffers mémoire MPI sont gérées avec

  `MPI_BUFFER_ATTACH (buffer, size, ierr)`

  `MPI_BUFFER_DETACH (buffer, size, ierr)`
- Le buffer est alloué dans la mémoire locale du processus expéditeur
- Le buffer est uniquement utilisable pour les messages bufférisés
- Un seul buffer alloué à la fois par processus
  ```

Bilan des routines disponibles

- Communications usuelles

Envois		
MPI_SEND	bloquant	standard
MPI_ISEND	non bloquant	standard
MPI_SSEND	bloquant	synchrone
MPI_ISSEND	non bloquant	synchrone
MPI_BSEND	bloquant	bufférisé
MPI_IBSEND	non bloquant	bufférisé
Réceptions		
MPI_RECV	bloquante	standard
MPI_IRECV	non bloquante	standard

Quelques règles simples ... selon les situations

- Initier les réceptions avant les envois
  $\Rightarrow$ En cas d'échange, mettre les appels à `MPI_IRECV` avant les `MPI_SEND`

- Recouvrir les communications par des calculs
  $\Rightarrow$ Utiliser des communications non bloquantes `MPI_ISEND` et `MPI_IRECV`

N.B. : les performances des différents types de communication point à point sont dépendantes (entre-autres !) des implémentations MPI.

Communications collectives

- La communication collective est une communication qui implique un ensemble de processus, tous ceux du communicateur fourni en argument
  $\Rightarrow$ C'est un argument essentiel

- En une seule opération, on effectue une série de communications point à point

- L'ensemble des processus effectuent le même appel avec des arguments correspondants

Communications collectives

- La sémantique et la syntaxe des opérations collectives sont analogues à celles des communications point à point
- Il n'y a pas d'étiquette dans les appels à ces routines
- Certaines routines ont un processus qui est seul expéditeur ou seul destinataire :
  $\Rightarrow$ il s'appelle le processus root
- Certains arguments n'ont alors de sens que pour le processus root

Communications collectives

Il existe différents types de communication collective :

Opérations de diffusion / collecte de données

- Diffusion globale de données : `MPI_BCAST`
- Distribution sélective de données : `MPI_SCATTER`
- Collecte de données réparties : `MPI_GATHER`
- Collecte, par tous les processus, de données réparties : `MPI_ALLGATHER`
- Distribution sélective, par tous les processus, de données réparties : `MPI_ALLTOALL`

Communications collectives

Opérations préalables sur des données réparties

- Opérations de réduction : MPI_REDUCE, MPI_REDUCE_SCATTER
- Opération de réduction partielle : MPI_SCAN
- Opération de réduction avec diffusion globale du résultat : MPI_ALLREDUCE

Barrières de synchronisation globale

- `CALL MPI_BARRIER (comm, ierr)`
- Chaque processus appelant est bloqué jusqu'à ce que tous les processus du communicateur font l'appel

Principales fonctions d'échanges









Diffusion globale de données

- `CALL MPI_BCAST (buf, count, datatype, root, comm, ierr)`
- Le processus root envoie un message :
  à tous les processus du communicateur comm,
  de longueur count,
  de type datatype,
  à partir de l'adresse buf



Diffusion globale de données

```
IMPLICIT NONE
INTEGER :: ierr = 0
INTEGER :: u, myrank, nbproc

CALL MPI_INIT (ierr)
CALL MPI_COMM_SIZE (MPI_COMM_WORLD, nbproc, ierr)
CALL MPI_COMM_RANK (MPI_COMM_WORLD, myrank, ierr)

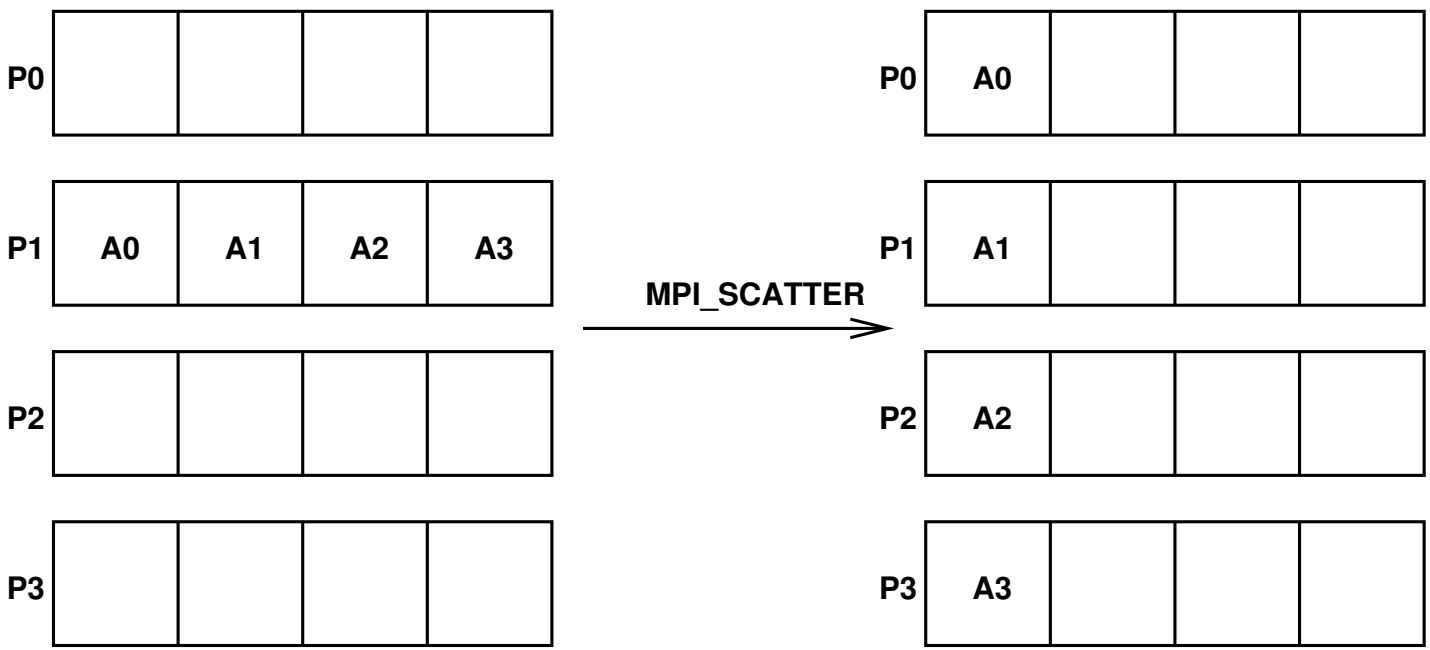
IF (myrank == 0) READ (5,*) u
CALL MPI_BCAST (u, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, &
                ierr)

WRITE (6,*) 'processus ', myrank, ' u = ', u
call MPI_FINALIZE (ierr)
END PROGRAM bcast
Exécution : echo 10 | mpirun -np 3 ./bcast.out
processus 0 u = 10
processus 2 u = 10
processus 1 u = 10
```

Distribution sélective de données

- ```
  CALL MPI_SCATTER (sendbuf, sendcount, sendtype, &
                    recvbuf, recvcount, recvtype, &
                    root, comm, ierr)
  ```

- Le processus root envoie un message étant :
  de longueur sendcount,
  de type sendtype,
  à partir de l'adresse sendbuf,
  à chacun des processus du communicateur comm

- Chaque processus du communicateur comm (root inclus) indique recevoir un message :
  de longueur recvcount,
  de type recvtype,
  à partir de l'adresse recvbuf
  ```

Distribution sélective de données

L'action de la routine `MPI_SCATTER` peut être vue comme

- La donnée initiale est partagée en n morceaux contigus de taille identique :
  $\Rightarrow$ le $i^{ème}$ morceau est envoyé au $i^{ème}$ processus
- Les arguments relatifs à la donnée envoyée n'ont de sens que pour le processus root



Distribution sélective de données

La routine MPI_SCATTER peut être vue comme

(i) le processus root effectue n envois :

```
CALL MPI_SEND (sendbuf+i.sendcount.extent(sendtype), &
               sendcount, sendtype, i, ...)
```

(ii) et chaque processus (root inclus) du communicateur exécute la réception suivante :

```
CALL MPI_RECV (recvbuf, recvcount, recvtype, &
               root, ...)
```

- extent(sendtype) = MPI_TYPE_EXTENT(sendtype) :
  le nombre d'octets en mémoire

Distribution sélective de données

```
INTEGER, PARAMETER :: nb_valeurs = 128
INTEGER :: i, myrank, nbproc, lbloc, ierr = 0
REAL(4), ALLOCATABLE, DIMENSION(:) :: valeurs, donnees

lbloc = nb_valeurs / nbproc
ALLOCATE (donnees(lbloc), STAT=ierr )

IF (myrank == 3) THEN
 ALLOCATE (valeurs(nb_valeurs), STAT=ierr )
 valeurs(:) = (/(1000.0_4+REAL(i,4),i=1,nb_valeurs)/)
END IF

CALL MPI_SCATTER (valeurs, lbloc, MPI_REAL, &
                  donnees, lbloc, MPI_REAL, 3, &
                  MPI_COMM_WORLD, ierr)

WRITE (6,*) 'processus ', myrank, ' a recu ', &
            donnees(1), ' a ', donnees(lbloc), &
            ' du processus 3'
```

