Introduction à MPI

Bibliothèque MPI

- ☑ Quelques rappels sur le parallélisme
- ☑ Qu'est ce que MPI?
- ☑ Pourquoi utiliser MPI?
- ☑ Bibliothèque MPI
  - ✓ Routines de base
  - ✓ Mode et type de communications
  - ✓ Complétion
- ☑ Programme MPI

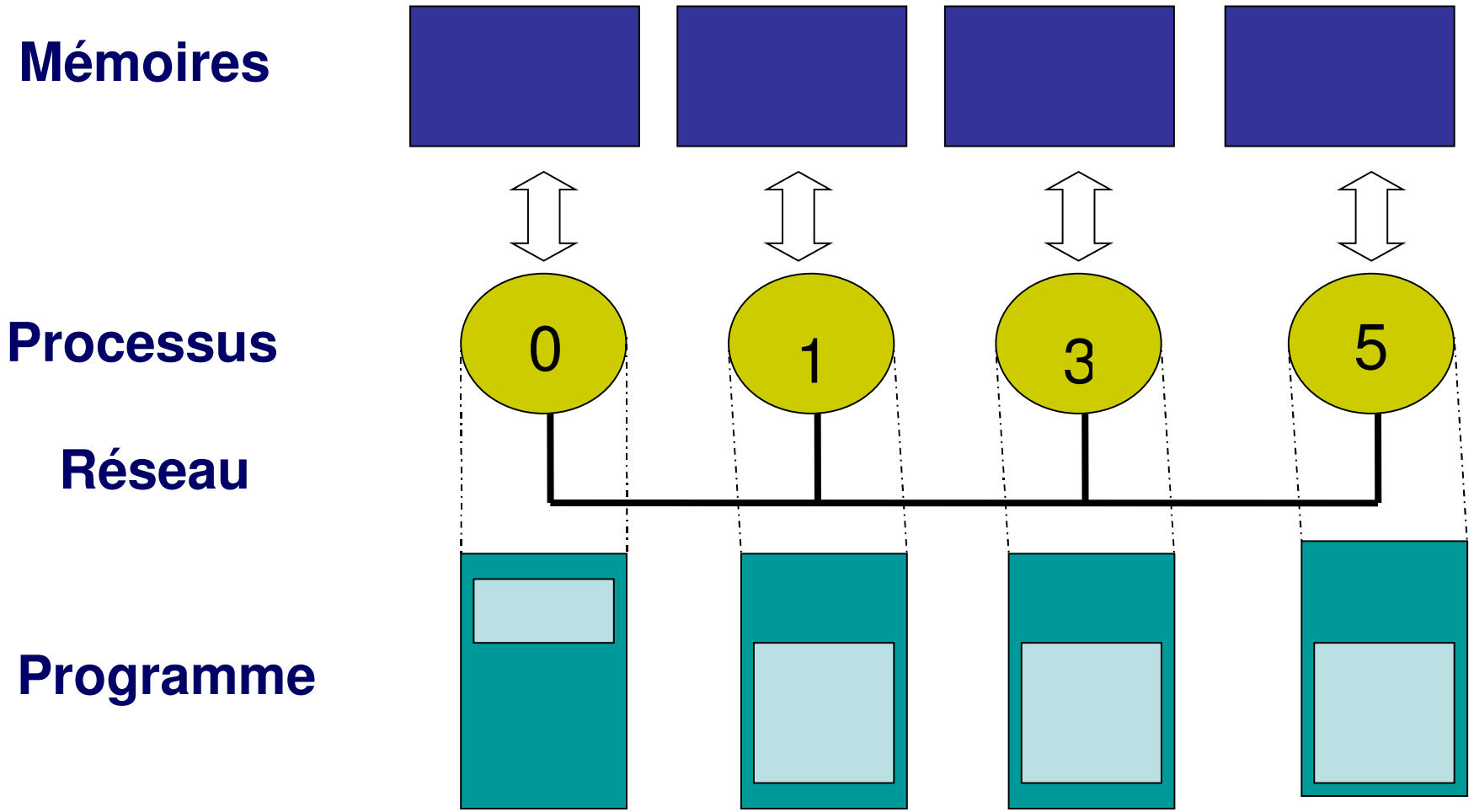
Quelques rappels

- **Architectures** parallèles
- Modèle de programmation **séquentiel**
- Modèle de programmation par **échange de messages**
- Modèle d'exécution **SPMD, MPMD**
- **Objectifs** de la programmation parallèle

Modèle par échange de messages
Principes

- **Plusieurs processus** travaillant sur des **données locales.**
- Chaque processus a ses propres variables et il n'a pas accès directement aux variables des autres processus
- **Le partage des données** entre processus se fait **par envoi et réception explicites de messages**
- Les processus peuvent s'exécuter sur des processeurs différents ou identiques.

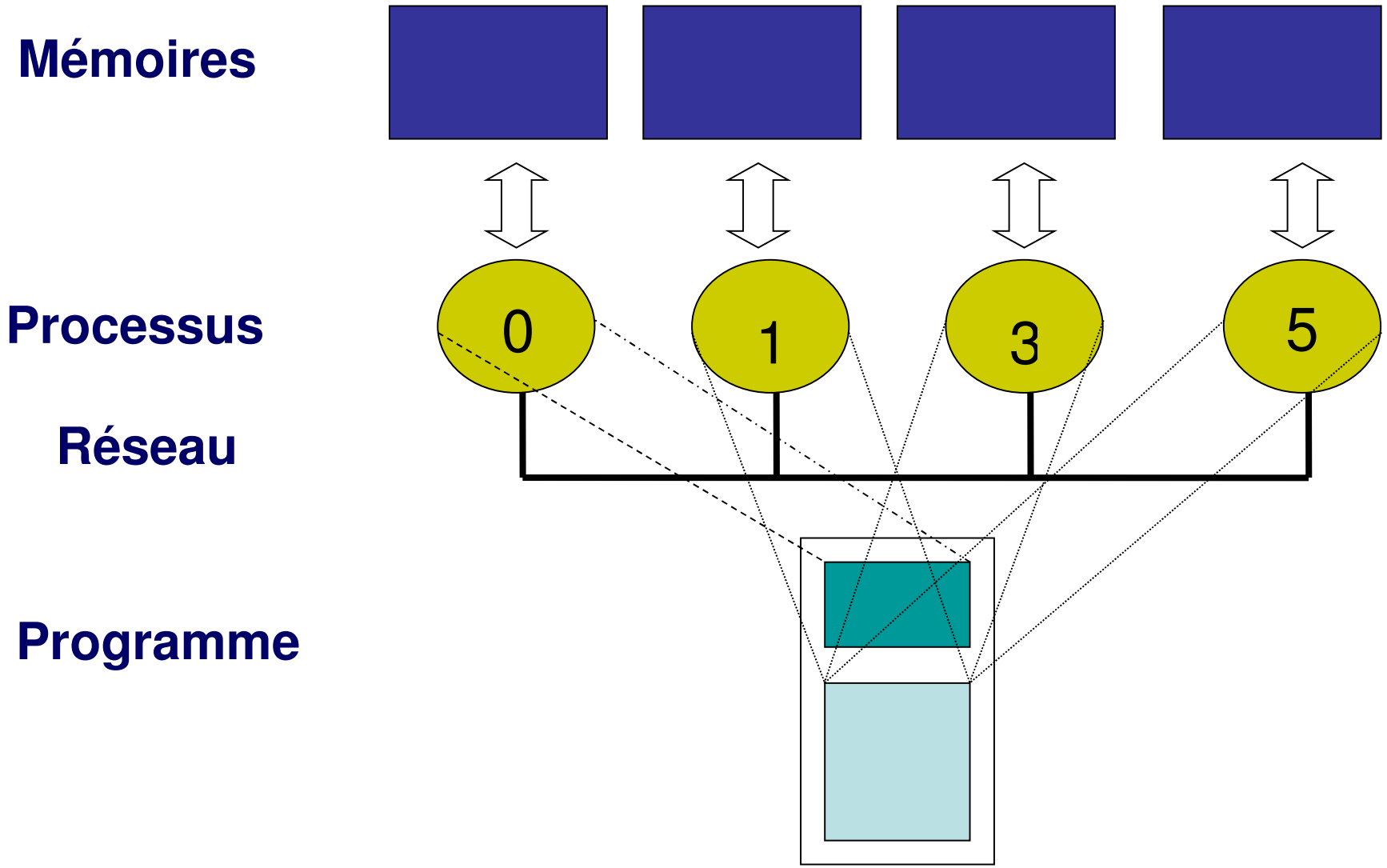
Modèle à échange de messages (MPMD)



Modèle par échange de messages
SPMD

- **S**ingle **P**rogram **M**ultiple **D**ata
- **Le même programme** s'exécute pour tous les processus
- **Toutes les machines** supportent ce modèle de programmation
- Cas particulier du **modèle MPMD**

Modèle à échange de messages (SPMD)



Modèle par échange de messages Avantages

- Peut être implémenter sur **une grande variété de plates-formes**
  - Calculateur à mémoire distribuée
  - Calculateur à mémoire partagée
  - Réseau de stations mono ou multi-processeurs
  - Station mono-processeur
- Permet en général **un plus grand contrôle de la localisation des données** et par conséquent une meilleure performance des accès aux données.

Objectifs de la programmation parallèle

- Obtenir de **meilleures performances** que la version séquentielle
- Traiter de façon performante **un volume de données plus important**
- **Quelques critères de performance**:
  - Équilibrer les charges sur chaque processus
  - Minimiser les communications
  - Recouvrir les communications par des calculs (attention à la synchronisation…)

Facteurs de performances communications

- Elles dépendent de deux types de critères:
  - **Le mode de transfert**
    - Avec **recopie** ou **sans recopie** des données transferrées
    - **Bloquant** et **non bloquant**
    - **Synchrone** et **non synchrone**
  - **Les performances du réseau**
    - **Latence**: temps mis pour envoyer un message de taille nulle
    - **Débit** ou **bande passante**

➔le choix d'un mode de transfert adapté et le recouvrement des communications par les calculs sont les deux voies à privilégier

Qu'est ce que MPI ?

- **Bibliothèque standard** de l'implémentation du modèle d'échange de messages
  - **MPI-1** (version définie en 1994)
    - Portabilité
    - Implémentation (souvent optimisée) laissée à l'initiative des constructeur
    - Valide sur la plupart des plates-formes
  - **MPI-2:** MPI_IO, process management, interface avec C++ et f90 …
- **Objectifs de MPI**
- Ce que ne fait pas partie de MPI

Apport de MPI-2 à MPI-1

- **Gestion dynamique des processus**
  - Possibilité de développer des codes MPMD
  - Support multi plates-formes
  - Démarrage et arrêt dynamique des sous-tâches
  - Gestion des signaux système
- **E/S parallèles**
- **Communication de mémoire à mémoire (put/get)**
  - **accès direct à la mémoire d'un processus distant (Remote Memory Access)**

Pourquoi utiliser (ou ne pas) MPI

- **Utiliser**:
  - Écrire un code parallèle portable
  - Écrire une application parallèle performante (localisation des données, bibliothèque //…)
  - Traiter un problème irrégulier qui ne peut être traité avec un modèle « data-parallel »
- **Ne pas l'utiliser**:
  - on obtient des performances avec un modèle « data-parallel » (OpenMp) => plus simple
  - On peut utiliser une bibliothèque parallèle existante
  - Pas besoin de parallélisme (type de problème, taille,….)

Bibliothèque MPI: Bibliographie

- **Les spécifications de la norme MPI:**

  ftp://ftp.irisa.fr/pub/netlib/mpi/drafts/draft_final.ps

- **Quelques ouvrages:**
  - **MPI :** The complete reference
    M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra
  - **Using MPI**: Portable Parallel Programming with the Message-Passing Interface (second edition)
    W. Gropp, Ewing Lusk and Anthony Skjellum
  - **Using MPI-2:** Advanced Features of the Message-Passing Interface
    W. Gropp, Ewing Lusk and Anthony Skjellum
  - Parallel Programming with MPI : Peter S. Pacheco

Bibliothèque MPI: Bibliographie

- **Documentations complémentaires** et descriptions de différentes implémentations

  http://www.mcs.anl.gov/mpi

  http://www.mpi-forum.org/index.html

- **Formations:**
  - **Tutoriaux:**

    www-unix.mcs.anl.gov/mpi/tutorial/gropp/talk.html
  - **Cours:**

    www.idris.fr/data/cours/cours-IDRIS.html

    foxtrot.ncsa.uiuc.edu:8900/webct/public/home.pl

Routines de base de MPI

Plusieurs types de routines dans MPI:

- **Gestion des communications**
  - Initialisation et fin des communications
  - Création de groupe, de communicateur et de topologie virtuelle
- Routines de **communications entre deux processus**
- Routines de **communications collectives**
- Création de **types dérivés**

Modes et types de communication
Critères de complétion

- **Modes de communications**
  - Standard
  - Synchrone
  - Bufferisé
  - « Ready »
- **Communication**
  - Bloquantes
  - Non-bloquantes

Programme MPI

- ☑ Fichier entête
- ☑ Convention des noms dans MPI
- ☑ Codes définis et utilisés par MPI
- ☑ Type de données MPI
- ☑ Communicateurs
- ☑ Informations sur un communicateur
- ☑ Exemple de programme MPI

Structure d'un programme MPI

include « fichier entête MPI »

- En C: **#include <mpi.h>**
- En fortran : **include 'mpif.h'**

Déclarations des variables

Initialisation de MPI (par chaque processus)

…

Calculs, communications et impressions

…

Fermeture de MPI (par chaque processus)

Conventions sur les noms dans MPI

- **Sous programmes**:
  - **Fonctions en C**: MPI_Xxxxx(parametres,…)

    ierr=MPI_Init(&argc,&argv)
  - **Routines en Fortran**: MPI_XXXXX(parametres,…)

    call MPI_INIT(IERR)
- **Constantes MPI**
  - En majuscules en C et en Fortran: MPI_COMM_WORLD, MPI_REAL,…
  - En C, des types sont prédéfinis
    - Type des communicateurs: MPI_Comm
    - …

Types et « handles » MPI

- **MPI handles**: référence aux structures de données MPI
  - En C: pointeur sur des objets de type MPI prédéfini
  - En Fortran: entier ou tableau d'entiers
    - **Exemple: MPI_COMM_WORLD**: entier(fortran), objet de type MPI_Comm en C
- **Types de données MPI** :

  Masque l'hétérogénéité de l'implémentation des types
  - Variables déclarées en C et Fortran
  - Utilisation des types MPI correspondants dans les appels aux routines MPI
- MPI permet de définir **ses propres types de données**

Communicateurs

- **Un communicateur** est un code MPI désignant un ensemble de processus succeptibles de communiquer entre eux
- **MPI_COMM_WORLD** est le communicateur par défaut constitué de tous les processus de l'application: il est créé par MPI à l'initialisation de MPI(MPI_INIT)
- Les processus peuvent communiquer uniquement si ils partagent un même communicateur
- Pour **réduire la portée des communications** à un sous-ensemble de processus, on devra **créer un nouveau communicateur** par un appel à une routine MPI
- **Un processus** :
  - Peut appartenir à plusieurs communicateurs
  - Possède un rang dans chaque communicateur

Informations sur le communicateur

- **Taille**: nombre de processus

En C: int MPI_Comm_size(MPI_Comm comm,int *size)

En Fortran:

```
integer :: COMM,size,ier
call MPI_COMM_SIZE(COMM,size,ier)
```

- **Rang d'un processus dans un commutateur**

En C: int MPI_Comm_rank(MPI_Comm comm,int * rang)