Distribution sélective de données

```
mpirun -np 4 ./scatter.out
processus 3 a recu 1097. a 1128. du processus 3
processus 1 a recu 1033. a 1064. du processus 3
processus 2 a recu 1065. a 1096. du processus 3
processus 0 a recu 1001. a 1032. du processus 3

mpirun -np 5 ./scatter.out
processus 1 a recu 1026. a 1050. du processus 3
processus 2 a recu 1051. a 1075. du processus 3
processus 3 a recu 1076. a 1100. du processus 3
processus 4 a recu 1101. a 1125. du processus 3
processus 0 a recu 1001. a 1025. du processus 3
```

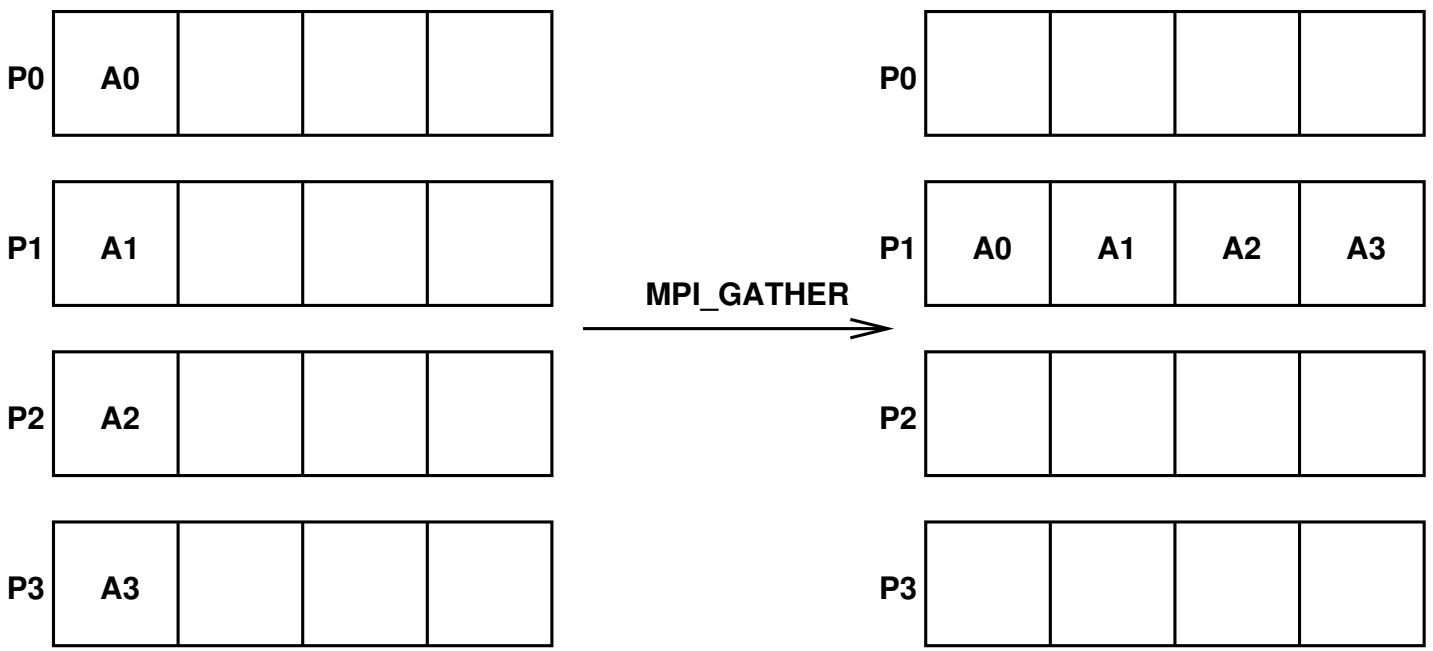
Distribution sélective de données

- L'affichage ne se fait pas forcément dans l'ordre des rangs
- Attention : Toutes les valeurs ne sont pas distribuées si le nombre de destinataires n'est pas un diviseur du nombre de données à envoyer : **il faut gérer l'éventuel reliquat**
- On peut être amené à gérer des paquets de taille variable
- La routine `MPI_SCATTERV` étend les possibilités aux distributions non uniformes.

Collecte de données réparties

Chaque processus envoie son buffer au processus root :
$\Rightarrow$ opération inverse de MPI_SCATTER

- ```
  CALL MPI_GATHER (sendbuf, sendcount, sendtype, &
                   recvbuf, recvcount, recvtype, &
                   root, comm, ierr)
  ```
- Chaque processus (root inclus) du communicateur comm indique envoyer un message :
  au processus root,
  de longueur sendcount, de type sendtype,
  à partir de l'adresse sendbuf

- Le processus root reçoit ces messages et les concatène dans l'ordre des rangs, chacun étant :
  de longueur recvcount,
  de type recvtype,
  à partir de l'adresse recvbuf
  ```

Collecte de données réparties

L'action de la routine `MPI_GATHER` peut être vue comme

- Chaque processus envoie un ensemble de données, de nature *compatible*
- Le processus root effectue n réceptions :
  $\Rightarrow$ Le morceau du $i^{ème}$ processus va en $i^{ème}$ position
- N.B. : les arguments relatifs à la donnée reçue n'ont de sens que pour le processus root



Collecte de données réparties

La routine MPI_GATHER peut être vue comme

(i) chaque processus (root inclus) du communicateur effectue l'envoi :

```
CALL MPI_SEND (sendbuf, sendcount, sendtype, &
               root, ...)
```

(ii) et le processus root effectue n réceptions :

```
CALL MPI_RECV (recvbuf+i.recvcount.extent(recvtype), &
               recvcount, recvtype, i, ...)
```

- extent(recvtype) = MPI_TYPE_EXTENT(recvtype) :
  le nombre d'octets en mémoire

Collecte de données réparties

```
INTEGER, PARAMETER :: nb_valeurs = 128
INTEGER :: i, myrank, nbproc, lbloc, ierr = 0
INTEGER, ALLOCATABLE,DIMENSION(:) :: valeurs, donnees

lbloc = nb_valeurs / nbproc
ALLOCATE (valeurs(lbloc), STAT=ierr )
valeurs(:) = (/(1000+i+lbloc*myrank,i=1,lbloc)/)

WRITE (6,*) 'processus ', myrank, ' possede ', &
            valeurs(1), ' a ', valeurs(lbloc)
IF (myrank == 2) &
      ALLOCATE (donnees(nb_valeurs), STAT=ierr)

CALL MPI_GATHER (valeurs, lbloc, MPI_INTEGER, &
                 donnees, lbloc, MPI_INTEGER, 2, &
                 MPI_COMM_WORLD, ierr)

IF (myrank == 2) &
      WRITE (6,*) 'processus 2 a recu ', donnees(1), &
                  '...', donnees(lbloc+1), '...', &
                  donnees(nb_valeurs)
```

Collecte de données réparties

```
mpirun -np 4 ./gather.out
processus 2 possede 1065 a 1096
processus 1 possede 1033 a 1064
processus 3 possede 1097 a 1128
processus 0 possede 1001 a 1032
processus 2 a recu 1001 ... 1033 ... 1128

mpirun -np 3 ./gather.out
processus 0 possede 1001 a 1042
processus 1 possede 1043 a 1084
processus 2 possede 1085 a 1126
processus 2 a recu 1001 ... 1043 ... -370086
```

Collecte de données réparties

- L'affichage ne se fait pas forcément dans l'ordre des rangs
- Attention : Toutes les valeurs ne sont pas initialisées et transmises correctement si le nombre d'expéditeurs n'est pas un diviseur du nombre de données à envoyer
- On peut être amené à gérer des paquets de taille variable
- La routine `MPI_GATHERV` étend les possibilités aux collectes non uniformes.

Collecte, par tous les processus, de données réparties

La routine MPI_ALLGATHER peut être vue comme MPI_GATHER où tous les processus sont destinataires du résultat et non pas uniquement le processus root

- ```
  CALL MPI_ALLGATHER (sendbuf, sendcount, sendtype, &
                      recvbuf, recvcount, recvtype, &
                      comm, ierr)
  ```

- Chaque processus du communicateur comm indique envoyer un message :
  de longueur sendcount, de type sendtype,
  à partir de l'adresse sendbuf

- Chaque processus reçoit ces messages et les concatène dans l'ordre des rangs, chacun étant :
  de longueur recvcount, de type recvtype,
  à partir de l'adresse recvbuf
  ```

Collecte, par tous les processus, de données réparties

Un appel à la routine MPI_ALLGATHER peut être vu comme :