En Fortran:

```
integer :: COMM,rang,ier
call MPI_COMM_RANK(COMM, rang,ier)
```

Initialisation MPI

- **Initialisation**
  - **En C**: ierr = MPI_Init(&argc,argv)
  - **En Fortran**: call MPI_INIT(ierr)
- **Informations sur le communicateur**
  - Rang du processus

  En C : int MPI_Comm_rank(MPI_Comm comm,int *rang)

  En Fortran: call MPI_COMM_RANK(comm,rang,ierr)
  - La taille du communicateur

  En C : int MPI_Comm_size(MPI_Comm comm,int *size)

  En Fortran : call MPI_COMM_SIZE(comm,size,ierr)
- **Fin MPI**
  - **En C**: ierr = MPI_Finalize()
  - **En Fortran**: call MPI_FINALIZE(ierr)

Premier programme MPI

```
   program premier_mpi
   include 'mpif.h'
! Declarations
  integer :: monrang,taille,ier

! Initialisation de MPI
  call MPI_INIT(ier)

! Récupère mon rang dans le communicateur par défaut
  call MPI_COMM_RANK(MPI_COMM_WORLD, monrang,ier)

! Récupère le nombre total de processus
  call MPI_COMM_SIZE(MPI_COMM_WORLD,taille,ier)

  print*, 'processeur ',monrang,' parmi ',taille, ' processus'

! Fin de MPI
  call MPI_FINALIZE(ier)

  end program premier_mpi
```

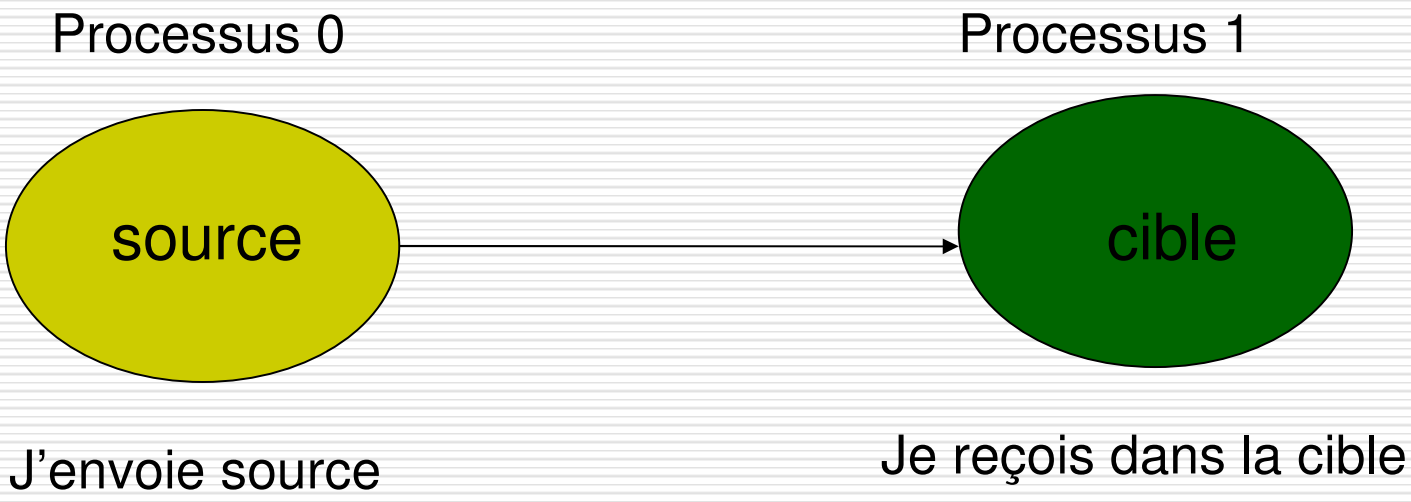
Bibliothèque MPI

Les sous-programmes peuvent être classés dans les catégories suivantes:

- ☑ Environnement
- ☑ Communication point à point
- ☑ Communications collectives
- ☑ Types de données dérivés
- ☑ Topologies
- ☑ Groupe et communicateurs

Concept de l'échange de message

Si un message est envoyé a un processus, celui-ci **doit** le recevoir



Communications

- Communications point à point
  - Mode de communication
  - Synchrone, asynchrone
  - Complétion
  - deadlock
- Communications collectives
  - Synchronisation globale
  - Transfert de données
  - Transfert et opération sur les données

Communications Point à Point

- **Concept**:
  - Communication entre un processus qui envoie (**émetteur**) et un processus qui reçoit (**destinataire**) identifiés par leur rang
  - Message: enveloppe + corps du message + tag
- **SEND et RECEIVE bloquant**
- **SEND et RECEIVE non- bloquant**
- Plusieurs messages peuvent être en attente de réception: **un attribut** (**tag**) permettra de les identifier
- **Plusieurs modes de transfert** faisant appel à des protocols différents

Messages

- **Enveloppe** : 4 parties
  - **Source**: rang du processus émetteur
  - **Destination**: rang du processus destinataire
  - **Communicateur groupe de processus** qui définit le contexte de communication de l'opération
  - **Tag** identifiant le message envoyé
- **Corps du message**: 3 parties
  - **Buffer**: les données du message
  - **Datatype**:le type des données
  - **Count**: nombre de données de type « datatype » dans le buffer

Envoi et réception bloquant

- Deux fonctions: **MPI_SEND** et **MPI_RECV**
- **MPI_SEND**:
  - le processeur émetteur est implicite, le destinataire est donné explicitement dans l'enveloppe
  - Adresse d'un buffer et nombre de données du type indiqué
- **MPI_RECV**:
  - l'enveloppe du message indique explicitement les processus émetteur et destinataire
  - Un « tag » permet au processus destinataire d'identifier le message à recevoir parmi plusieurs messages en attente
  - Le processus destinataire fournit un espace de stockage suffisant pour réceptionner les données

Envoi bloquant: **MPI_SEND**

- **Message:**
  - **Corps**: Adresse du buffer, effectif, type des données
  - **Enveloppe**: destinataire, tag, communicateur
- **Appel Fortran**:

```
MPI_SEND(buf,count,DTYPE, dest,tag,comm,ierr)
```

- **Appel C**

```
int MPI_Send(void *buf,int count,MPI_DATATYPE dtype,int
    dest,int tag, MPI_Comm comm);
```

Réception bloquante: MPI_RECV

- **Message:**
  - **Corps**: buffer,effectif,type de données
  - **Enveloppe**: émetteur,tag,communicateur, recepteur
  - **Status**: tableau contenant des informations sur le message reçu (source, tag, count)
- **Mots-clés**: MPI_ANY_TAG, MPI_ANY_SOURCE
- **Appel Fortran:**

```
MPI_RECV(buf,count,DTYPE,source,tag,comm,status,ierr)
```

- **Appel C:**

```
int MPI_RECV(void *buf,int count,MPI_Datatype dtype ,int source,int tag,MPI_Comm
    comm,MPI_Status *status)
```

Type de données de base - Fortran

TYPE MPI	Type Fortran
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER
MPI_PACKED	Types hétérogènes

Types de données de base: C

Type MPI	Type C
MPI_SHORT	signed short
MPI_INT	signed int
MPI_LONG	signed long int
MPI_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_PACKED	Types hétérogènes

```
program simple_send_receive

   include 'mpif.h'
   integer myrank,ierr,status(MPI_STATUS_SIZE)
   real a(100)
! Initialisation de MPI
   call MPI_INIT(ierr)
! Recupere mon rang
   call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
! Le processus 0 envoie, le processus 1 reçoit
   if (myrank .eq. 0) then
      a=4.
     call MPI_SEND(a,100,MPI_REAL,1,17,MPI_COMM_WORLD,ierr)
   else if (myrank .eq. 1) then
     call MPI_RECV(a,100,MPI_REAL,0,17, MPI_COMM_WORLD,status,ierr)
   endif
! Termine MPI
   call MPI_FINALIZE (ierr)
end program simple_send_receive
```

MPI_SEND - Runtime

- **Deux comportements possibles:**
  - Copie du message dans un buffer MPI
    - Le processus émetteur continue son travail
    - Le message sera transféré lorsque le processus destinataire sera prêt.
  - Le message reste dans les variables du programme
    - Le processus émetteur est bloqué jusqu'à ce que le processus destinataire soit prêt.
- **Ce comportements dépend de**:
  - La taille du message
  - L'implémentation de MPI (laisser MPI faire le choix)

Communication bloquante - complétion

- **Pour MPI_SEND:**
  - Les données envoyées peuvent être modifiées sans compromettre la réception.
    - **Bufferisé**: lorsque la bufferisation est terminée.
    - **Non bufferisé**: lorsque la réception est terminée.
- **MPI_RECV**
  - Le message à recevoir est arrivé et les données sont copiées dans les variables du processus destinataire

DEADLOCK Probable: Dépend de la taille du message

```
program simple_deadlock

   include 'mpif.h'
   integer myrank,ierr,status(MPI_STATUS_SIZE)
   real a(100),b(100)

! Initialisation de MPI
   call MPI_INIT(ierr)

! Recupere mon rang
   call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)

! Le processus 0 envoie, le processus 1 reçoit
   if (myrank .eq. 0) then
      call MPI_SEND(a,100,MPI_REAL,1,17,MPI_COMM_WORLD,ierr)
      call MPI_RECV(b,100,MPI_REAL,1,19,MPI_COMM_WORLD,status,ierr
   else if (myrank .eq. 1) then
       call MPI_SEND(b,100,MPI_REAL,0,19,MPI_COMM_WORLD,ierr
      call MPI_RECV(a,100,MPI_REAL,0,17,MPI_COMM_WORLD,status,ierr)
   endif

! Termine MPI
   call MPI_FINALIZE (ierr)
```

```
program simple_deadlock

   include ‘mpif.h’
   integer myrank,ierr,status(MPI_STATUS_SIZE)
   real a(100),b(100)

! Initialisation de MPI
   call MPI_INIT(ierr)

! Recupere mon rang
   call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)

! Le processus 0 envoie, le processus 1 reçoit
   if (myrank .eq. 0) then
      call MPI_RECV(b,100,MPI_REAL,1,19,MPI_COMM_WORLD,status,ierr)
      call MPI_SEND(a,100,MPI_REAL,1,17,MPI_COMM_WORLD,ierr)
   else if (myrank .eq. 1) then
      call MPI_SEND(b,100,MPI_REAL,0,19,MPI_COMM_WORLD,ierr)
      call MPI_RECV(a,100,MPI_REAL,0,17,MPI_COMM_WORLD,status,ierr)

   endif
…
```

SEND et RECEIVE non-bloquant

- **Deux appels séparés par communication**:
  - Initialisation de la communication
  - Complétion de la communication
- Un identificateur de requête de communication
- **Permet** :
  - d'éviter les deadlocks
  - De superposer les communications et les calculs

L'algorithme doit gérer la synchronisation des processus

- **Les étapes de communication sous-jacentes sont les mêmes**, ce qui diffère ce sont les interfaces à la bibliothèque

Envoi non-bloquant

- **Appel de MPI_ISEND**
  - Initialise un envoi
  - Retour immédiat au programme
  - Même argument que MPI_SEND
  - + un argument identifiant la requête
- **Appel de MPI_IRECV**
  - Initialise une réception
  - Retour immédiat au programme
  - Pas d'argument ''status'' (pas de complétion)
  - Un argument identifiant la requête

Requête - complétion

- **Test de complétion**

  sans bloquer le processus sur la complétion:

  MPI_TEST(requete_id,flag,status,ierr) Fortran

  - **requete_id**: (in/out) entier
  - **flag**: (out) de type logique (true si il y a complétion)
  - **status**: (out) argument tableau de MPI_STATUS_SIZE éléments
  - **ierr** : (out) code d'erreur entier

- **Attendre la complétion**: MPI_WAIT

  MPI_WAIT(requête_id,status,ierr) Fortran

Avantages et inconvénients

- **Avantages**:
  - Évite les deadlocks
  - Couvrir les communications par des calculs
- **Inconvénients**
  - Augmente la complexité des algorithmes
  - Difficile à déboguer

Communications non_bloquantes

```
program simple_deadlock

! Declarations et include …
...
! Initialisation de MPI
    call MPI_INIT(ierr)

! Recupere mon rang
    call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)

! Le processus 0 envoie, le processus 1 reçoit

     if (myrank .eq. 0) then
        call MPI_IRECV(b,100,MPI_REAL,1,19,MPI_COMM_WORLD,request,ierr)
        call MPI_SEND(a,100,MPI_REAL,1,17,MPI_COMM_WORLD,ierr)
         call MPI_WAIT(request,status,ierr)
    else if (myrank .eq. 1) then
        call MPI_IRECV(a,100,MPI_REAL,1,17,MPI_COMM_WORLD,request,ierr
        call MPI_SEND(b,100,MPI_REAL,0,19,MPI_COMM_WORLD,ierr)
         call MPI_WAIT(request, status,ierr)
     endif
...
```

4 Modes d'envoi

- **Mode standard**
  - Communications bloquantes et non-bloquantes
  - Retourne au programme après complétion
  - Bufferisé ou non bufferisé au choix de MPI (dépend de l'implémentation de MPI)
- **Mode synchronisé**
  - Synchronise les processus d'envoi et de réception. L'envoi du message est terminé si la réception a été postée et la lecture terminée.
  - Evite la recopie de message.
- **Mode « ready »**
  - L'appel du processus destinataire a déjà été posté avant l'appel du processus émetteur (dans le cas contraire le résultat est indéfini)=> optimisation
- **Mode bufferisé**
  - MPI utilise un buffer
  - Le buffer est géré par le programmeur

8 fonctions SEND dans MPI

SEND Mode	bloquant	non-bloquant
Standard	**MPI_SEND**	**MPI_ISEND**
Synchrone	**MPI_SSEND**	**MPI_ISSEND**
Ready	**MPI_RSEND**	**MPI_IRSEND**
Bufferisé	**MPI_BSEND**	**MPI_IBSEND**

Optimisations

- **Algorithmique**: le rapport communication sur calcul doit être aussi faible que possible
- **Communications**: Particulièrement important lorsque le part des communications est importante par rapport au calcul
  - Recouvrir les communications par les calculs
  - Éviter la recopie des messages dans un espace mémoire temporaire
  - Minimiser les surcoûts occasionner par les appels répétitifs aus sous-programmes de communication

Optimisation: synchrone - bloquant

- **MPI_SSEND**()
- Permet d'éviter la recopie de message dans un espace temporaire
- Performant sur une application bien équilibrée
- Associés dans certaines implémentations à des copies mémoire à mémoire
- Il suffit de remplacer le MPI_SEND par MPI_SSEND

Optimisation: synchrone – non bloquant

- **MPI_ISSEND**() et **MPI_IRECV**()
- Permet de recouvrir les communications par les calculs
- **MPI_WAIT()** synchronisation du processus jusqu'à terminaison de la requête
- **MPI_TEST()**: test la complétion de la requête

Optimisation: conseil1

- Éviter si possible la recopie temporaire de message en utilisant la routine **MPI_SSEND()**
- Superposer les communications par des calculs lorsque c est possible en utilisant les routines **MPI_ISSEND()** et **MPI_IRECV()**

Communications persistantes

- **Contexte**: boucler sur un envoi ou une réception de message ou la valeur des données manipulées changent mais ni l'adresse, ni le type
- **Utilisation**
  - Créer un schéma persistant de communication à l'extérieur de la boucle
    - Envoi standard: **MPI_SEND_INIT()**
    - Envoi synchrone: **MPI_SSEND_INIT()**
    - Envoi bufferisé: **MPI_BSEND_INIT()**
    - Réception standard: **MPI_RECV_INIT()**
  - Activer la requête d'envoi ou de réception à l'intérieur de la boucle: **MPI_START(requette,code)**
  - Libérer la requête en fin de boucle: **MPI_REQUEST_FREE(requette, code)** elle ne sera libérée qu'une fois qu'elle sera réellement terminée