- Tous les processus effectuent un appel à MPI_GATHER

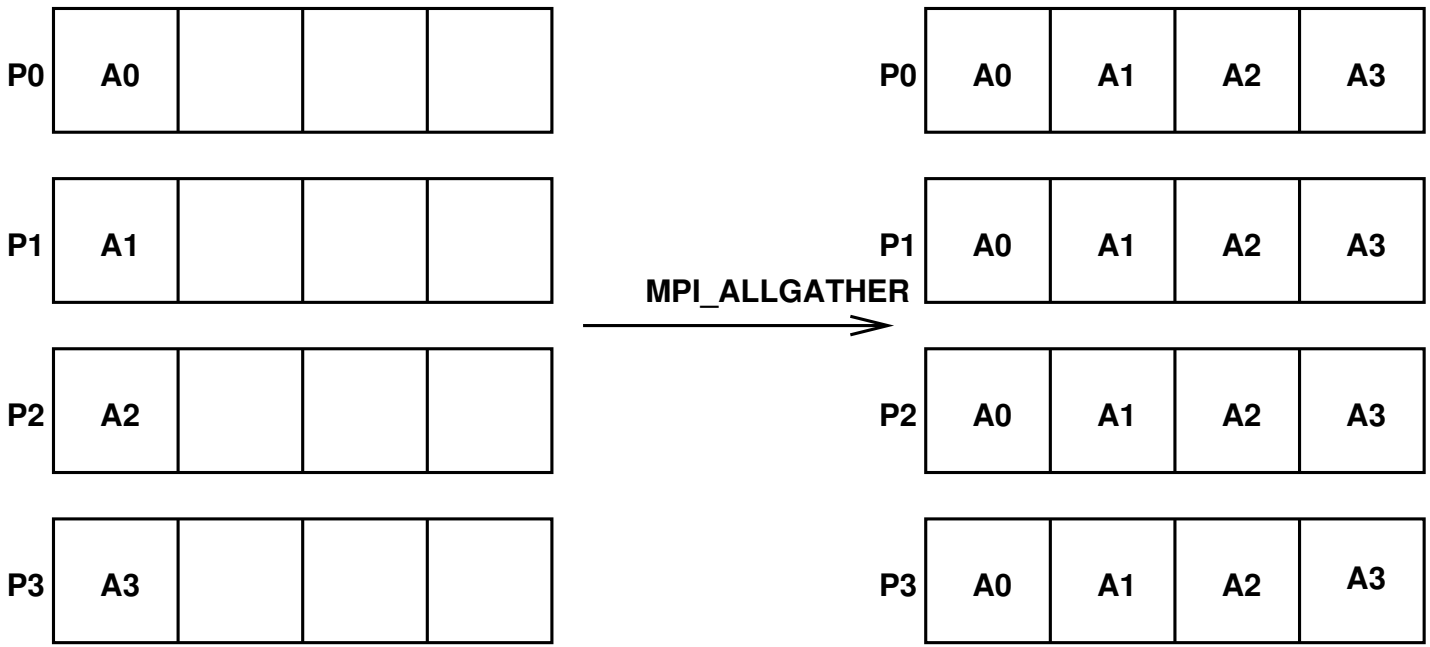
```
CALL MPI_GATHER (sendbuf, sendcount, sendtype, &
                 recvbuf, recvcount, recvtype, &
                 root, comm, ierr),
```

- Tous les processus effectuent un appel à MPI_BCAST

  $\Rightarrow$ Même processus root !

```
CALL MPI_BCAST (recvbuf, recvcount, recvtype, &
                root, comm, ierr)
```

Collecte, par tous les processus, de données réparties



Collecte, par tous les processus, de données réparties

```
INTEGER, PARAMETER :: nb_valeurs = 128
INTEGER :: i, myrank, nbproc, lbloc, ierr = 0
INTEGER, ALLOCATABLE, DIMENSION(:) :: valeurs, donnees

lbloc = nb_valeurs / nbproc
ALLOCATE (valeurs(lbloc), donnees(nb_valeurs) )
valeurs(:) = (/(1000+i+lbloc*myrank,i=1,lbloc)/)

WRITE (6,*) 'processus ', myrank, ' possede ', &
              valeurs(1), ' a ', valeurs(lbloc)

CALL MPI_ALLGATHER (valeurs, lbloc, MPI_INTEGER, &
                    donnees, lbloc, MPI_INTEGER, &
                    MPI_COMM_WORLD, ierr)

WRITE (6,*) 'processus ', myrank, ' a recu ', &
              donnees(1), '...', donnees(lbloc+1), &
              '...', donnees(nb_valeurs)
```

Collecte, par tous les processus, de données réparties

```
mpirun -np 2 ./allgather.out
processus 0 possede 1001 a 1064
processus 0 a recu 1001 ... 1065 ... 1128
processus 1 possede 1065 a 1128
processus 1 a recu 1001 ... 1065 ... 1128

mpirun -np 3 ./allgather.out
processus 0 possede 1001 a 1042
processus 2 possede 1085 a 1126
processus 1 possede 1043 a 1084
processus 0 a recu 1001 ... 1043 ... -370086
processus 1 a recu 1001 ... 1043 ... -370086
processus 2 a recu 1001 ... 1043 ... -370086
```

Collecte, par tous les processus, de données réparties

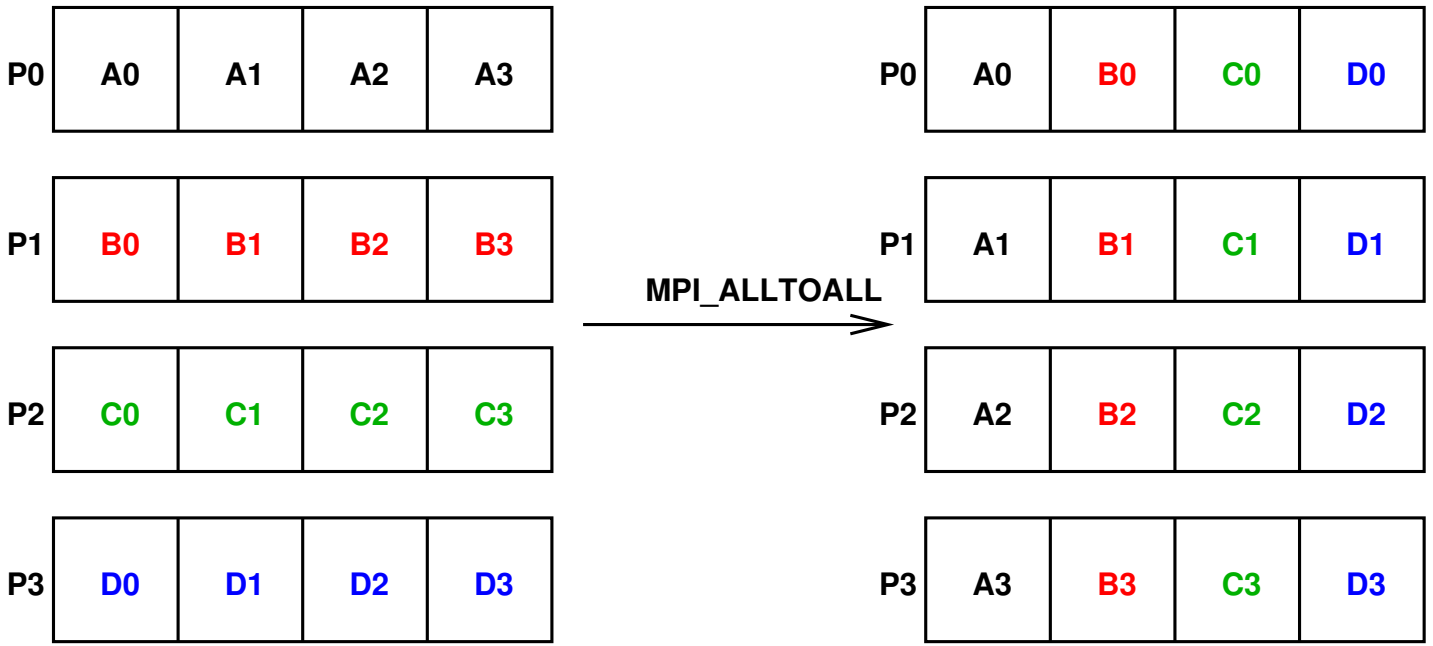
- L'affichage ne se fait pas forcément dans l'ordre des rangs
- Attention : Toutes les valeurs ne sont pas initialisées et transmises correctement si le nombre d'expéditeurs n'est pas un diviseur du nombre de données à envoyer : **il faut gérer l'éventuel reliquat**
- On peut être amené à gérer des paquets de taille variable
- La routine `MPI_ALLGATHERV` étend les possibilités aux collectes non uniformes

Distribution sélective, par tous les processus, de données réparties

La routine MPI_ALLTOALL est une extension de MPI_ALLGATHER dans le cas où chaque processus envoie des données distinctes à chacun des destinataires

- ```
  CALL MPI_ALLTOALL (sendbuf, sendcount, sendtype, &
                     recvbuf, recvcount, recvtype, &
                     comm, ierr)
  ```
- Chaque processus envoie un message étant :
  de longueur sendcount, de type sendtype,
  à partir de l'adresse sendbuf,
  à chacun des processus du communicateur comm

- Chaque processus du communicateur comm indique recevoir un message :
  de longueur recvcount, de type recvtype,
  à partir de l'adresse recvbuf,
  de chacun des processus du communicateur comm
  ```

Distribution sélective, par tous les processus, de données réparties

- Tous les arguments sont significatifs **pour tous les processus**

Un appel à la routine MPI_ALLTOALL peut être vu comme si tous les processus effectuent *n* envois et *n* réceptions distincts

```
CALL MPI_SEND (sendbuf+i.sendcount.extent(sendtype), &
               sendcount, sendtype, i, ...)

CALL MPI_RECV (recvbuf+j.recvcount.extent(recvtype), &
               recvcount, recvtype, j, ...)
```

Le j^{ème} bloc du i^{ème} processus est envoyé au j^{ème} processus et placé dans son i^{ème} bloc

Distribution sélective, par tous les processus, de données réparties



La routine `MPI_ALLTOALLV` ajoute de la flexibilité à `MPI_ALLTOALL` dans les tailles et adresses de localisation dans les buffers d'envoi et de réception

Distribution sélective, par tous les processus, de données réparties

```
INTEGER, PARAMETER :: nb_valeurs = 4
INTEGER :: i, myrank, nbproc, lbloc, ierr = 0
INTEGER, DIMENSION(nb_valeurs) :: valeurs, donnees

lbloc = nb_valeurs / nbproc
valeurs(1:nb_valeurs) = &
(/(1000+i+lbloc*myrank*100, i=1,nb_valeurs)/)

WRITE (6,*) 'processus ', myrank, ' avant ', &
            valeurs(1:nb_valeurs)

CALL MPI_ALLTOALL (valeurs, lbloc, MPI_INTEGER, &
                   donnees, lbloc, MPI_INTEGER, &
                   MPI_COMM_WORLD, ierr)

WRITE (6,*) 'processus ', myrank, ' apres ', &
            donnees(1:nb_valeurs)
```

Diffusion sélective, par tous les processus, de données réparties

```
mpirun -np 4 ./alltoall.out

processus 0 avant 1001, 1002, 1003, 1004
processus 1 avant 1101, 1102, 1103, 1104
processus 2 avant 1201, 1202, 1203, 1204
processus 3 avant 1301, 1302, 1303, 1304

processus 0 apres 1001, 1101, 1201, 1301
processus 1 apres 1002, 1102, 1202, 1302
processus 2 apres 1003, 1103, 1203, 1303
processus 3 apres 1004, 1104, 1204, 1304
```

- L'affichage ne se fait pas forcément dans l'ordre des rangs

Opérations préalables sur des données réparties

Ces fonctions réalisent une opération globale de réduction, i.e. l'obtention d'un résultat à partir de données réparties

- ```
  CALL MPI_REDUCE (sendbuf, recvbuf, count, &
                   datatype, op, root, comm, ierr)
  ```

- Depuis tous les processus du communicateur comm, on envoie un message :
  de longueur count, de type datatype,
  à partir de l'adresse sendbuf,
  pour faire l'opération op sur ces valeurs.
  $\Rightarrow$ Réception du résultat à partir de l'adresse recvbuf du processus root

- Si le nombre de données envoyées est supérieur à un, l'opération op est appliquée sur chacune des données

- l'opération op est supposée associative
  ```

Opérations préalables sur des données réparties

- Opérations prédéfinies :

MPI_MAX	:	maximum	MPI_MIN	:	minimum
MPI_SUM	:	somme	MPI_PROD	:	produit
MPI_LAND	:	ET logique	MPI_LOR	:	OU logique

  MPI_LXOR : OU exclusif logique
  MPI_MAXLOC : maximum et position
  MPI_MINLOC : minimum et position

- Les opérations MPI_MAXLOC et MPI_MINLOC nécessitent un type de données valeur, indice pour être utilisées
  $\Rightarrow$ Les types proposés par MPI : MPI_2INTEGER, MPI_2REAL, MPI_2DOUBLE_PRECISION, MPI_2COMPLEX, MPI_2DOUBLE_COMPLEX

- Opérations de réductions personnelles :
  les opérateurs MPI_OP_CREATE et MPI_OP_FREE permettent de créer et détruire des opérations personnalisées

Opérations préalables sur des données réparties

```
INTEGER, PARAMETER :: nb = 5
INTEGER :: i, myrank, nbproc, lbloc, ierr = 0
REAL(8), DIMENSION(nb) :: ain, aout
...
IF (myrank == 0) THEN
   ain(1:nb) = (/ (1000.0_8 + REAL(i,8), i=1,nb) /)
ELSE
   ain(1:nb) = (/ (myrank * 10.0_8, i=1,nb) /)
END IF

CALL MPI_REDUCE (ain, aout, nb, MPI_DOUBLE_PRECISION,&
                 MPI_SUM, 0, MPI_COMM_WORLD, ierr)

IF (myrank == 0) &
   WRITE (6,'(A,I2,A,20E11.4)') 'Rang ', myrank, &
                                ' somme = ', aout(1:nb)
...
mpirun -np 4 ./reduce.out
Rang 0 somme = 0.1061E+04 0.1062E+04 0.1063E+04 0.1064E+04 0.1065E+04
```

Opérations préalables sur des données réparties

- `CALL MPI_REDUCE_SCATTER (sendbuf, recvbuf, recvcounts,&`
  `datatype, op, comm, ierr)`

- On effectue une opération distribuée sur toutes les données définies par sendbuf, count, et datatype, avec
  count $= \sum_i recvcounts[i]$

- Ensuite on partage les résultats en bloc selon recvcounts
  Le i-ème bloc est envoyé au i-ème processus et stocké dans le buffer défini par recvbuf, recvcounts[i], et datatype

Opérations préalables sur des données réparties

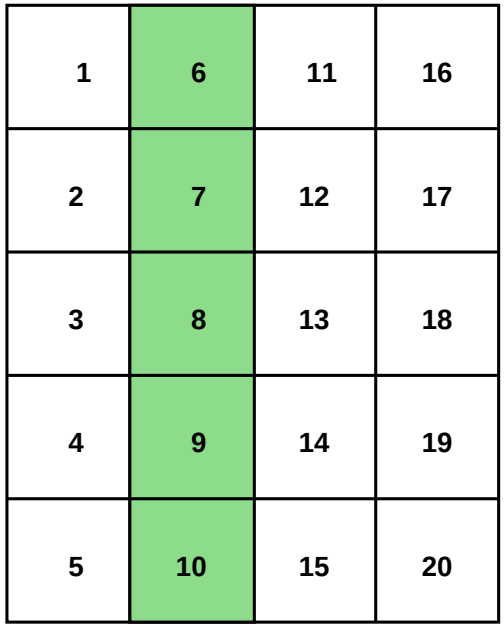
- ```
  CALL MPI_SCAN (sendbuf, recvbuf, count,&
                 datatype, op, comm, ierr)
  ```

- On effectue une opération distribuée partielle sur toutes les données, utilisant les rangs des processus

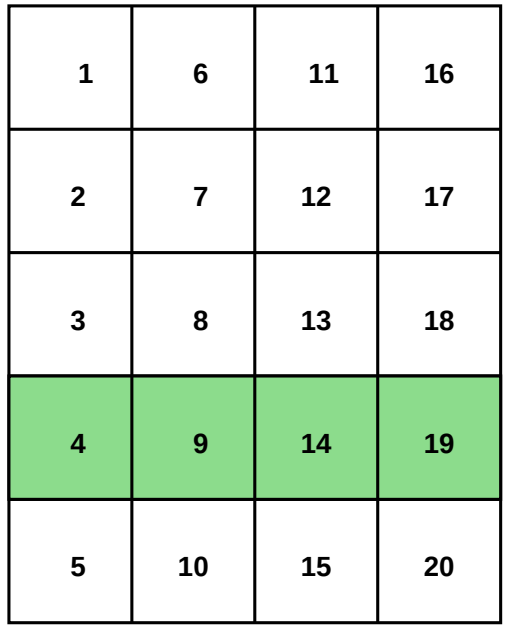
- Le i-ème processus reçoit le résultat de l'opération op effectuée sur toutes les données des processus de rang 0 à i inclus du communicateur comm.

- Les opérations supportées, leur syntaxe, les contraintes sur les buffers d'envoi et de réception sont les mêmes que pour MPI_REDUCE.
  ```

Opérations préalables sur des données réparties

- MPI_REDUCE_SCATTER : quadruplet avec 3 processus, on choisit de mettre deux données au dernier

P0	a0	b0	c0	d0
P1	a1	b1	c1	d1
P2	a2	b2	c2	d2

$\longrightarrow$ MPI_REDUCE_SCATTER $\longrightarrow$

P0	$\sum_i a_i$			
P1	$\sum_i b_i$			
P2	$\sum_i c_i$	$\sum_i d_i$		

- MPI_SCAN : quadruplet avec 3 processus

P0	a0	b0	c0	d0
P1	a1	b1	c1	d1
P2	a2	b2	c2	d2

$\longrightarrow$ MPI_SCAN $\longrightarrow$

P0	a0	b0	c0	d0
P1	a0+a1	b0+b1	c0+c1	d0+d1
P2	a0+a1+a2	b0+b1+b2	c0+c1+c2	d0+d1+d2

Opérations préalables sur des données réparties

- `CALL MPI_ALLREDUCE (sendbuf, recvbuf, count, &`
  `datatype, op, comm, ierr)`

- Depuis tous les processus du communicateur comm, on envoie un message :
  de longueur count, de type datatype,
  à partir de l'adresse sendbuf,
  pour faire l'opération op sur ces valeurs.
  $\Rightarrow$ Réception du résultat à partir de l'adresse recvbuf
  pour tous les processus

- Si le nombre de données envoyées est supérieur à un, l'opération op est appliquée sur chacune des données

- L'opération op est supposée associative

Types de données dérivés

- On peut envoyer autre chose que des buffers contigus de données de même type !

- Types simples : entiers, réels, ...

  `MPI_INTEGER` `MPI_REAL`
  `MPI_DOUBLE_PRECISION` `MPI_COMPLEX`
  `MPI_LOGICAL` `MPI_CHARACTER`

- Types complexes :

  - homogène : toutes les données sont de même type (sections de tableau),
  - hétérogène : les structures en C, les types dérivés en Fortran

- Il faut gérer ces types, au sens MPI :

  création : `MPI_TYPE_xxx`,
  validation : `MPI_TYPE_COMMIT`,
  destruction : `MPI_TYPE_FREE`

Validation - destruction des types crées

- Tout type crée par l'utilisateur doit être validé par un appel à `MPI_TYPE_COMMIT`
- ```
  CALL MPI_TYPE_COMMIT (typename, ierr)
  INTEGER :: typename
  ```
- Cette validation ne concerne que le squelette de la structure, pas l'usage qui en est fait
  ```

Validation - destruction des types crées

- A la fin de l'exécution, tout type crée par l'utilisateur doit être détruit
- ```
  CALL MPI_TYPE_FREE (typename, ierr)
  INTEGER :: typename
  ```
- Cela libère les ressources mémoire occupées par ce type crée
- Cette libération n'a aucun effet sur d'autres types crées à partir de celui qui est ainsi détruit :
  $\Rightarrow$ Un type A crée pour fabriquer un type B peut être détruit à l'issue de la création du type B
  ```

Types homogènes : valeurs contigües

- `CALL MPI_TYPE_CONTIGUOUS (count, oldtype, newtype, ierr)`
- Construction du type newtype, avec count éléments du type oldtype

Types homogènes : valeurs contigües



- `CALL MPI_TYPE_CONTIGUOUS (5, MPI_INTEGER, colonne, ierr)`