Communications persistantes: exemples

```
if (rang == 0) then
        call MPI_SSEND_INIT(c,m*m,MPI_REAL,1,etiquette,& MPI_COMM_WORLD,requete0,code)
        do k = 1, 1000 !*** J'envoie un gros message
                call MPI_START(requete0,code)
                call sgetrf(na, na, a, na, pivota, code)
                call MPI_WAIT(requete0,statut,code) !*** Ce calcul modifie le contenu du tableau C
                c(1:nb,1:nb) = matmul(a(1:nb,1:nb),b)
        end do
elseif (rang == 1) then
        call MPI_RECV_INIT(c,m*m,MPI_REAL,0,etiquette,&
                MPI_COMM_WORLD,statut,requete1,code)
        do k = 1, 1000
                call sgetrf(na, na, a, na, pivota, code) !*** Je reçois le gros message
                call MPI_START(requete1,code) !*** Ce calcul est indépendant du message
                call sgetrf(nb, nb, b, nb, pivotb, code)
                call MPI_WAIT(requete1,statut,code) !*** Ce calcul dépend du message précédent
                a(:,:) = transpose(c(1:na,1:na)) + a(:,:)
        end do
end if
```

Optimisation: conseil2

- Minimiser les surcoûts induits par les appels répétitifs aux fonctions de communications par un schéma de communication persistant
- Recouvrir les communications avec des calculs tout en évitant la recopie temporaire des messages (MPI_SSEND_INIT()): un schéma persistant est activé de façon non bloquante par MPI_START

Communications collectives

- Procédures de communication sur **un groupe de processus**
  - Série de communications point à point en une seule opération
  - Routines fournies par MPI utilisant des **algorithmes optimaux**
- Le groupe de processus est identifié par un **communicateur**
- Pour chaque processus, l'appel se termine quand l'opération collective est terminée
- **La gestion des étiquettes** est à la charge du système, elles ne sont pas données explicitement à l'appel

Communications collectives MPI

- **3 types de routines:**
  - **Synchronisation globale: MPI_BARRIER()**
  - **Transfert des données**:
    - Diffusion globale des données: **MPI_Bcast()**
    - Diffusion sélective des données: **MPI_Scatter()**
    - Collecte des données réparties: **MPI_Gather()**
    - Collecte par tous les processus des données réparties: **MPI_ALLGather()**
    - Diffusion sélective, par tous les processus, des données réparties: **MPI_ALLTOALL()**
  - **Communication et opérations sur les données**
    - Réduction de type prédéfinie ou personnel: **MPI_REDUCE()**
    - Réduction avec diffusion du résultat: **MPI_ALLREDUCE()**
- Version vectorielles de ces routines

Barrière de synchronisation

- Impose un **point synchronisation à tous les processus** du communicateur
- Routine **MPI_BARRIER**

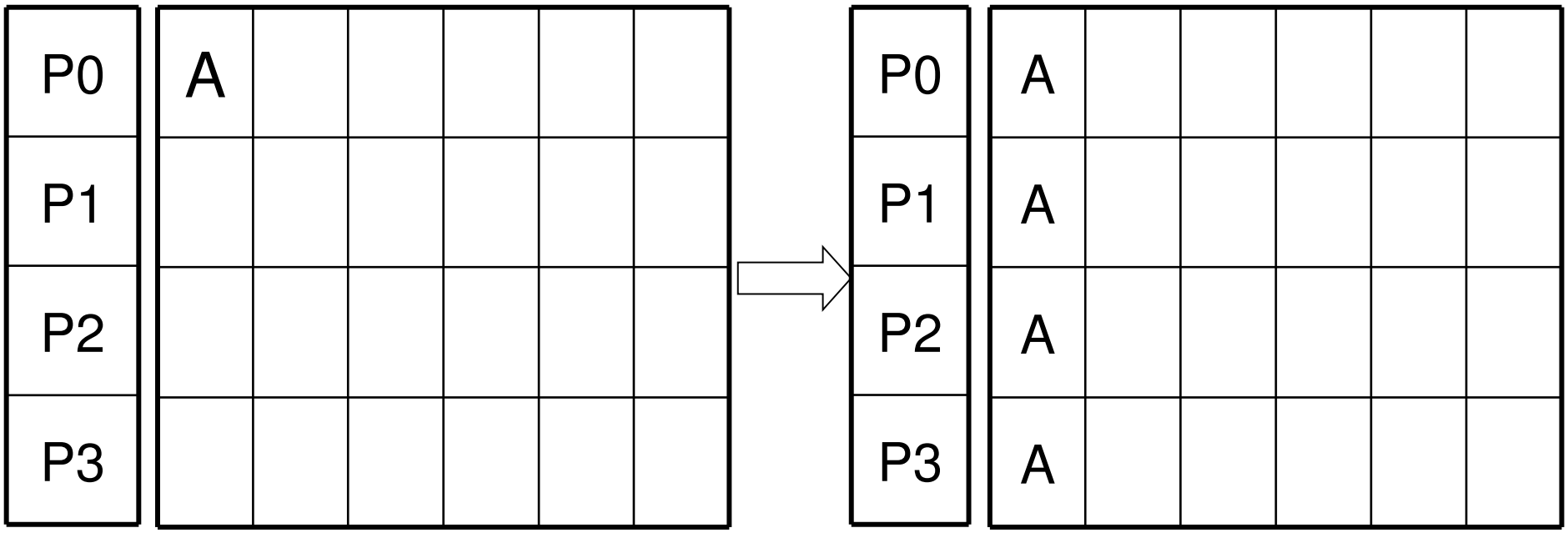
  Fortran:

```
integer :: comm, ierror
call MPI_BARRIER(comm,ierror)
```

  C:

```
int MPI_Barrier(comm)
MPI_Comm comm
```

Broadcast de A de P0 vers tous les processus



Fortran:

call MPI_Bcast(send_buffer,send_count,send_type, rank,comm)

send_buffer : adresse de début du buffer envoyé

send_count : nombre d'éléments

rank : processus qui envoie

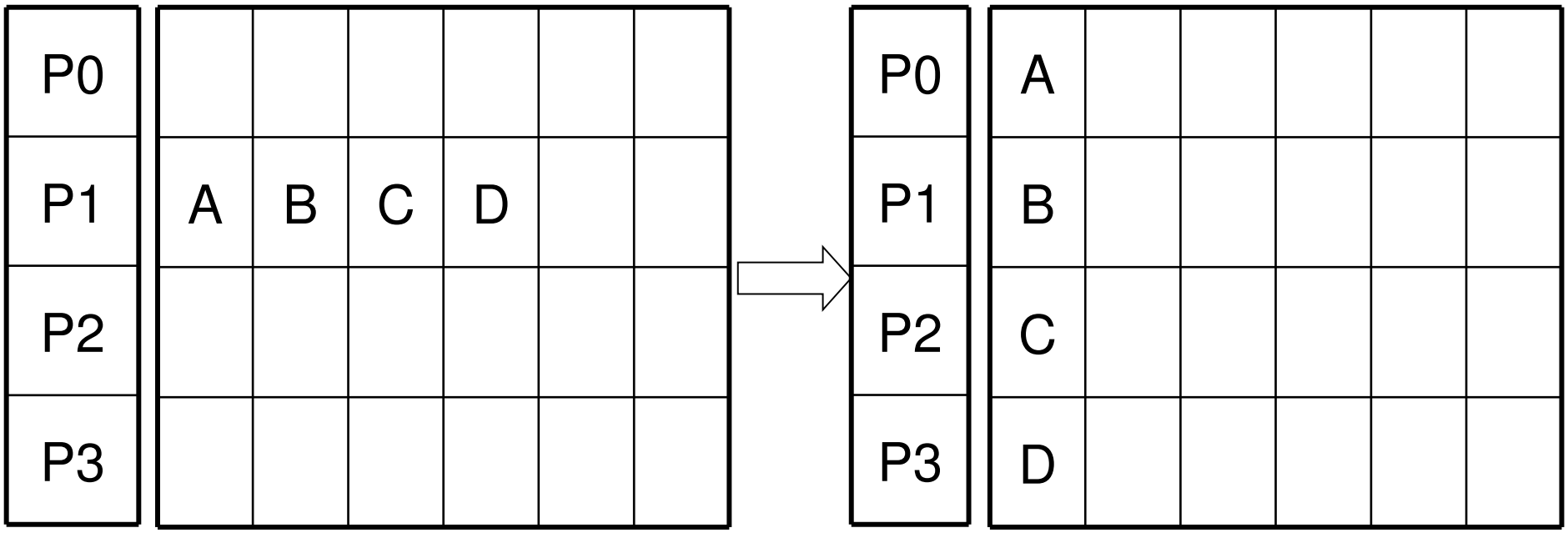
comm : communicateur MPI

Diffusion Générale : MPI_Bcast()

```
program bcast_idris
        implicit none
        include 'mpif.h'
        integer, parameter              :: nb_valeurs=128
        integer                         ::rang,valeur,ier

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

        if (rang == 2) valeur=rang+1000
        call MPI_Bcast(valeur,1,MPI_INTEGER,2, &
          MPI_COMM_WORLD,ier)

        print *,'moi processeur ',rang, j''ai reçu',valeur,'du processus

        call MPI_FINALiZE(ier)
end program gather_idris
```

```
% mpirun –np 4 bcast
Moi processeur 2, j'ai reçu 1002 du processeur 2
Moi processeur 0, j'ai reçu 1002 du processeur 2
Moi processeur 1, j'ai reçu 1002 du processeur 2
Moi processeur 3, j'ai reçu 1002 du processeur 2
```

Diffusion sélective: MPI_Scatter()



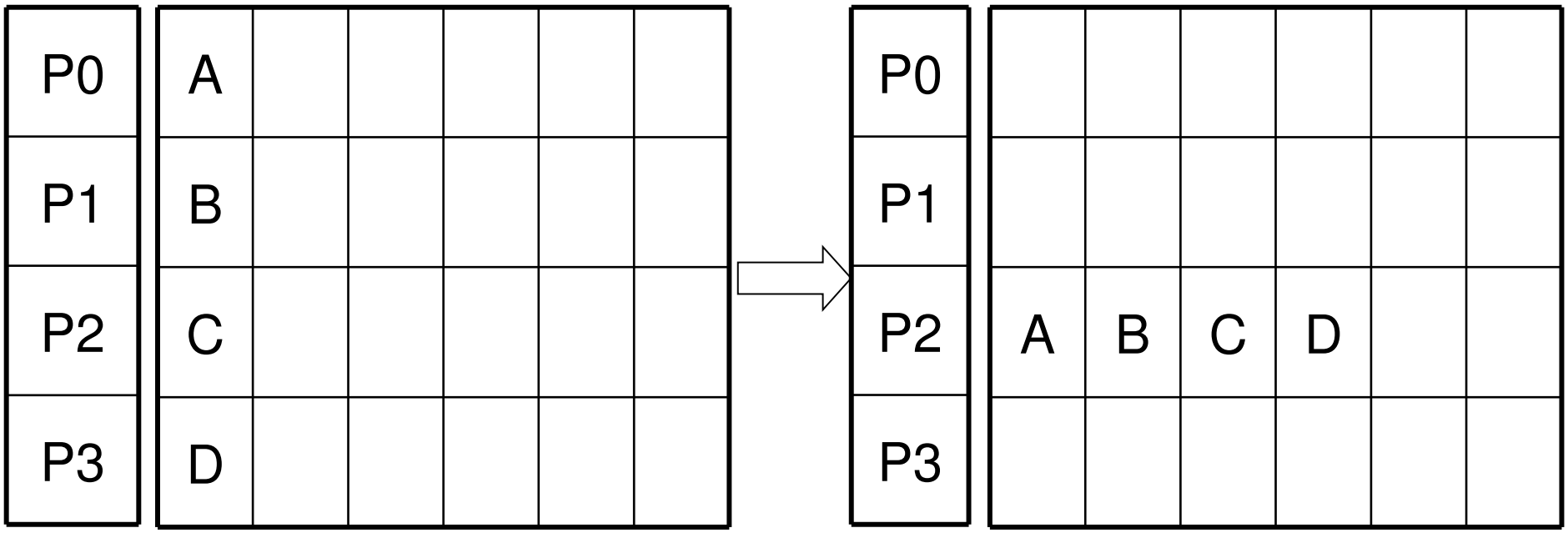
Fortran:

```
call MPI_Scatter(send_buffer,send_count,send_type
    recv_buffer,recv_count,recv_type,rank,comm,ierror)
integer send_count,send_type,recv_type,rank,ierror
<don_datatype> send_buffer,recv_buffer
```

Diffusion Sélective : MPI_Scatter()

```
program scatter_idris
        implicit none
        include 'mpif.h'
        integer, parameter                  :: nb_valeurs=128
        integer                             ::nb_procs,rang,long_tab,i,ier
        real,allocatable,dimension(:)   ::valeurs,donnees

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

        long_tab=nb_valeurs/nb_procs
        allocate(donnees(long_tab))
        if (rang == 2) then
          allocate(valeurs(nb_valeurs)
          valeurs(:)=(/(1000.+i,i=1,nb_valeurs)/)
        end if

        call MPI_SCATTER(valeurs,long_tab,MPI_REAL,donnees, &
            long_tab,MPI_REAL,2,MPI_COMM_WORLD,ier)

        if (rang == 2) print *,'moi processeur 2, j''ai reçu',donnees(1),&
          donnees(long_tab+1),'…',donnees(nb_valeurs)

        call MPI_FINALIZE(ier)
end program scatter_idris
```

Collecte : MPI_GATHER()



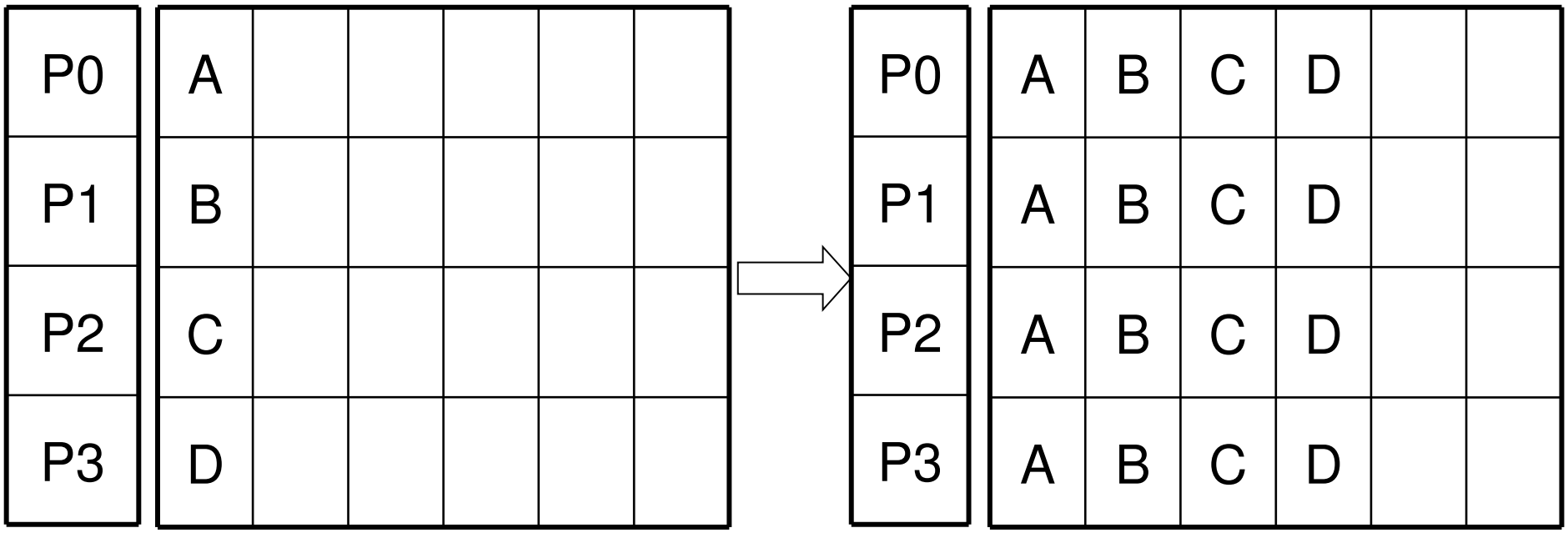
Fortran:

```
call MPI_Gather(send_buffer,send_count,send_type
    recv_buffer,recv_count,recv_type,rank,comm,ierror)
integer send_count,send_type,recv_type,rank,ierror
<don_datatype> send_buffer,recv_buffer
```