```
CALL MPI_TYPE_COMMIT (colonne, ierr)
INTEGER :: colonne
```

Types homogènes : valeurs distantes d'un pas constant

- ```
  CALL MPI_TYPE_VECTOR (count, blocklength, stride, &
                        oldtype, newtype, ierr)
  ```

- Construction du type newtype, avec count blocs, formés chacun de blocklength éléments du type oldtype et espacés d'un pas stride (exprimé en nombre d'éléments)
  ```

Types homogènes : valeurs distantes d'un pas constant



- `CALL MPI_TYPE_VECTOR (4, 1, 5, MPI_INTEGER, ligne, ierr)`

```
CALL MPI_TYPE_COMMIT (ligne, ierr)
INTEGER :: ligne
```

Types homogènes : valeurs distantes d'un pas constant

```
CALL MPI_TYPE_VECTOR (3, 2, 5, MPI_INTEGER, type_bloc, ierr)

CALL MPI_TYPE_COMMIT (type_bloc, ierr)
INTEGER :: type_bloc

 CALL MPI_SEND   (a(3,2), 1, type_bloc, 1,
                  MPI_COMM_WORLD, ierr)

 CALL MPI_SEND   (a(3:4,2:4), 6, MPI_INTEGER,
                  1, MPI_COMM_WORLD, ierr)
```

1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19
5	10	15	20

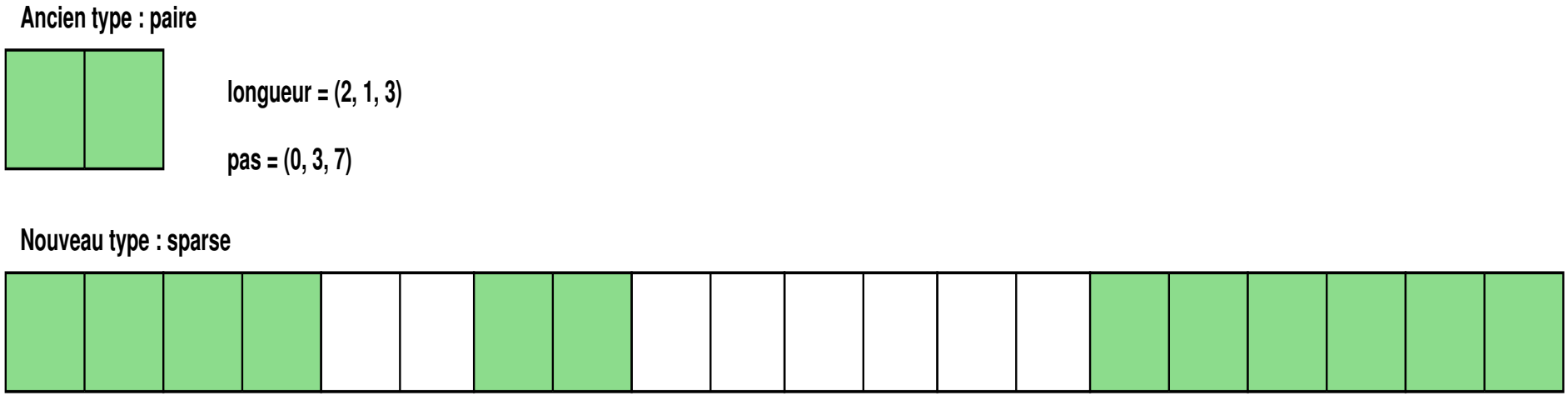
Ici c'est analogue mais faire attention aux communications non bloquantes

Types homogènes : valeurs distantes d'un pas variable

- ```
  CALL MPI_TYPE_INDEXED (count, blocklengths, &
                         displs, oldtype, newtype, ierr)
  ```

- Construction du type newtype, avec count blocs, formés de blocklengths[i] éléments du type oldtype et décalés chacun d'un pas displs[i]

- L'argument blocklengths contient les longueurs des blocs exprimées en nombre d'éléments de l'ancien type oldtype

- L'argument displs contient les distances à l'origine pour chaque bloc, exprimées en nombre d'éléments de l'ancien type oldtype
  ```

Types homogènes : valeurs distantes d'un pas variable



- `CALL MPI_TYPE_INDEXED (3, longueur, pas, paire, sparse, ierr)`

- `CALL MPI_TYPE_COMMIT (sparse, ierr)`
  `INTEGER :: sparse`

Types homogènes : sections de tableau

- Extraire une portion de tableau pour l'envoyer est utile notamment en décomposition de domaine.
- La norme MPI 2.0 fournit une nouvelle fonction qui s'inspire du Fortran 90.
- Le sous-programme `MPI_TYPE_CREATE_SUBARRAY` permet de créer un sous-tableau à partir d'un tableau.
-
```
CALL MPI_TYPE_CREATE_SUBARRAY (rank, shape_tab, shape_subtab, &
                               coord_debut, ordre, oldtype, newtype, ierr)
```
- Construction du type newtype, de rang rank, de profil shape_subtab, extrait à partir de l'élément en coord_debut selon le rangement mémoire ordre.

Types homogènes : sections de tableau, rappels Fortran90

- Un tableau est un ensemble d'éléments du même type :

```
REAL, DIMENSION(-1:3,2,0:5) :: tab
```

- Le rang (`rank`) d'un tableau est son nombre de dimensions
- L'étendue (`extent`) est le nombre d'éléments dans une dimension d'un tableau
- Le profil (`shape`) d'un tableau est un vecteur dont chaque élément est l'étendue de la dimension correspondante
- La taille (`size`) d'un tableau est le produit des éléments du vecteur correspondant à son profil
- Deux tableaux sont dits conformants s'ils ont même profil

Types homogènes : sections de tableau

On crée le type bloc :

```
rank = 2
shape_tab = (/ nb_lignes, nb_colonnes /)
shape_subtab = (/ 2, 3 /)
coord_debut = (/ 2, 1 /)
CALL MPI_TYPE_CREATE_SUBARRAY (rank, shape_tab, shape_subtab, &
                coord_debut, mpi_order_fortran, mpi_integer, block, ierr)
CALL MPI_TYPE_COMMIT (block, ierr)
```

Le processus 0 envoie les données au 1 :

```
CALL MPI_SEND   (a, 1, block, 1, 99,
                 MPI_COMM_WORLD, ierr)

CALL MPI_RECV   (b, 1, block, 0, 99,
                 MPI_COMM_WORLD, status, ierr)
```

1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19
5	10	15	20

Types hétérogènes

- `CALL MPI_TYPE_STRUCT (count, blocklengths, &`
  `displs, oldtype, newtype, ierr)`

- Construction du type newtype, avec count blocs, formés chacun de blocklengths[i] éléments de type oldtype[i] et décalés chacun d'un pas displs[i]

- L'argument blocklengths contient les longueurs des blocs exprimées en nombre d'éléments

- L'argument displs contient les distances à l'origine pour chaque bloc exprimées en nombre d'octets

- L'argument oldtypes contient les différents types de données, un pour chaque bloc

Types hétérogènes

- C'est le constructeur de type le plus général
- Il étend `MPI_TYPE_INDEXED` aux cas où chaque bloc de données, est de type quelconque
- Compte tenu de l'hétérogénéité des données des blocs, les décalages se calculent en prenant les différences des adresses mémoires des éléments avec `MPI_ADDRESS` :
  - on stocke les adresses mémoire des champs du type,
  - on calcule les décalages entre chaque adresse et l'adresse du premier champ

  $\Rightarrow$ `CALL MPI_ADDRESS (location, adresse, ierr)`
- Le calcul des adresses mémoire dépend de l'implémentation !

Types hétérogènes

Construction d'un type particule avec les coordonnées, la masse et l'espèce chimique ...

```
TYPE particule
   REAL :: masse
   INTEGER :: espece
   REAL, DIMENSION(3) :: coords
END TYPE particule

INTEGER, DIMENSION(3) :: oldtypes, blocklengths
INTEGER, DIMENSION(3) :: displs, adrs

INTEGER :: type_p

TYPE (particule) :: p
```

Types hétérogènes

```
oldtypes = (/ MPI_REAL, MPI_INTEGER, MPI_REAL /)

blocklengths = (/ 1, 1, 3 /)

CALL MPI_ADDRESS (p%masse , adrs(1), ierr)
CALL MPI_ADDRESS (p%espece, adrs(2), ierr)
CALL MPI_ADDRESS (p%coords, adrs(3), ierr)

DO i = 1, 3
   displs(i) = adrs(i) - adrs(1)
END DO

CALL MPI_TYPE_STRUCT (3, blocklengths, displs, &
                      oldtypes, type_p, ierr)

CALL MPI_TYPE_COMMIT (type_p, ierr)
```

Communicateurs

- Le communicateur contient tout ce qui est nécessaire pour fournir le cadre adapté aux opérations de communication dans MPI

- MPI fournit un communicateur par défaut, `MPI_COMM_WORLD`, qui contient tous les processus de l'application

- On peut créer des communicateurs de deux manières :
  - directement à partir d'un autre communicateur ;
  - à partir des groupes (au sens MPI) (non considéré ici) ;

- Il y a deux sortes de communicateurs :
  - les intra-communicateurs : pour les opérations sur un groupe de processus au sein d'un communicateur ;
  - les inter-communicateurs : pour les communications entre deux groupes de processus

Communicateurs issus d'un autre communicateur

Cette technique permet de :

- partitionner en une seule fois un communicateur ;
- donner le même nom à tous les communicateurs crées ;
- ne pas manipuler des groupes.
- Il n'y a pas de recouvrement possible
  $\Rightarrow$ En un appel, chaque processus n'appartient qu'à un seul communicateur
- Les fonctions de gestion des communicateurs sont :
  `MPI_COMM_SIZE` : le nombre de processus du communicateur
  `MPI_COMM_RANK` : le rang au sein du communicateur
  `MPI_COMM_SPLIT` : le partitionnement d'un communicateur
  `MPI_COMM_FREE` : la destruction du communicateur
  ...

Création d'un communicateur à partir d'un autre communicateur

- La routine MPI_COMM_SPLIT permet de partitionner un communicateur :

```
CALL MPI_COMM_SPLIT (comm, color, key, new_comm, ierr)
INTEGER :: color, key, new_comm
```

- Tous les processus du communicateur doivent l'exécuter

- L'argument color (non négatif) permet de distinguer les différents communicateurs crées :
  $\Rightarrow$ Tous les processus avec la même valeur appartiendront au même nouveau communicateur

Création d'un communicateur à partir d'un autre communicateur

- L'argument key permet d'ordonner (définir le rang) des processus au sein de leur nouveau communicateur
- Mettre key à zéro comme valeur pour le processus qui sera celui de futur rang 0 (dans le nouveau communicateur) et des valeurs strictement positives pour les autres
- Un processus qui ne fera partie d'aucun des nouveaux communicateurs affecte la valeur `MPI_UNDEFINED` à l'argument color
  $\Rightarrow$ La valeur en sortie de new_comm sera alors `MPI_COMM_NULL`

Communicateur pair / communicateur impair

```
INTEGER :: myrank, ierr = 0
INTEGER :: color, key
CALL MPI_COMM_RANK (MPI_COMM_WORLD, myrank, ierr)

IF ( MOD(myrank,2) == 0 ) THEN
   color = 0
   IF (myrank == 4) THEN
      key = 0
   ELSE
      key = myrank + 10
   END IF
ELSE
   color = 1
   IF (myrank == 1) THEN
      key = 0
   ELSE
      key = myrank + 10
   END IF
END IF

CALL MPI_COMM_SPLIT (comm, color, key, new_comm, ierr)
```

Topologies

- Topologie = attribut supplémentaire d'un communicateur pour mieux "coller" à la réalité
- Numéroter les processus de 0 à (n-1) n'est pas toujours la meilleure logique de communication pour l'application parallèle
- Grilles 2D, 3D de processus souvent mieux adaptées
- Cela a aussi une action sur la distribution des processus MPI sur les processeurs sous-jacents grâce à une renumérotation possible des processus
  ... et donc aussi des conséquences sur les performances
- Topologies cartésiennes pour les grilles de processus, graphes pour les géométries plus complexes

Répartition des processus par dimension

- Nombre de processus par dimension : `MPI_DIMS_CREATE`
- `CALL MPI_DIMS_CREATE (nbproc, nbdims, dims, ierr)`
- L'argument nbdims : le nombre de dimensions de la grille
  `INTEGER :: nbdims`
- L'argument dims : en sortie, le nombre de processus par dimension
  `INTEGER, DIMENSION(1:nbdims) :: dims`
- L'argument dims peut être "pré-rempli" pour imposer le nombre de processus dans une ou plusieurs dimensions :
  $\Rightarrow$ Attention, dans ce cas il faut que $\prod_{i, dims(i) \neq 0} \text{dims}(i) \mid \text{nbproc}$

Répartition des processus par dimension

Exemples :

`dims` avant l'appel	Appel à la fonction Appel à la fonction	`dims` en sortie
(0,0)	`MPI_DIMS_CREATE` (6, 2, dims, ierr)	(3,2)
(0,0)	`MPI_DIMS_CREATE` (7, 2, dims, ierr)	(7,1)
(0,3,0)	`MPI_DIMS_CREATE` (6, 3, dims, ierr)	(2,3,1)
(0,3,0)	`MPI_DIMS_CREATE` (7, 3, dims, ierr)	erreur !

3 n'est pas un diviseur de 7 ....

Topologie cartésienne

- Création de la grille cartésienne des processus : MPI_CART_CREATE
  $\Rightarrow$ création d'un nouveau communicateur
- ```
  CALL MPI_CART_CREATE (MPI_COMM_WORLD, nbdims, &
                        dims, period, reorg, newcomm, ierr)
  ```
- L'argument nbdims : le nombre de dimensions de la grille
  ```
  INTEGER :: nbdims
  ```
- L'argument dims : le nombre de processus par dimension
  ```
  INTEGER, DIMENSION(1:nbdims) :: dims
  ```
- L'argument period : la périodicité dans chaque dimension
  ```
  LOGICAL, DIMENSION(1:nbdims) :: period
  ```
- L'argument reorg : réordonner les processus ?
  ```
  LOGICAL :: reorg
  ```
- L'argument newcomm : en sortie, le nouveau communicateur attaché à la topologie
  ```
  INTEGER :: newcomm
  ```
  ```

Topologie cartésienne : exemple

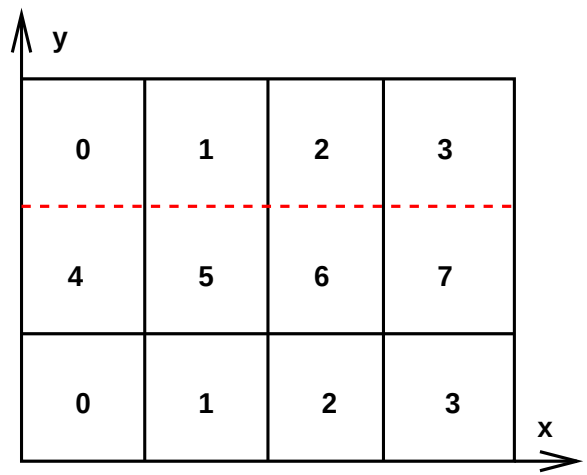
- Topologie cartésienne 2D avec 8 processus, périodicité en y
- `CALL MPI_CART_CREATE (MPI_COMM_WORLD, nbdim, &`
  `dims, period, reorg, comm2d, ierr)`

nbdims = 2

dims = (/4, 2/)

period = (/.FALSE., .TRUE./)

reorg = .TRUE.

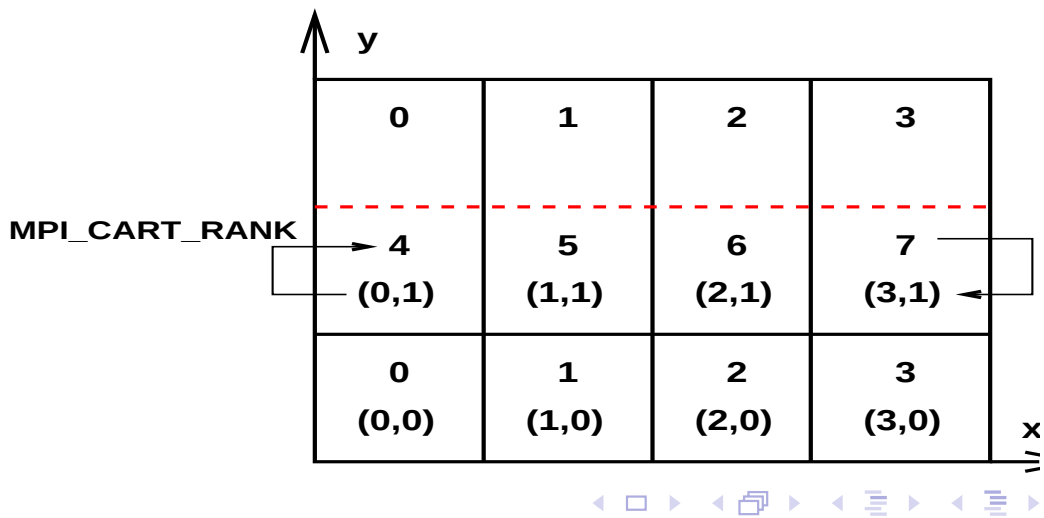


Recherche des coordonnées

- Recherche des coordonnées, connaissant le rang dans le communicateur de la grille :
- `CALL MPI_CART_COORDS (comm, rank, nbdims, coords, ierr)`
- L'argument coords est un tableau d'étendue le nombre de dimensions de la grille (nbdims)

  `INTEGER, DIMENSION(1:nbdims) :: coords`
- Recherche du rang connaissant les coordonnées dans le communicateur de la grille :
- `CALL MPI_CART_RANK (comm, coords, rank, ierr)`

  `INTEGER :: rank`



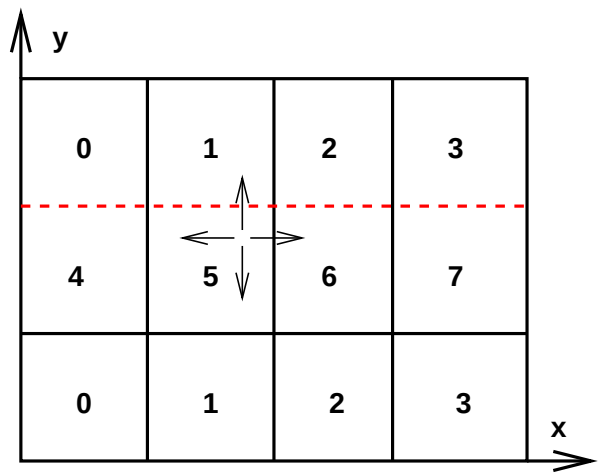
Recherche des voisins

- Recherche des voisins dans chaque dimension :
- `CALL MPI_CART_SHIFT (comm, direction, disp, &`
  `rank_left, rank_right, ierr)`
- L'argument direction est la dimension concernée,
  $\Rightarrow$ i.e. la coordonnée modifiée par le disp
  `INTEGER :: direction`
- L'argument disp indique le déplacement
  `INTEGER :: disp`
- Les arguments rank_left et rank_right sont les rangs des voisins
  `INTEGER :: rank_left, rank_right`
- En cas de "sortie" de la dimension, la valeur retournée est
  `MPI_PROC_NULL`