Collecte : MPI_Gather()

```
program gather_idris
        implicit none
        include 'mpif.h'
        integer, parameter                  :: nb_valeurs=128
        integer                             ::nb_procs,rang,long_tab,i,ier
        real,allocatable,dimension(:)   ::valeurs
        real,dimension(nb_valeurs)      :: donnees

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

        long_tab=nb_valeurs/nb_procs
        allocate(valeurs(long_tab))
        valeurs(:)=(/(1000.+rang*long_tranche+i,i=1,long_tab)/)

        call MPI_GATHER(valeurs,long_tab,MPI_REAL,donnees, &
            long_tab,MPI_REAL,2,MPI_COMM_WORLD,ier)

        if (rang == 2) print *,'moi processeur 2, j''ai reçu',donnees(1),&
          donnees(long_tab+1),'…',donnees(nb_valeurs)

        call MPI_FINALiZE(ier)
end program gather_idris
```

Collecte Générale: MPI_ALLGather()



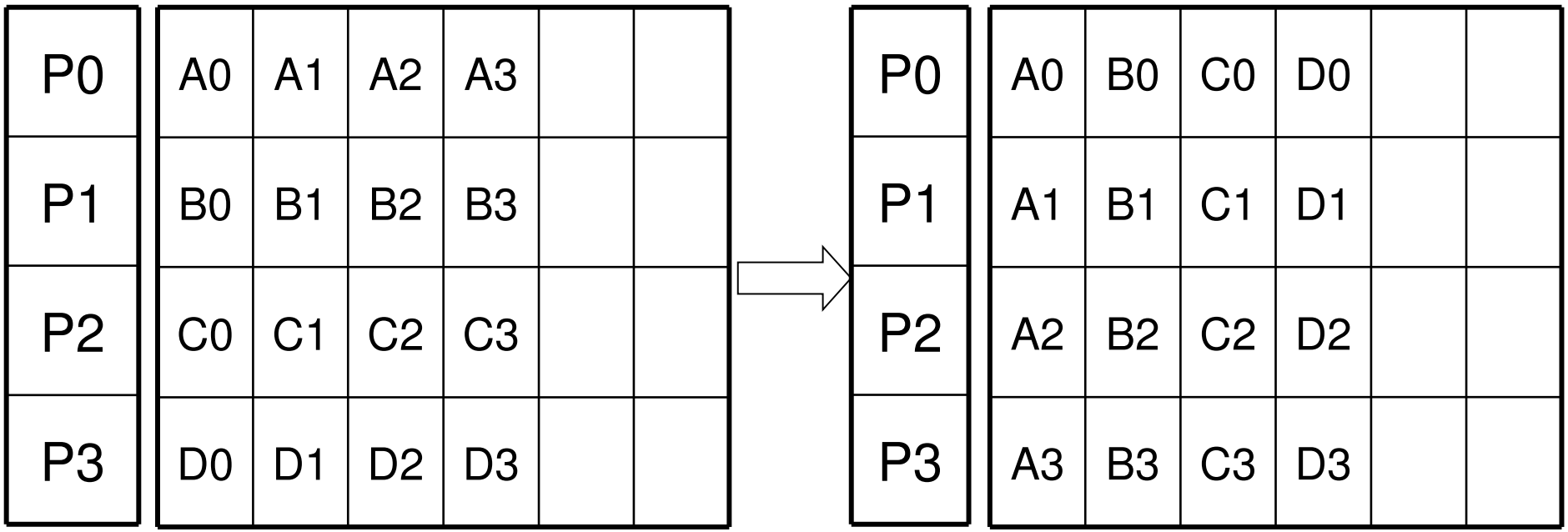
Fortran:

```
call MPI_ALLGather(send_buffer,send_count,send_type
    recv_buffer,recv_count,recv_type,comm,ierror)
integer send_count,send_type,recv_type,ierror
<don_datatype> send_buffer,recv_buffer
```

Collecte Générale: MPI_ALLGather()

```
program allgather_idris
        implicit none
        include 'mpif.h'
        integer, parameter                  :: nb_valeurs=128
        integer                             ::nb_procs,rang,long_tab,i,ier
        real,allocatable,dimension(:)   ::valeurs
        real,dimension(nb_valeurs)      :: donnees

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

        long_tab=nb_valeurs/nb_procs
        allocate(valeurs(long_tab))
        valeurs(:)=(/(1000.+rang*long_tranche+i,i=1,long_tab)/)

        call MPI_ALLGATHER(valeurs,long_tab,MPI_REAL,&
            donnees,long_tab,MPI_REAL,MPI_COMM_WORLD,ier)

        if (rang == 2) print *,'moi processeur 2, j''ai reçu',donnees(1),&
          donnees(long_tab+1),'…',donnees(nb_valeurs)

        call MPI_FINALiZE(ier)
end program gather_idris
```

Échanges croisés: MPI_ALLTOALL()



Réductions Réparties

- Réduction des données réparties sur un ensemble de processus avec récupération du résultat
  - sur un seul processus: MPI_REDUCE()
  - Sur tous les processus: MPI_ALLREDUCE()
- Réduction:
  - Prédéfinie par MPI: somme,max,min,…(Somme, maximum,… des éléments d'un vecteur): MPI_REDUCE() et MPI_ALLREDUCE()
  - Définie par le programmeur : MPI_OP_CREATE et MPI_OP_FREE
- Réduction partielle: MPI_SCAN()

Opérations de réduction définie par MPI

Nom	Opération
MPI_SUM	Somme des éléments
MPI_PROD	Produit des éléments
MPI_MAX	Recherche du maximum
MPI_MAXLOC	Recherche de l'indice du maximum
MPI_MINLOC	Recherche de l'indice du minimum
MPI_LAND	ET Logique
MPI_LOR	OU logique
MPI_LXOR	OU exclusif logique

Types dérivés: sommaire

- Motivations
- Envoi de données non contiguës
  - Envoi de messages successifs
  - Utilisation de buffer: **MPI_PACK, MPI_UNPACK**
  - Utilisation de types dérivés
- Types dérivés MPI
  - Définition et validation (**MPI_TYPE_STRUCT,…**)
  - Méthode de définition de type prédéfinie
    - **MPI_TYPE_CONTIGUOUS**
    - **MPI_TYPE_VECTOR**
    - **MPI_TYPE_HVECTOR**
    - **MPI_TYPE_INDEXED**
    - **MPI_TYPE_HINDEXED**
- Utilisation de type dérivé MPI pour les types dérivés utilisateur

Types dérivés : Introduction

Motivations:

- ✓ Transmettre des données de **type mixte** et/ou **non contiguës en mémoire**
- ✓ Utiliser des structures de données plus complexes
- ✓ Utiliser des types dérivés MPI pour les **types dérivés utilisateurs** (mapping)

Données de type mixte ou non contigues en mémoire

Plusieurs stratégies pour transmettre des données de type mixte ou non contigues en mémoire.

- ☑ Envoyer chaque paquet de données **homogènes** contiguës **en mémoire** à l'aide de messages séparés
- ☑ Copier les données dans **un buffer (MPI_PACK)**
- ☑ **Utiliser les type dérivés MPI** pour décrire
  - ✓ Un bloc de **données homogènes contiguës** en mémoire
  - ✓ Un bloc de données **homogènes non contiguës** en mémoire séparées par **un pas fixe**
  - ✓ Un bloc de **données homogènes non contiguës** en mémoire séparées par **un pas variable**
  - ✓ Un bloc de données **non homogènes et non contiguës**

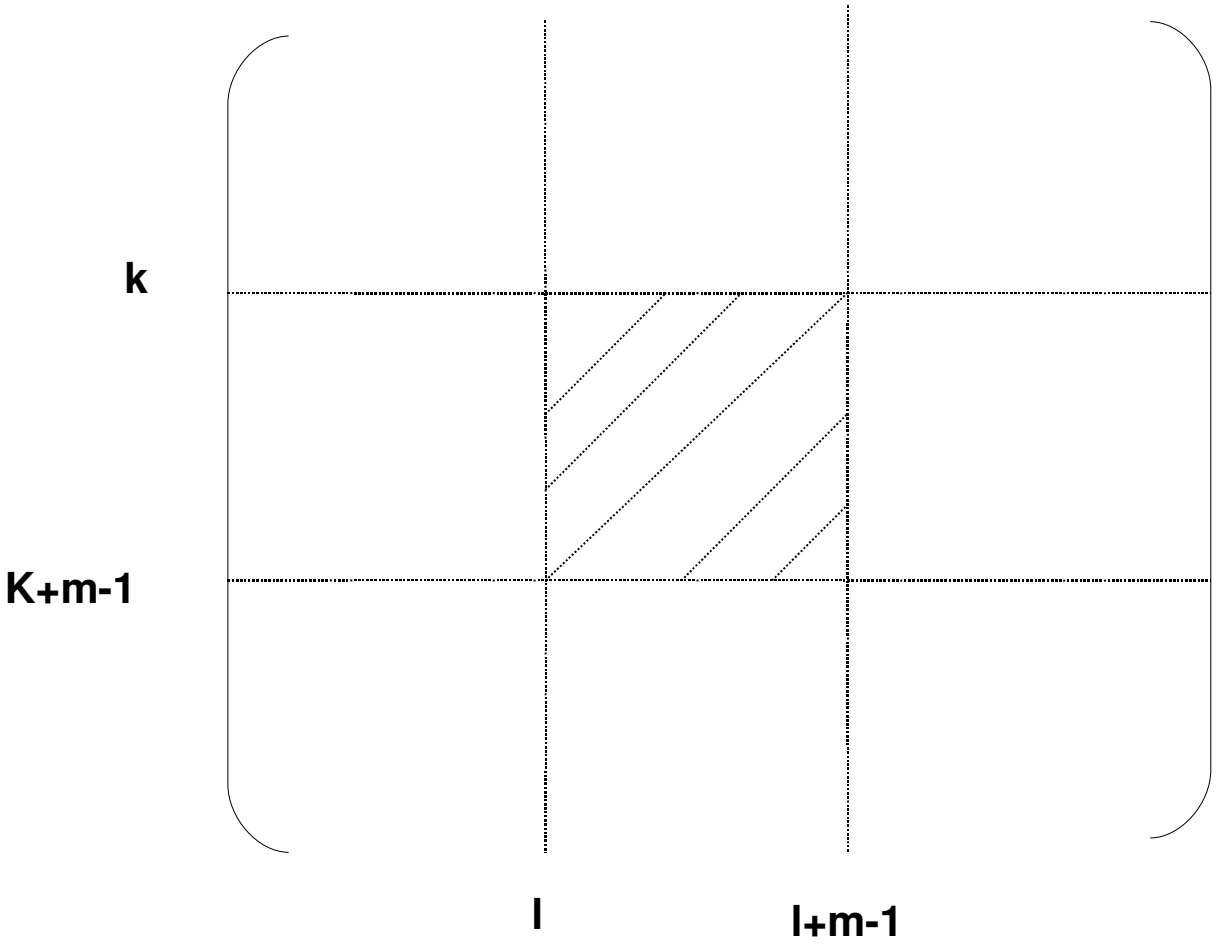
Utilisation d'un buffer (MPI_PACK)

- **Stocker des données de types mixtes:** Appels successifs à MPI_PACK(…,count)
- **Envoi du buffer avec le type « MPI_PACK»**
- **Réception des données bufferisées**
  - **Données homogènes:** en indiquant leur type dans l'appel de la réception
  - **Données non homogènes**:
    - Réception avec le type MPI_PACK
    - Appel à MPI_UNPACK()

Utilisation d'un buffer (MPI_PACK)

- **Évaluation de la taille** utile du buffer après l'appel de la routine **MPI_PACK_SIZE**
- **Conclusion**
  - **Flexibilité** dans l'utilisation de données de type mixte non contiguës de façon incrémentale
  - **Le coût** réside dans
    - l'utilisation de l'espace mémoire
    - Le temps CPU pour copier et récupérer les données dans le buffer

Définition d'une sous-matrice de la matrice réelle A(n,n)



Utilisation d'un buffer (MPI_PACK)

- **Envoi:**
  - ✓ Appel de **MPI_PACK** pour la bufferisation
  - ✓ Envoi du buffer avec le type MPI_PACK

```
count=0
call MPI_PACK(m,1,MPI_INTEGER,
&     buffer,bufsize,count,MPI_COMM_WORLD,ierror
do  j=1,m
      call MPI_PACK(a(k,l+j-1),n,MPI_DOUBLE,
&     buffer,bufsize,count,MPI_COMM_WORLD,ierror)
 enddo
 call MPI_SEND(buffer,count,MPI_PACKED,dest,tag
&  ,MPI_COMM_WORLD,ierror)
```

Utilisation d'un buffer (MPI_PACK)

- **Réception :**
  - Données homogènes: réception avec le type (ex:MPI_DOUBLE)
  - Données hétérogènes: avec le type MPI_PACKED et appel de la routine MPI_UNPACKED

Utilisation des types dérivés

- Les phases de bufferisation sont intégrées dans les appels de communication(send et receive)
- **Elle évite**:
  - **Les étapes de bufferisation** avant l'envoi et de débufferisation après la réception
  - **La copie** entre le buffer intermédiaire(MPI_PACK) et le buffer de communication (MPI_SEND)
- En général **plus efficace** que la méthode de bufferisation (MPI_PACK,MPI_UNPACK)

Définition d'un type Dérivé

Un type dérivé est constitué de **plusieurs composants**

- **Constructions de 3 tableaux** contenant:
  - La **longueur** de chaque composant
  - Sa **localisation:** appel de **MPI_ADDRESS**
  - Son **type**
- **Définir le nouveau type**: appel **MPI_TYPE_STRUCT()**
- **Valider ce type** pour les procédures de **communication:** appel de **MPI_TYPE_COMMIT()**
- **Libérer un type MPI**: appel à **MPI_TYPE_FREE()**

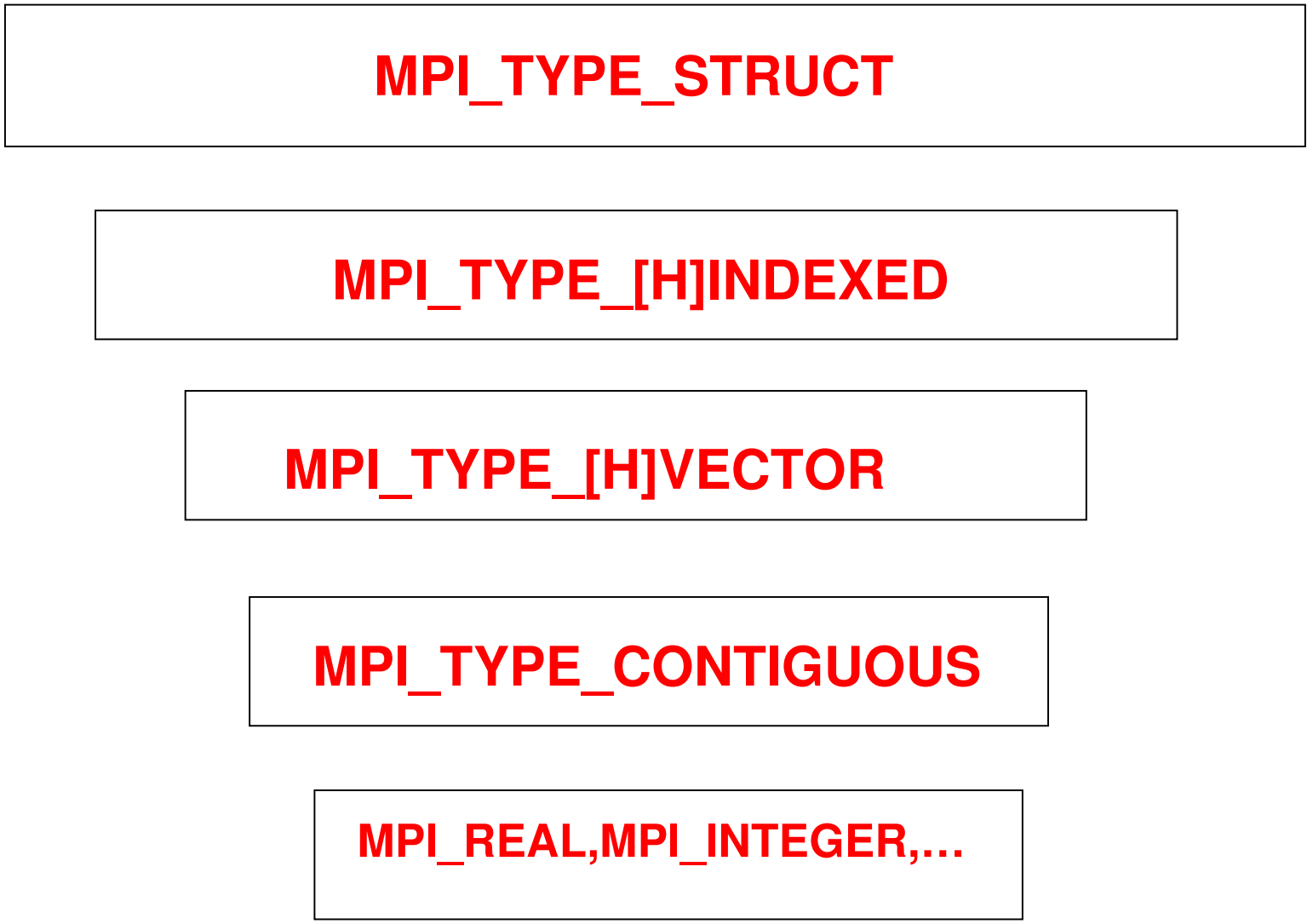
Envoi d'une sous-matrice A(k+i-1,i=1,m;l+j-1,j=1,n)

```
...
! Construction des 3 tableaux : loca,lena,typa
  do i=1,m
    lena(i)=n
    call MPI_ADDRESS(A(k,l+i-1),loca(i),ierror)
    typa(i)=MPI_DOUBLE
  enddo

! Définition du type MPI
  call MPI_TYPE_STRUCT(m,lena,loca,typa,mon_type_mpi,ierror)

! Validation du type MPI
  call MPI_TYPE_COMMIT(mon_type_mpi,ierror)

! Envoi de la sous-matrice
  call MPI_SEND(MPI_BOTTOM,1,mon_type_mpi,
 & dest,tag,MPI_COMM_WORLD,ierror)

! Libération du type MPI
  call MPI_TYPE_FREE(mon_type_mpi,ierror)
...
```

Autres définitions de type dérivé MPI

- **Données homogènes contiguës en mémoire**: MPI_TYPE_CONTIGUOUS
- **Données homogènes non contiguës en mémoire espacées à pas constant**: MPI_TYPE_VECTOR (MPI_TYPE_HVECTOR)
- **Blocs de données de longueur variable et espacés à pas variable** : MPI_TYPE_INDEXED(MPI_TYPE_HINDEXED)

Hiérarchie de types dérivés MPI



Types dérivés
données homogènes contiguës

1	6	11	16	21	26
2	7	12	17	22	27
3	8	13	18	23	28
4	9	14	19	24	29
5	10	15	20	25	30

call **MPI_TYPE_CONTIGUOUS**(**5**,**MPI_REAL**,**nouveau_type**,**ierror**)

Types dérivés
Données homogènes à pas constant

- **Blocs de données homogènes distants d'un pas constant.** Le pas est donné en nombre d'éléments.

```
…
integer,intent(in) :: nombre_blocs,longueur_bloc
integer,intent(in) :: pas  ! En nombre d'elements
integer,intent(in) :: ancien_type
integer,intent(out) :: nouveau_type,ierror
…
 call MPI_TYPE_VECTOR(nombre_blocs,longueur_bloc,
&  pas,ancien_type,nouveau_type,ierror)
…
```

Type dérivé: type_ligne

1	6	21	26	31	36
2	7	22	27	32	37
3	8	23	28	33	38
4	9	24	29	34	39
5	10	25	30	35	40

call MPI_TYPE_VECTOR(6,1,5,MPI_REAL,nouveau_type,code)

MPI_TYPE_HVECTOR identique à MPI_TYPE_VECTOR, la distance entre deux blocs est comptée en nombre de bytes

Types dérivés
Données homogènes à pas variable

- **MPI_TYPE_INDEXED()** :
  - permet de créer une structure de données composée de blocs de longueur variable, espacés d'un pas variable
- **MPI_TYPE_HINDEXED()** :
  - même fonctionnalité que MPI_TYPE_INDEXED sauf que le pas séparant deux blocs est donné en octets
  - Utile lorsque le type des éléments de base n'est pas un type intrinsèque MPI
  - Attention à la portabilité de cette routine

Types dérivés
Données homogènes à pas variable

nb=3 ; longueur_blocs=(/2,1,2/ ;deplacements=(/0,3,7/)

Ancien type

Nouveau type

```
integer,parameter,intent(in)       :: nb=3
integer,intent(in),dimension(nb) :: longueurs_blocs,deplacements
integer,intent(in)                 :: ancien_type
integer,intent(out)                :: nouveau_type,ierror
call MPI_TYPE_INDEXED(nb,longueurs_blocs,deplacements,
    ancien_type,nouveau_type,ierror)
```

nb=4 ; longueur_blocs=(/2,1,2,1/ ;deplacements=(/2,10,14,24/)

ancien_type

nouveau_type

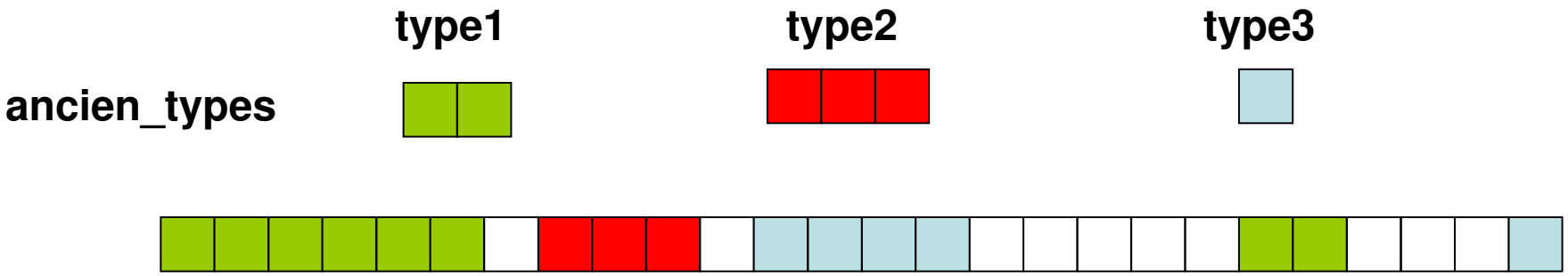
```
integer,parameter,intent(in)       :: nb=4
integer,intent(in),dimension(nb) :: longueurs_blocs,deplacements
integer,intent(in)                      :: ancien_type
integer,intent(out)                     :: nouveau_type,ierror

call MPI_TYPE_HINDEXED(nb,longueurs_blocs,deplacements,
     ancien_type,nouveau_type,ierror)
```

Types Dérivés
Données Hétérogènes

- **MPI_TYPE_STRUCT:** constructeur de type dérivé le plus général
- Mêmes fonctionnalités que **MPI_TYPE_INDEXED** mais il permet **la réplication de blocs hétérogènes**
- **Mêmes paramètres que MPI_TYPE_INDEXED** mais
  - **ancien_types** est un vecteur
  - Le calcul de déplacement se fait sur des différences d'adresses
- MPI_ADRESS() est un routine portable MPI qui retourne l'adresse d'une variable

nb=5 ; longueur_blocs=(/3,1,4,1,1/); deplacements=(/0,7,11,20,25/)

ancien_types=(type1,type2,type3,type1,type3)




```
integer,parameter,intent(in)          :: nb=5
integer,intent(in),dimension(nb)      :: longueurs_blocs,deplacements
integer,intent(in),dimension(nb)      :: ancien_types
integer,intent(out)                   :: nouveau_type,ierror

call MPI_TYPE_STRUCT(nb,longueurs_blocs,deplacements,
     ancien_types,nouveau_type,ierror)
```

Envoi d'une particule du proc 0 au proc 1(ex IDRIS)

```
program interaction_particules
…
Definition du type « particule »
type particule
   character(len=5)       :: type
   integer                :: masse
   real,dimension(3)      :: coord
   logical                :: classe
end type particule
…
nitialisation MPI
call MPI_INIT(ier)
call MPI_COMM_RANK(MPI_COMM_WORLD,rang,ier)
…
Construction du type MPI
types=(/MPI_CHARACTER,MPI_INTEGER,MPI_REAL,MPI_LOGICAL/)
longueurs_blocs=(/5,1,3,1/)
```

```
  call MPI_ADDRESS(p(1)%type  ,adresse(1),ier)
  call MPI_ADDRESS(p(1)%masse, adresse(2),ier)
  call MPI_ADDRESS(p(1)%coords, adresse(3),ier)
  call MPI_ADDRESS(p(1)%classe, adresse(4),ier)
! Calcul des deplacements
  do i=1,4
    deplacement(i)=adresse(i)-adresse(1)
  enddo
  call MPI_TYPE_STRUCT(4,longueurs_blocs,deplacement,
 &        types,type_particule,ier)
  call MPI_TYPE_COMMIT(type_particule,ier)
  ....
! Envoi des particules de 0 vers 1
  if (rang .eq.0) then
    call MPI_SEND(p(1)%type,n,type_particule,1,etiquette,
 &    MPI_COMM_WORLD,ier)
   else
    call MPI_RECV( p(1)%type,type_particule,0,etiquette,
 &    MPI_COMM_WORLD,statut,ier)
  ...
```

```
! Fin MPI
  call MPI_TYPE_FREE(type_particule)
  call MPI_FINALIZE(ier)
  end program interaction _ particules
```

Informations sur un type dérivé MPI

- Taille totale: **MPI_TYPE_SIZE()**
- Bornes inférieures et supérieures:
  - **MPI_TYPE_LB()**
  - **MPI_TYPE_UB()**

MPI_TYPE_SIZE≤MPI_TYPE_Ub()-MPI_TYPE_LB()

- **MPI_TYPE_EXTENT()**

Types Dérivés : Conclusions

- ☑ Outils puissants et portables de **description de donnée**
  - ✓ Adaptation au types utilisateurs
- ☑ Ils rendent les échanges de données:
  - ✓ Plus simples et plus claires
  - ✓ Plus performants dans le cas de structure de données hétérogènes
- ☑ L'association des topologies et des types dérivés simplifie l'écriture des problèmes de décomposition de domaine

Introduction à MPI

Communicateurs - Topologie

Communicateurs

- **Ensemble de processus actifs**
- Toutes les opérations de communications se font sur un communicateur
- Le communicateur par défaut est **MPI_COMM_WORLD** qui comprend tous les processus actifs
- **À tout instant, on peut connaître**:
  - **Le nombre de processus** gérés par un communicateur donné
  - **Le rang du processus courant** dans un communicateur donné

Communicateurs

- MPI_COMM_WORLD est le communicateur par défaut créé par MPI à l'initialisation
- Il peut être utile de définir un nouveau communicateur constitué d'un sous groupe de processus qui pourront communiquer entre eux
- Deux types de communicateurs:
  - **Intra-communicateur**: défini un contexte de communication entre processus
  - **Inter-communicateur**: défini un contexte de communications entre communicateur

Groupes et communicateurs

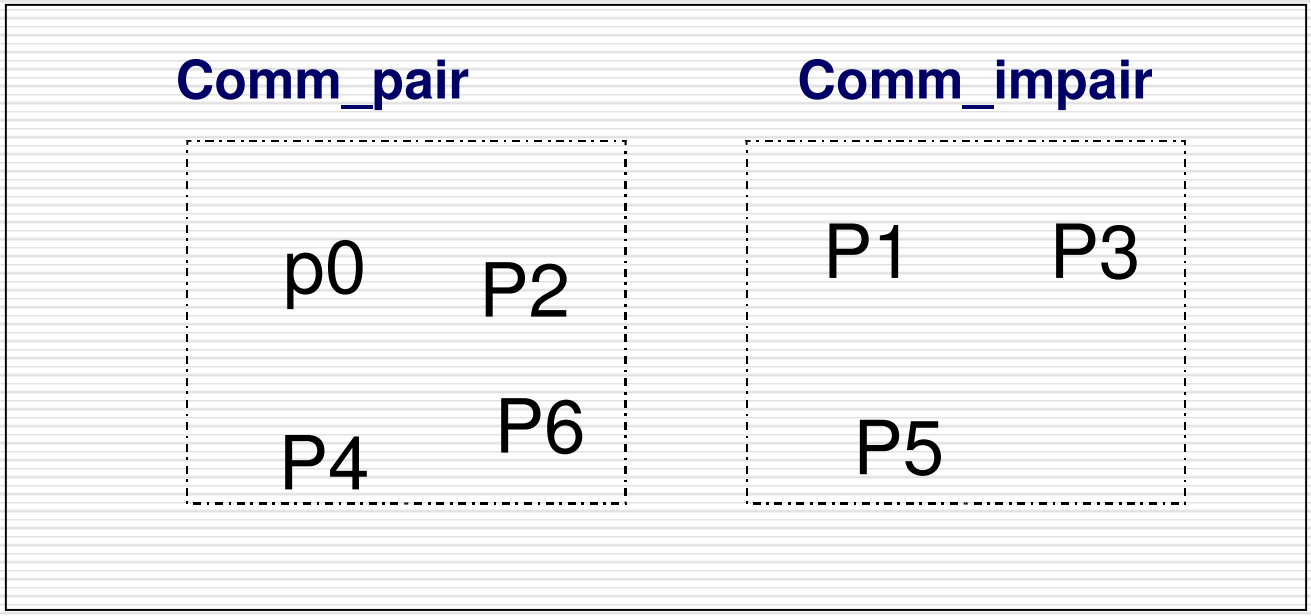
- Un groupe est un ensemble de processus
- Un contexte de communication :
  - permet de délimiter l'espace de communication
  - Il est géré uniquement par MPI
- Un communicateur est constitué:
  - D'un groupe de processus
  - D'un contexte de communication mis en place par MPI à la création du communicateur
- Deux méthodes pour construire un communicateur
  - Par l'intermédiaire d'un groupe
  - À partir d'un communicateur existant