Recherche des voisins

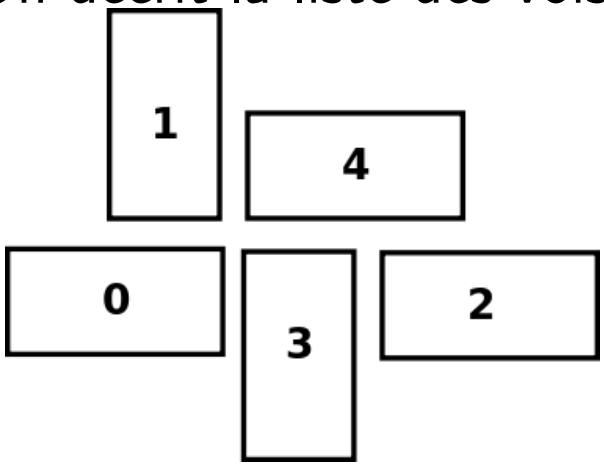
- Recherche des voisins direct Gauche / Droite (ou Ouest / Est) par le processus 5
- `CALL MPI_CART_SHIFT (comm2d, direction, disp, &`
  `rank_left, rank_right, ierr)`

direction = 0
disp = 1
⇒ rank_left = 4
⇒ rank_right = 6



Graphe de processus

- La décomposition de domaine mène parfois à un voisinage irrégulier
- On décrit la liste des voisins à l'aide de deux vecteurs



processus	voisins
0	1, 3
1	0, 4
2	3, 4
3	0, 2, 4
4	1, 2, 3

```
index=(/        2,   4,   6,     9,   12/)
mes_vois=(/1,3, 0,4, 3,4, 0,2,4, 1,2,3/)
```

index(i)-index(i-1) : nbre de voisins du proc i
mes_vois(j), pour index(i)+1 $\leq$ j $\leq$ index(i+1) : liste des voisins de i
N.B. : index(1) : nbre de voisins de 0 ;
mes_vois(j), pour 1 $\leq$ j $\leq$ index(1) : liste des voisins de 0

Graphe de processus

- Demander la création d'une topologie en graphe :

```
CALL MPI_GRAPH_CREATE (comm, nbproc, index, ivois, reorg,
                       comm_graphe, ierr)
```

- Obtenir le nombre de voisins pour un processus donné :

```
CALL MPI_GRAPH_NEIGHBORS_COUNT (comm_graphe, myrank, nb_vois, ierr)
```

- Obtenir la liste des voisins pour un processus donné :

```
CALL MPI_GRAPH_NEIGHBORS (comm_graphe, myrank, nb_vois,
                          mes_vois, ierr)
```

- Obtenir le nombre de nœuds du graphe :

```
CALL MPI_GRAPHDIMS_GET (comm_graphe, nb_noeuds, nb_aretes, ierr)
```

- Obtenir le nombre d'arêtes du graphe :

```
CALL MPI_GRAPH_GET (comm_graphe, nb_max_noeuds, nb_max_vois,
                    index, vois, ierr)
```

Entrées / sorties collectives : MPI I/O

- Un des gros apports de MPI 2
- Interface de haut niveau
- Opérations collectives pour les opérations courantes
- Nombreuses fonctionnalités
- On peut en parler pendant des heures ...

Création d'un fichier

```
PROGRAM open01
USE mpi
IMPLICIT NONE
INTEGER :: descripteur, ierr
CALL MPI_INIT (ierr)
CALL MPI_FILE_OPEN (MPI_COMM_WORLD, "fichier.txt", &
                    MPI_MODE_RDWR + MPI_MODE_CREATE, &
                    MPI_INFO_NULL, descripteur, ierr)
CALL MPI_FILE_CLOSE (descripteur, ierr)
CALL MPI_FINALIZE (ierr)
END PROGRAM open01
```

- les attributs s'additionnent en Fortran ou se combinent ("&") en C/C++
- On peut fournir d'autres informations (selon implémentation !) sinon, une valeur nulle, MPI_INFO_NULL

Généralités

- Les transferts de données entre fichiers et zones mémoire des processus se font via des appels explicites à des sous-programmes MPI de lecture et d'écriture.

- On distingue 3 propriétés des accès aux fichiers :
  - le `positionnement`, qui peut être **explicite** ou **implicite** (via des pointeurs, **individuels** ou **partagés** par tous les processus) ;
  - la `synchronisation`, les accès pouvant être bloquants ou non ;
  - le `regroupement`, les accès pouvant être collectifs (tous les processus ayant ouvert le fichier) ou propres à un ou plusieurs processus.

- Il y a de nombreuses variantes possibles ...

Compléments accès

- Il est possible de mélanger les types d'accès effectués à un même fichier au sein d'une application.

- Les zones mémoire accédées sont décrites par trois quantités :
  - l'adresse initiale de la zone concernée ;
  - le nombre d'éléments pris en compte ;
  - le type de données, qui doit correspondre à une suite de copies contiguë du type élémentaire de donnée (`etype`) de la `vue` courante.

- Tous les processus d'un communicateur au sein duquel un fichier est ouvert participeront aux opérations collectives ultérieures d'accès aux données.

Définition des vues

- Les `vues` sont un mécanisme souple et puissant pour décrire les zones accédées dans les fichiers.

- Les `vues` sont construites à l'aide de **types dérivés** MPI.

- Chaque processus a sa propre `vue` (ou ses propres `vues`) d'un fichier, définie par trois variables :
  - un **déplacement initial**,
  - un **type élémentaire de données**,
  - un **motif**.

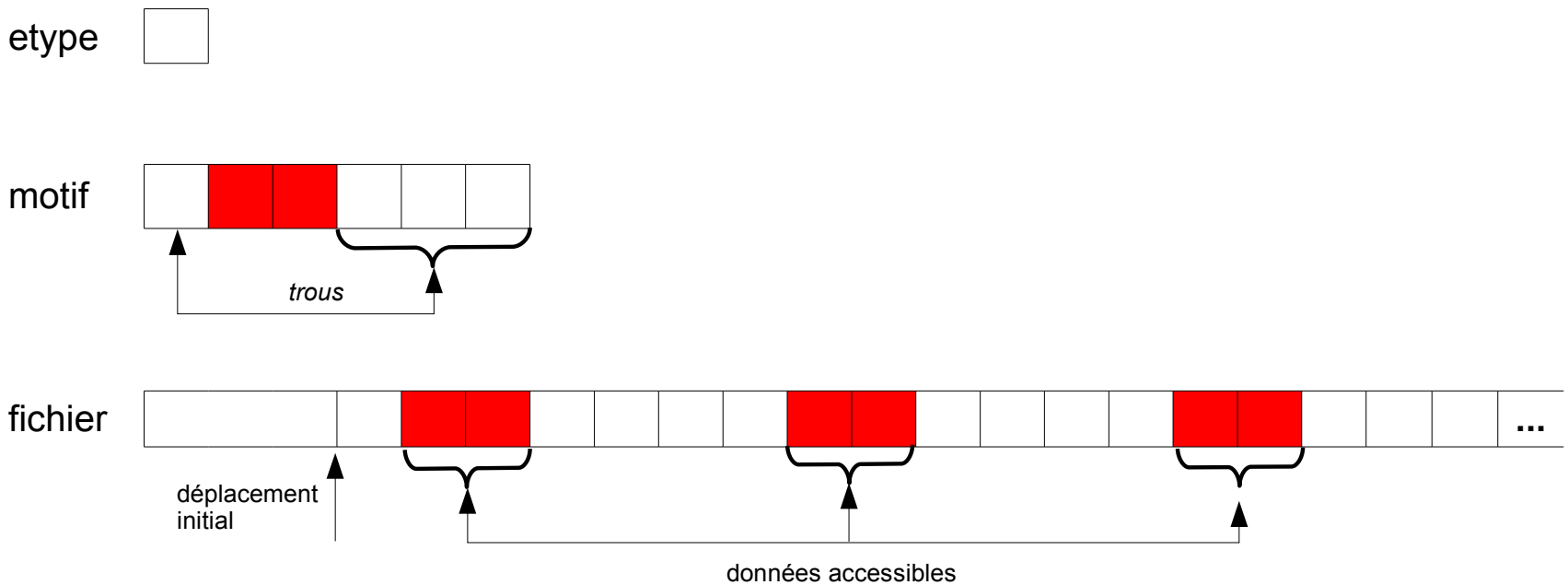
  Une `vue` est définie comme la répétition du motif, une fois le déplacement initial effectué.

- Des processus différents peuvent avoir des **vues différentes** d'un même fichier, de façon à accéder à des parties différentes.