Groupes et communicateurs

Routines MPI pour:

- Construire des groupes de processus:
  - **MPI_GROUP_INCL()**
  - **MPI_GROUP_EXCL()**
- Construire un communicateur:
  - **MPI_CART_CREATE()**
  - **MPI_CART_SUB()**
  - **MPI_COMM_CREATE()**
  - **MPI_COMM_DUP()**
  - **MPI_COMM_SPLIT()**
- Les groupes et les communicateurs peuvent être libérés: **MPI_GROUP_FREE(),MPI_COMM_FREE()**

Création de deux communicateurs Exemple



Deux groupes de processus dans un communicateur

Construction de communicateurs associés

Communicateurs issus de groupes

- Les étapes:
  - Connaître le groupe associé à MPI_COMM_WORLD: **MPI_COMM_GROUP()**
  - Créer le groupe des processus pairs: **MPI_GROUP_INCL()**
  - Créer le communicateur associé à ce groupe: **MPI_COMM_CREATE()**
  - Déterminer le rang dans le nouveau communicateur d'un processus de rang donné: **MPI_GROUP_TRANSLATE_RANKS()**
  - Effectuer les communications dans les nouveaux communicateurs: **MPI_BCAST()** par exemple

Communicateur issu d'un groupe

Des routines MPI permettent de:

- Comparer deux groupes
  - **MPI_GROUP_COMPARE**(group1,group2,result)
- D'appliquer des opérateurs ensemblistes
  - **MPI_GROUP_UNION**(group1,group2,nv_groupe)
  - **MPI_GROUP_INTERSECTION**(group1,group2, nv_groupe)
  - **MPI_GROUP_DIFFERENCE**(group1,group2, nv_groupe)
- Un groupe qui est vide prend la valeur **MPI_GROUP_EMPTY**

```
 program GroupePairImpair (exemple IDRIS)
 include 'mpif.h'
! Déclarations
 integer, parameter :: m=4
 integer,dimension(:),allocatable  :: rangs_pair
 integer   ::a(m),comm_pair,comm_impair,dim_rangs_pair,iproc,i…
 ….
 call MPI_INIT(ier)
 call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,ier)
 call MPI_COMM_RANK(MPI_COMM_WORLD,rang,ier)

! Initialisation du vecteur a
 a=0.
 if (rang == 2) a=2.
 if (rang == 3) a=3.
! Enregister le rangs des processus pairs
 dim_rangs_pair=int((nb_procs+1)/2)
 allocate(rangs_pair(dim_rangs_pair))
 do iproc=0,nb_procs-1,2
   i=i+1
   rangs_pair(i)=iproc
 enddo
```

```
! Connaître le group associe au communicateur MPI_COMM_WORLD
  call MPI_COMM_GROUP(MPI_COMM_WORLD,grp_monde,ier)

! Créer le groupe des procesus pairs
  call MPI_GROUP_INCL(MPI_COMM_WORLD,dim_rangs_pair,
          rangs_pair,grp_pair,ier)

! Créer le communicateur des processus pairs
  call MPI_COMM_CREATE(MPI_COMM_WORLD,grp_pair,
          comm_pair,ier)

! Créer le group des processus impairs
  call MPI_GROUP_EXCL(grp_monde,dim_rangs_pair,rangs_pair,
          grp_impair,ier)

! Créer le communicateur des processus impairs
  call MPI_COMM_CREATE(MPI_COMM_WORLD,grp_impair,
          comm_impair,ier)
```

```
  if (mod(rang,2) == 0 ) then
!  Trouver le rang du processus 2 dans « comm_pair »
    call MPI_GROUP_TRANSLATE_RANKS(grp_monde,1,2,&
          grp_pair,rang_ds_pair,ier)
! Diffuser a seulement aux processus de rangs pairs
    call MPI_BCAST(a,m,MPI_INTEGER,rangs_ds_pair,comm_pair,ier)
! Destruction du communicateur comm_pair
    call MPI_COMM_FREE(comm_pair,ier)
  else
! Trouver le rang du processeur 3 dans « comm_impair »
    call MPI_GROUP_TRANSLATE_RANKS(grp_monde,1,3,&
          grp_impair,rang_ds_impair,ier)
! Diffuser a seulement aux processus de rangs impairs
    call MPI_BCAST(a,m,MPI_INTEGER,rangs_ds_impair,&
          comm_impair,ier)
! Destruction du communicateur comm_impair
    call MPI_COMM_FREE(comm_impair,ier)
  endif
  print*,'processus ',rang, '  a = ',a

  call MPI_FINALIZE(ier)
  end program GroupePairImpair
```

Communicateur issu d'un communicateur

Pour éviter:

- De nommer différemment des communicateurs construits
- De passer par des groupes
- De laisser le choix à MPI d'ordonner les processus dans les nouveaux communicateurs
- Pour éviter les tests conditionnels, en particulier lors de l'appel de routines de communication

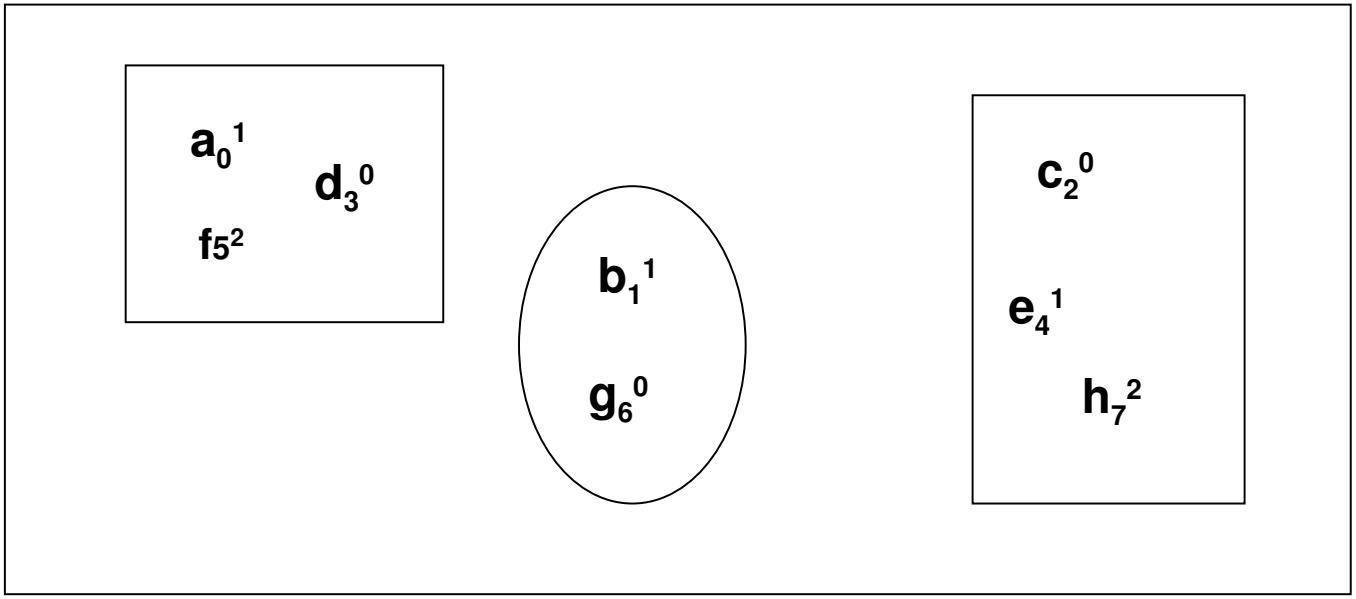
MPI_COMM_SPLIT permet de:

- Partitionner un communicateur en autant de communicateurs que l'on veut.

MPI_COMM_SPLIT(comm,couleur,clef,nv_comm,ier)

rang	0	1	2	3	4	5	6	7
Processus	a	b	c	d	e	f	g	h
couleur								
clef	2	15	0	0	1	3	11	1

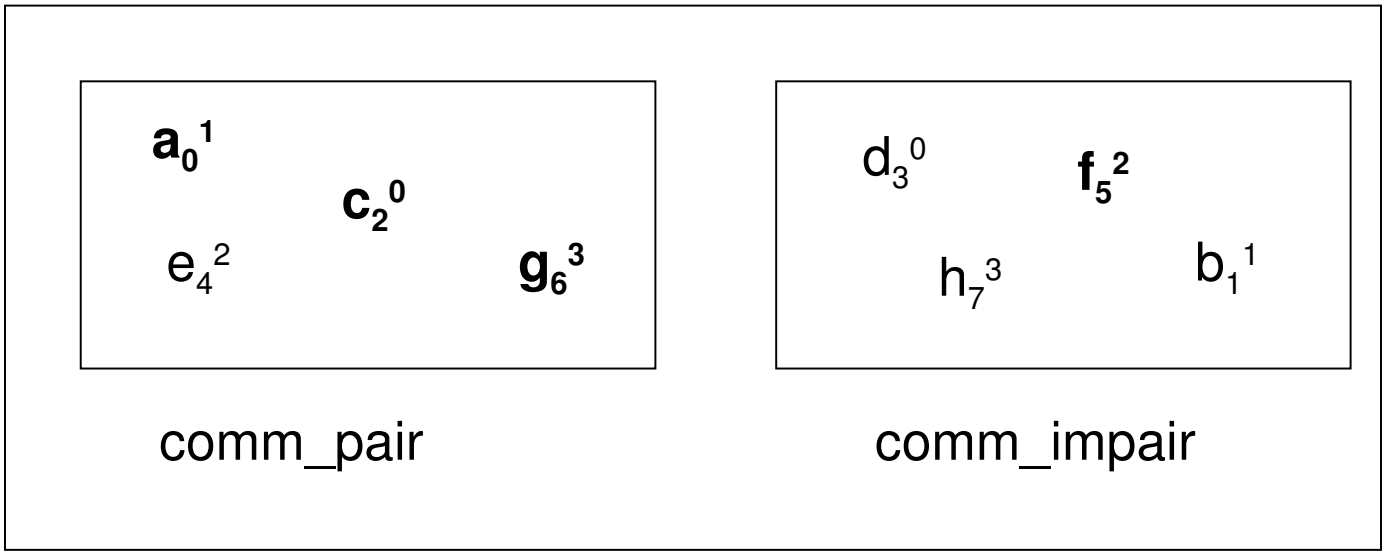
MPI_COMM_WORLD



Construction d'un communicateur avec MPI_COMM_SPLIT

Construction de communicateurs pair et impair

rang	0	1	2	3	4	5	6	7
processus	a	b	c	d	e	f	g	h
couleur								
clef	1	1	0	0	4	5	6	7



Construction de communicateurs pair et impair avec MPI_COMM_SPLIT

```
program CommPairImpair (exemple IDRIS)
  include 'mpif.h'
! Déclarations
  integer, parameter        :: m=4
  integer                   :: clef,couleur,comm,rang,nb_procs,ier
  real,dimension(m)         ::
  ....
  call MPI_INIT(ier)
  call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,ier)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,ier)

! Initialisation du vecteur a
  a=0.
  if (rang == 2) a=2.
  if (rang == 3) a=3.
! Enregister le rangs des processus pairs
  dim_rangs_pair=int((nb_procs+1)/2)
  allocate(rangs_pair(dim_rangs_pair))
  do iproc=0,nb_procs-1,2
    i=i+1
    rangs_pair(i)=iproc
  enddo
```

Création des variables clef(rang) et couleur (groupe)

```
if (mod(rang,2) == 0 ) then
   ! Couleurs et clefs de processus pairs
   couleur=0
   if (rang == 2 ) then
     clef=0
   else if (rang == 0) then
     clef=1
   else
     clef=rang
   endif
else
   ! Couleur des processus impairs
   couleur=1
   if (rang == 3) then
         clef=1
   else
         clef=rang
  endif
```

```
!  Créer les communicateurs pairs et impairs
  call MPI_COMM_SPLIT(MPI_COMM_WORLD,couleur,&
                clef,comm,ier)


! Le processus 0 de chaque communicateur diffuse son
! message à chaque processus de son groupe
  call MPI_BCAST(a,m,MPI_REAL,0,comm,ier)

! Destruction des communicateurs
  call MPI_COMM_FREE(comm,ier)
  print*,'processus ',rang, '  a = ',a

! Fin MPI
  call MPI_FINALIZE(ier)

end program      CommPairImpair
```

Topologies

- **Motivations**
  - Utilisation fréquente des matrices et de la décomposition de domaine dans les algorithmes parallèles
  - Correspondance entres les grilles de données et de processus favorise
    - Les performances
    - La facilité d'implémentation
    - Clareté des implémentations
- **2 types de topologies**
  - Cartésienne: grille régulière
  - Graphe de processus: grille irrégulières

Topologie cartésienne

- Grille de processus définie par:
  - Sa dimension
  - Sa périodicité
  - Type de numérotation dans la grille
- Chaque processus
  - Est identifié par des coordonnées dans la grille
  - Peut récupérer le rang de ses voisins dans un direction donnée

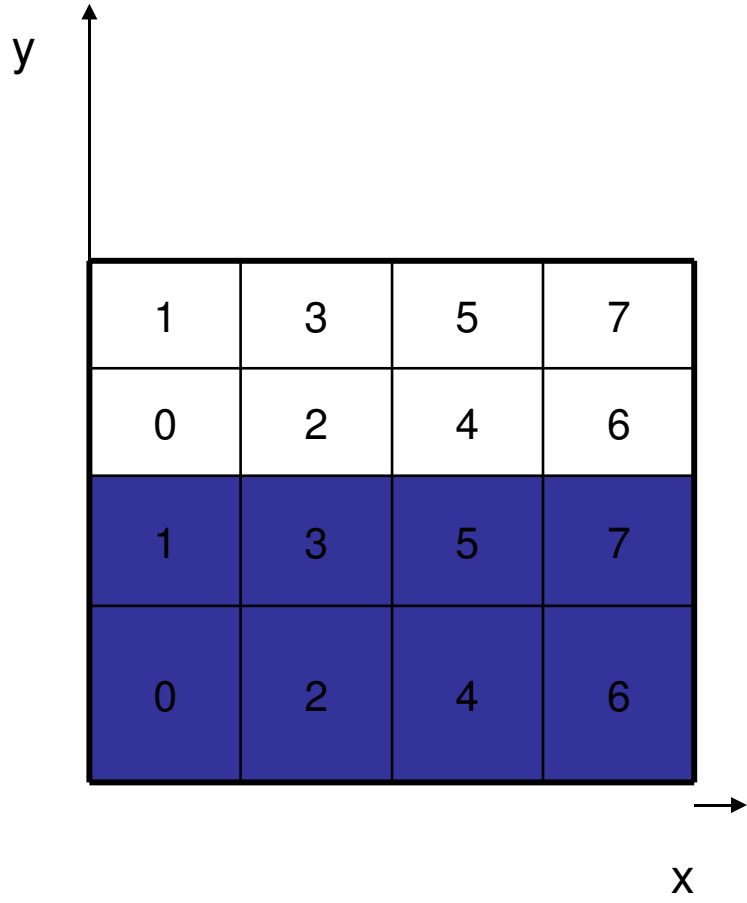
Définition d'une topologie cartésienne

- Appel **par tous les processus** d'un communicateur donné du sous-programme MPI, MPI_CART_CREATE

```
integer,intent(in) :: comm_ancien,ndims
integer, dimension(ndims),intent(in) :: dims
logical,dimension(ndims),intent(in) :: periods
logical,intent(in) :: reorganisation

integer,intent(out) :: comm_nouveau,ierr

call MPI_CART_CREATE(comm_ancien,ndims,dims,
     periods,reorganisation,comm_nouveau,ier)
```

Topologie cartésienne 2D

```
include ‘mpif.h’
integer                          ::  comm2_D,ierr
integer,parameter                 :: ndims=2
integer,dimension(ndims) ::dims
logical,dimension(ndims) :: periods
logical                          :: reorganisation
...
dims(1)=4
dims(2)=2
periods(1)=.false.
periods(2)=.true.
reorganisation=.true.
…
call MPI_CART_CREATE(MPI_COMM_WORLD,
          ndims,dims,periods,reorganisation,
          comm_2d,ierr)
```



Topologie cartésienne

- MPI_DIMS_CREATE() : retourne le nombre de processus dans chaque dimension en fonction du nombre de processus
- call MPI_DIMS_CREATE(nnoeuds,ndims,dims,ier)

MPI a le choix pour les valeurs de dims qui sont à zéro

dims en entrée	**call MPI_DIMS_CREATE**	dims en sortie
(0,0)	(8,2,dims,ier)	(4,2)
(0,0,0)	(16,3,dims,ier)	(4,2,2)
(0,4,0)	(16,3,dims,ier)	(2,4,2)
(0,3,0)	(16,3,dims,ier)	error

Topologie cartésienne

- MPI_CART_RANK() retourne le rang du processus

```
integer,intent(in)                    :: comm_cart
integer,dimension(ndims),intent(in)   :: coords
integer,intent(out)                   :: rang, ierror
call MPI_CART_RANK(comm_cart,coords,rang,ierror)
```

- MPI_CART_COORDS() retourne les coordonnées d'un processus de rang donné dans la grille

```
integer,intent(in)                    :: comm_cart
integer,intent(in)                    :: rang
integer,dimension(ndims),intent(out)  :: coords
integer, intent(out)                  :: ierror
call MPI_CART_COORDS(comm_cart,rang,coords,ierror)
```

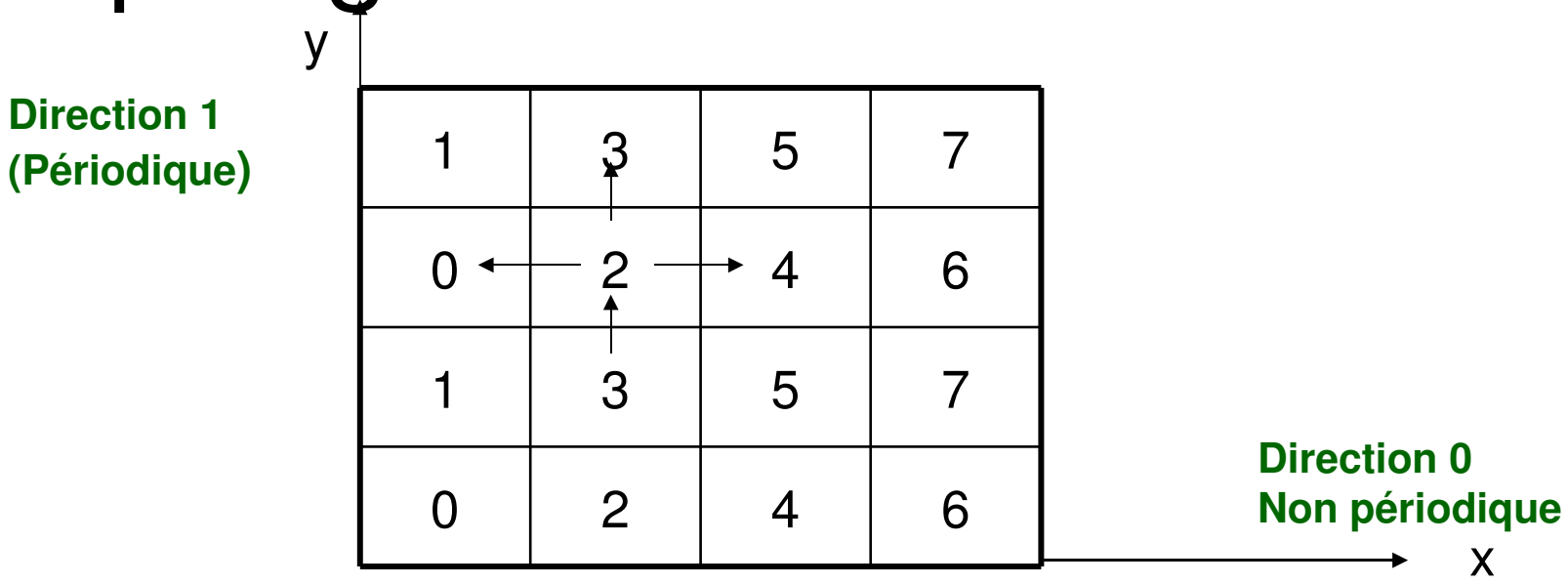
Topologie cartésienne: voisins

- MPI_CART_SHIFT retourne le rang des processus voisins dans une direction donnée du processus appelant:

```
integer,intent(in)  :: comm_cart,direction, pas
integer,intent(out)  :: rang_preced,rang_suivant,ierror
call MPI_CART_SHIFT(comm_cart,direction,pas,
     rang_preced,rang_suivant,ierror)
```

  - **direction** : axe de déplacement (0 1ère dimension, 1 deuxième,2 …)
  - **pas** : pas de déplacement

Topologie cartésienne: **voisins**



call MPI_CART_SHIFT(comm_cart,**0,1**,rg_gauche,rg_droit,ier)

…

Pour le processus 2, rg_gauche=0 et rg_droit=4

call MPI_CART_SHIFT(comm_cart,**1,1**,rg_bas,rg_haut,ier)

…

Pour le **processus 2, rg_bas=3** et **rg_haut=3**

Subdiviser une topologie cartésienne

Subdiviser une topologie cartésienne **2D** (ou **3D**) revient à créer autant de communicateurs qu'il y a de **lignes** ou de **colonnes** (resp de **plans**) dans la grille initiale.

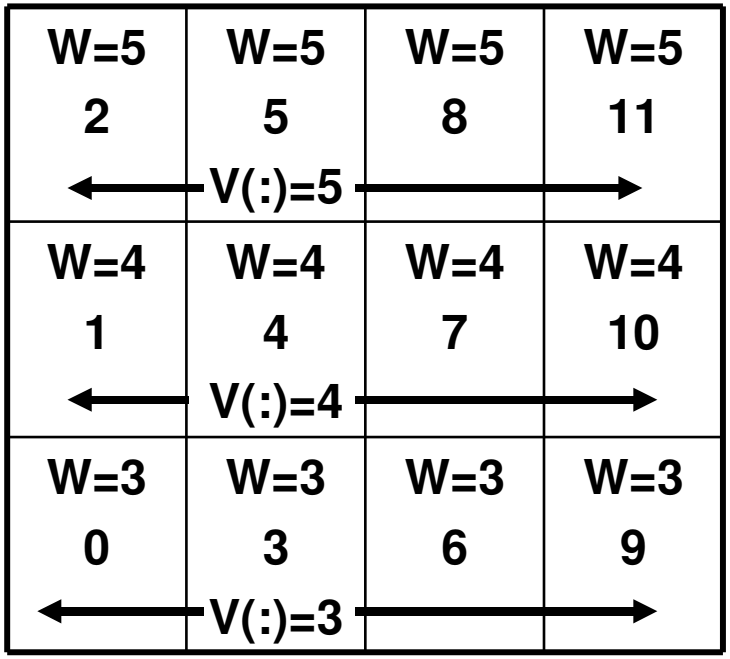
- Intérêt:
  - Effectuer des opérations collectives restreintes à une ligne ou une colonne de processus en 2D(ou à des plans dans le cas 3D)
- Deux méthodes:
  - Utiliser le sous-programme **MPI_COMM_SPLIT()**
  - Utiliser le sous-programme **MPI_CART_SUB()** prévu à cet effet

```
  program commCartSub
  implicit none
  include ‘mpif.h’
! Declacrations
  integer,parameter            :: NDim=2, m=4
  integer                      :: Comm2d,Comm1D,rang,ier
  integer,dimension(NDim) :: Dim,Coord
  logical,dimension(NDim)  :: Periode,Subdivision
  logical                      :: ReOrdonne
  real,dimension(m)            ::V=0.
  real                         :: W=0.

! Initialisation
  call MPI_INIT(ier)
  …
! Création de la grille 2D initiale
  Dim(1)=4; Dim(2)=3
  Periode=.false.  ;  ReOrdonne=.false.
  call MPI_CART_CREATE(MPI_COMM_WORLD,NDim,Dim,&
            Periode,ReOrdonne,Comm2D,ier)
  call MPI_COMM_RANK(Comm2D,rang,ier)
  call MPI_CART_COORDS(Comm2D,rang,NDim,Coord,ier)
```

```
! Initialisation du vecteur V
  if (Coord(1) == 1) V=real(rang)

! Chaque ligne de la topologie est une topologie cartésienne 1D
  Subdivision(1)=.true.
  Subdivision(2)=.false.

! Subdivision de la grille cartésienne 2D
  call MPI_CART_SUB(Comm2D,Subdivision,Comm1D,ier)

! Les processus de la colonne 2 distribue le vecteur V aux
!  processus de leur ligne
  call MPI_SCATTER(V,1,MPI_REAL,W,1,MPI_REAL,1,Comm1D,ier)

  print*,' Rang : ',rang, 'Coordonnées : ',Coord, 'W = ',W

! Fin MPI
  call MPI_FINALIZE(ier)

  end program CommCartSub
```

Subdivision d'une topologie cartésienne

V(:)=0 2 W=0	V(:)=5 5 W=0	V(:)=0 8 W=0	V(:)=0 11 W=0
V(:)=0 1 W=0	V(:)=4 4 W=0	V(:)=0 7 W=0	V(:)=0 10 W=0
V(:)=0 0 W=0	V(:)=3 3 W=0	V(:)=0 6 W=0	V(:)=0 9 W=0

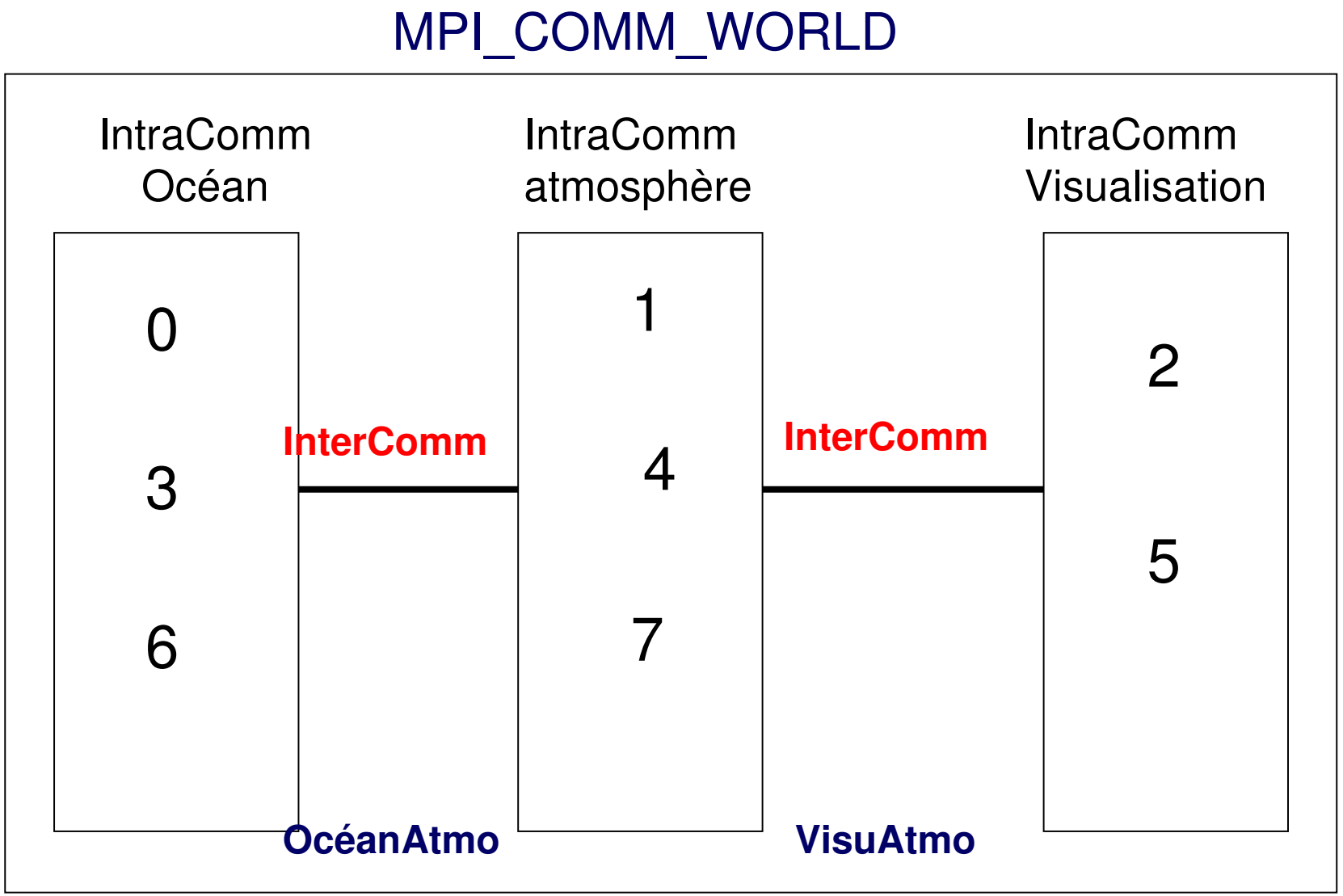


Représentation initiale d'un tableau V dans la grille 2D et
Représentaion finale après la distribution (MPI_SCATTER) de
Celui-ci sur la grille 2D dégénérée

Intra et Inter Communicateur

- Tous les communicateurs vus jusqu'à présent sont des **intracommunicateurs**
- Les processus de deux **intracommunicateurs distincts** ne peuvent communiquer entre eux que si il existe un **lien de communication entre ces deux intracommunicateurs**
- Un intercommunicateur est un communicateur qui établit un lien entre deux intracommunicateurs
- Le sous-programme **MPI_INTERCOMM_CREATE()** permet de construire un **intercommunicateur**
- Dans **MPI-1**, seules les communications point à point dans un intercommunicateur sont permises. Cette limitation disparaît en **MPI-2**

Intercommunicateur: Couplage Océan-atmosphère



Couplage Océan/Atmosphère

```
program OceanAtmosphere
impicit none
include ‘mpif.h’

integer,parameter
integer :: ier,nb_procs,rang,nbIntraComm,couleur,IntraComm

! Initialisation MPI
call MPI_INIT(ier)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,ier)
call MPI_COMM_RANK(MPI_COMM_WORLD,rang,ier)

! Constructions des trois intracommunicateurs
nbIntraComm=3
couleur=mod(rang,nbIntraComm)
call MPI_COMM_SPLIT(MPI_COMM_WORLD,couleur,rang,&
        IntraComm,ier)
```

```
! Construction des deux Intercommunicateurs et calcul
  select case (couleur)
    case(0)
    !  Intercommunicateur OceanAtmosphere
    ! Le group 0 communique avec le groupe 1
    call MPI_INTERCOMM_CREATE(IntraComm,0,MPI_COMM_WORLD,&
            1,tag1,CommOceanAtmo,ier)
    call ocean(IntraComm,CommOceanAtmo)
   case(1)
   ! Intercommunicateur OceanAtmosphere
    ! Le group 1 communique avec le groupe 0
    call MPI_INTERCOMM_CREATE(IntraComm,0,MPI_COMM_WORLD,&
            0,tag1,CommOceanAtmo,ier)
    call atmosphere(IntraComm,CommOceanAtmo,CommVisuAtmo)
    case(2)
    ! Intercommunicateur CommVisuAtmo pour que le groupe 2
    ! Communique avec le group 1
    call MPI_INTERCOMM_CREATE(IntraComm,0,MPI_COMM_WORLD,&
            1,tag2,CommVisuAtmo,ier)
    call visualisation(IntraComm,CommVisuAtmo)
  end select
```