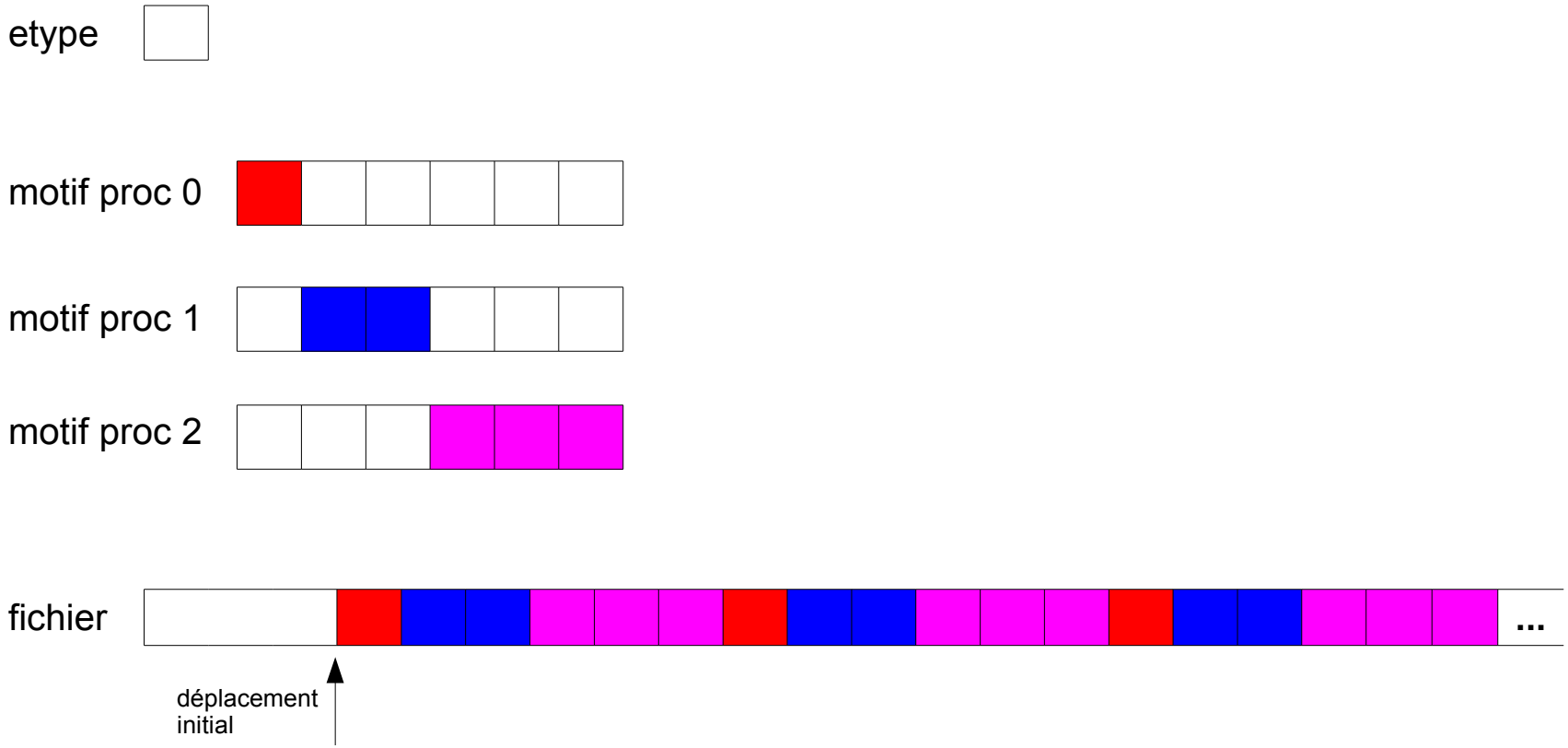
Définition de vues

- Si le fichier est ouvert en écriture, les zones décrites par les types élémentaires et les motifs ne peuvent pas se recouvrir, même partiellement.

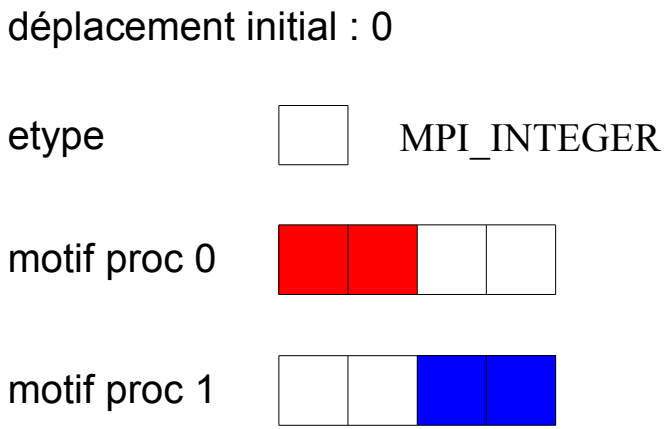
- La routine `MPI_FILE_SET_VIEW` permet de construire la vue en fournissant un `motif`



Définition de motifs différents selon les processus



Utilisation d'une vue : exemple

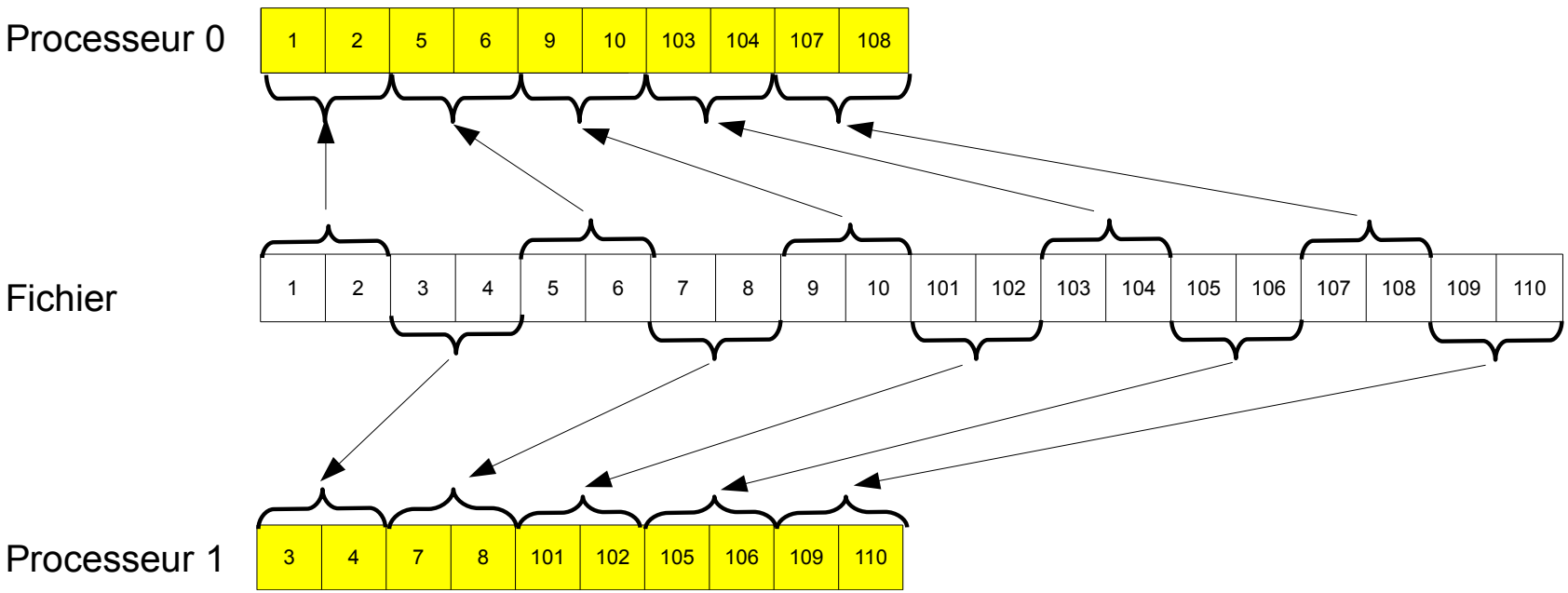



```
PROGRAM read_view02
USE mpi
IMPLICIT NONE

INTEGER, PARAMETER :: nb_valeurs = 10
INTEGER :: myrank, descripteur, coord, motif, ierr
INTEGER(KIND=MPI_OFFSET_KIND) :: deplacement_initial
INTEGER, DIMENSION(nb_valeurs) :: valeurs
INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status

CALL MPI_INIT (ierr)
CALL MPI_COMM_RANK (MPI_COMM_WORLD, myrank, ierr)
```

Utilisation d'une vue : exemple (suite)

```
CALL MPI_FILE_OPEN (MPI_COMM_WORLD, "donnees.dat", MPI_MODE_RDONLY, &
                    MPI_INFO_NULL, descripteur, ierr)
IF (myrank == 0) THEN
   coord = 1
ELSE
   coord = 3
END IF
CALL MPI_TYPE_CREATE_SUBARRAY (1, (/4/), (/2/), (/coord-1/), &
                               MPI_ORDER_FORTRAN, MPI_INTEGER, &
                               motif, ierr)

CALL MPI_TYPE_COMMIT (motif, ierr)

deplacement_initial = 0_MPI_OFFSET_KIND
CALL MPI_FILE_SET_VIEW (descripteur, deplacemnt_initial, MPI_INTEGER, &
                        motif, "native", MPI_INFO_NULL, ierr)

CALL MPI_FILE_READ (descripteur, valeurs, nb_valeurs, MPI_INTEGER, status, ierr)

CALL MPI_FILE_CLOSE (descripteur, ierr)
CALL MPI_FINALIZE (ierr)
END PROGRAM read_view02
```

Utilisation d'une vue : exemple (fin)



Conclusions

- Le passage de messages est une technique de parallélisation portable
- Elle permet de travailler sur tout type d'architecture
- Elle est orientée parallélisme à gros grains
- Elle permet les échanges au sein de n'importe quel groupe de processus
- Elle permet les échanges de n'importe quel type de données