Introduction à MPI

MPI_2

Ce qui manque à MPI-1

- Gestion dynamique des processus
- Entrées/sorties parallèles
- Interfaçage Fortran90 et C++
- Extention des communications collectives aux intercommunicateurs
- Les communications mémoire à mémoire
- Possibilité de définir des interfaces externes (gstion des processus légers…)
- Possibilité de gérer des applications temps réels multiprocesseurs

MPI-2 offre les 6 premières possibilités (voir IDRIS)

Implémentation de MPI-2

	LAM	MPICH	Fujitsu	NEC	IBM
Gestion dynamique	I	NI	I	I	NI
Copie mém à mém	PI	NI	I	I	I
Interfaçage F90 C++	NI	PI	I	I	NI
MPI-IO	PI	PI	I	I	I
Types dérivés	PI	PI	I	I	I

I : implémenté
NI: non implémenté
PI: partiellement implémenté

Ce transparent est tiré du cours de l'IDRIS, voir cours MPI-2

E/S Parallèles- MPI-IO
Introduction

- Les applications qui font des calculs volumineux font en général également de nombreuses entrées-sorties.
- Le traitement de ces entrées-sorties ont une part importante dans le temps globale l'application
- L'optimisation des E/S d'une application parallèle se fait:
  - Par leur parallèlisation
  - Et /ou leur recouvrement par des calculs
  - Et/ou par des opérations spécifiques prises en charge par le système

MPI-IO: Lectures/Ecriture

- Appels explicites à des sous-programmes
- Trois propriétés des accès aux fichiers:
  - **Positionnement**:
    - Explicite: par ex, nb octets depuis le début du fichier
    - Implicite :pointeur
      - Individuel à chaque processus
      - Commun à tous les processus
  - **Synchronisation**: bloquant ou non-bloquant
  - **Regroupement**: collectif ou propre à un ou plusieurs processus

E/S parallèles

- Qu'est ce ?
  - E/S effectuées par plusieurs tâches ou processus
    - À partir d'un même espace mémoire (calculateur mémoire distribuée)
    - À partir d'espaces mémoire séparés (cluster)
- Avantages
  - **Éviter de créer des goulets d'étranglement**
  - Techniques au niveau de la programmation permettant de gérer **les lectures/écritures asynchrones**
  - **Performances**: opérations spécifiques prises en charge par le système

E/S parallèles:interface MPI-2

- Basée sur deux concepts de MPI
  - **Communicateur**
  - **Type dérivé MPI**
- Fermeture et ouverture de fichier sont des **opérations collectives associées à un communicateur**
- Chaque **opération de transfert** est associée à **un type de donnée**
  - **Type de base MPI**
  - **Type dérivé MPI**: transfert d'un zone discontinue de la mémoire vers un stockage continu sur fichier ou inversement

MPI-2: quelques définitions

- **Type élémentaire de données:** unité de données pour calculer les positionnements permettant d' accéder aux données, **type MPI de base** ou **type dérivé MPI**
- **Motif:** masque qui constitue la base du partitionnement d'un fichier entre processus
- **Vue**: ensemble des données visibles d'un fichier . Ensemble ordonné de types élémentaires
- **Position**(offset): position dans le fichier exprimée en nombre de type élémentaires, relativement à la vue courante.

MPI-2: quelques définitions

- **Descripteur**(file handle): objet caché créé à l'ouverture et détruit à la fermeture du fichier
- **Pointeurs** tenu à jour automatiquement par MPI (associé à chaque file handle)
  - **Individuel :** propre à chaque processus ayant ouvert le fichier
  - **Partagé :** commun à chaque processus ayant ouvert le fichier

MPI-2: lectures / écritures

- **Appels explicites à des sous-programmes**
- **Accès définis par 3 propriétés**
  - **Positionnement**:
    - **Explicite**: par exemple, nombre d'octets depuis le début du fichier
    - **Implicite**: pointeur géré par le système
      - **Individuel** à chaque processus
      - **Partagé** par tous les processus du communicateur
  - **Synchronisation**: accès bloquants ou non bloquants
  - **Regroupement**: collectifs ou propres à un ou plusieurs processus

MPI-2: fonctions de base

- **MPI_FILE_OPEN():** associe un descripteur à un fichier
- **MPI_FILE_SEEK():** déplace le pointeur de fichier vers une position donnée
- **MPI_FILE_READ():** lit à partir de la position courante une quantité de données(pointeur individuel)
- **MPI_FILE_WRITE():** écrit une quantité de données à partir de la position courante (pointeur individuel)
- **MPI_FILE_CLOSE():** élimine le descripteur
- **MPI_FILE_SYNC():** force l'écriture sur disque des buffers associés à un descripteur
- …

Lectures/ Écritures individuelles

- **Propres à un ou plusieurs processus** du communicateur dans lequel a été ouvert le fichier
- Via un **déplacement explicite**
  - MPI_FILE_READ_AT,MPI_FILE_WRITE_AT
- Via un **déplacement implicite**: **pointeur**
  - **Individuel** : propre à chaque processus
    - MPI_FILE_READ,MPI_FILE_WRITE
  - **Partagé**: commun à tous les processus du communicateur dans lequel le fichier a été ouvert
    - MPI_FILE_READ_SHARED,MPI_FILE_WRITE_SHARED

E/S individuelles
Déplacements implicites individuels

- **Un pointeur individuel par fichier et par processus** est géré automatiquement par le système
- **Les pointeurs partagés** ne sont pas modifiés
- Pour un processus et un fichier donné, à chaque accès, **le pointeur est positionné sur le type élémentaire suivant**

E/S individuelles
Déplacements implicites partagés

- Il existe **un et un seul pointeur partagé par fichier**
- Il est **commun à tous les processus** du communicateur dans lequel a été ouvert le fichier
- Tous les processus, utilisant le pointeur partagé d'un fichier doivent utiliser la **même vue du fichier**
- Dans le cas d'E/S individuelles, il faut **gérer explicitement l'ordonnancement des processus**
- **Les pointeurs individuels** ne sont jamais modifiés
- Après chaque accès, le pointeur est **positionné sur le type élémentaire suivant**

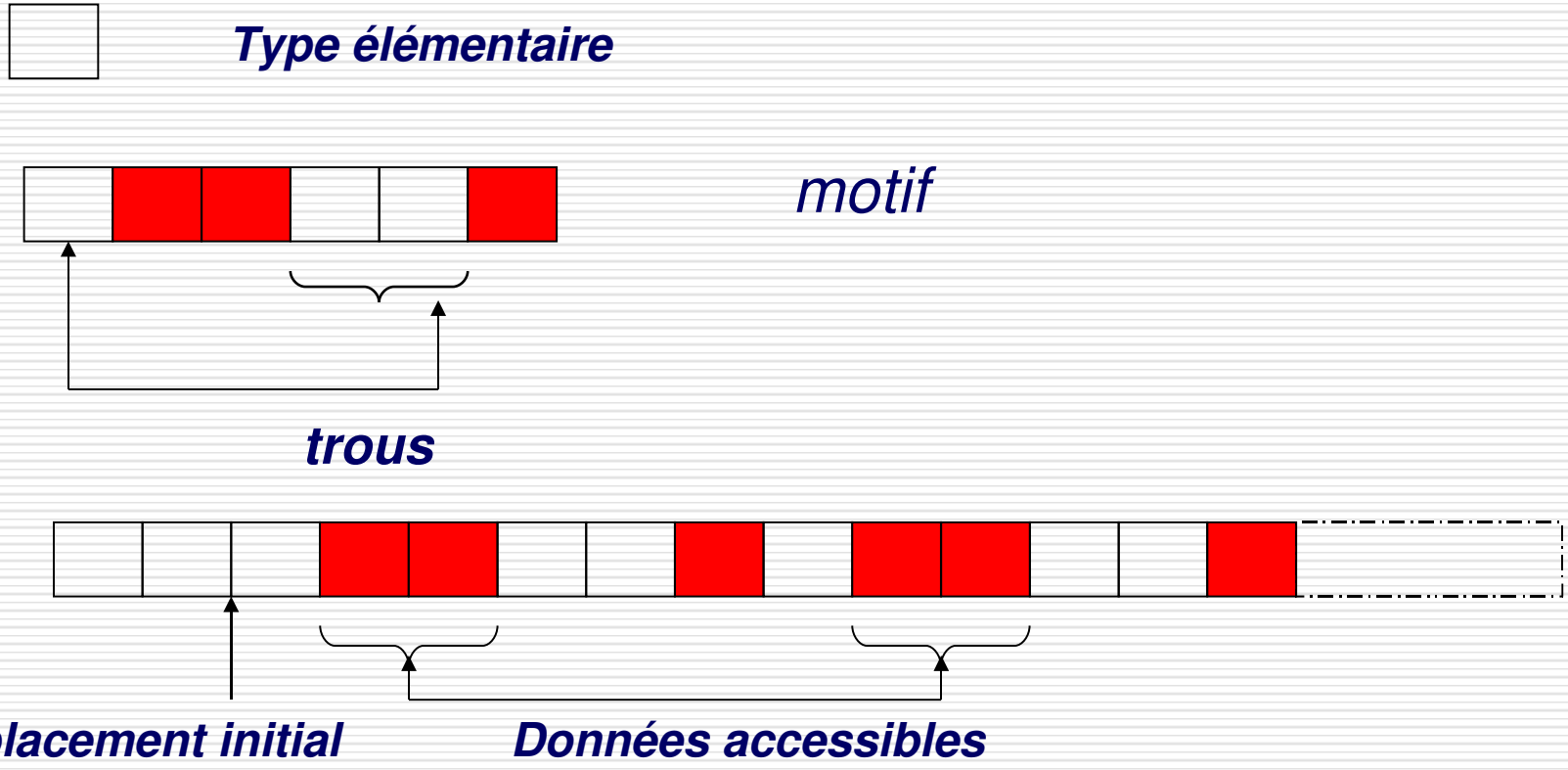
Lectures/ Écritures collectives

- **Tous les processus** du communicateur dans lequel un fichier est ouvert **participent aux opérations collectives**
- les opérations collectives sont généralement **plus performantes** car elles permettent davantage d'optimisation automatiques
- Les opérations collectives sont **effectuées dans l'ordre des rangs des processus**. Le traitement est **déterministe**

MPI-IO: les vues

- Mécanismes permettant de décrire les **zones accédées dans un fichier**
- Construites à partir des **types dérivés MPI**
- **Chaque processus a sa propre vue d'un fichier** définit par:
  - **Déplacement initial**
  - **Type élémentaire de données**
  - **Motif** : une vue est une suite de motifs
- Il est possible de définir **des trous** dans la vue de façon à ne pas tenir compte de certaines parties
- Des processus peuvent avoir **des vues différentes du fichier** de façon à accéder à des parties complémentaires (E/S individuelles)

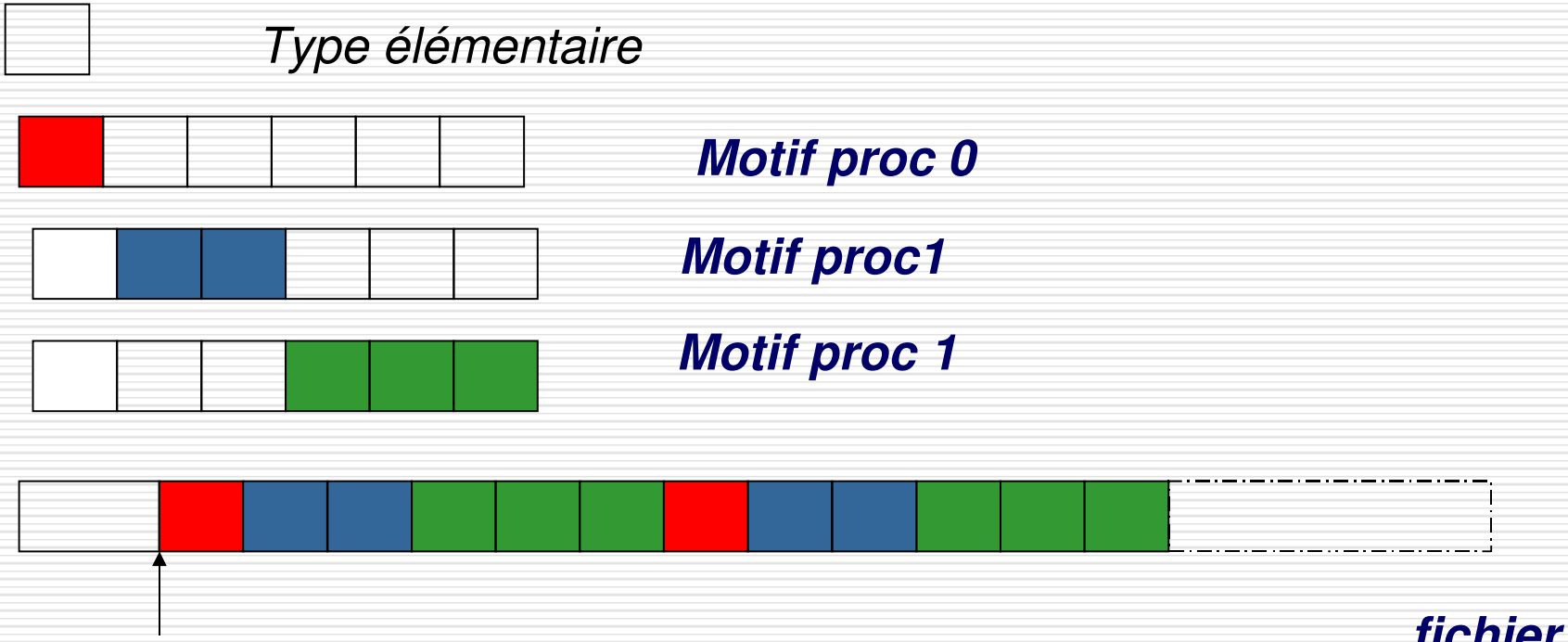
MPI-IO: définition d'une vue



```
program vue
...
! Ouverture du fichier
call MPI_FILE_OPEN(MPI_COMM_WORLD, ″ donnees.dat ″,&
       MPI_RDONLY,descripteur,ier)
....
! Definition d'une vue
deplacement(1)=1
deplacement(2)=4
deplacement(3)=0
longueur(1)=1
longueur(2)=2
longueur(3)=3

call MPI_TYPE_INDEXED(3,longueur,deplacement,MPI_INTEGER,&
       motif,ier)
call MPI_COMMIT(motif,ier)
```

```
deplacement_initial=2*nb_octets_entier
call MPI_FILE_SET_VIEW(descripteur,deplacement_initial,&
      MPI_INTEGER,˝ native ˝,MPI_INFO_NULL, motif,ier)
call MPI_FILE_GET_VIEW(descripteur,deplacement_initial,&
       type-elem,motif,representation,ier)

call MPI_FILE_READ(descripteur,valeurs,nb_valeurs,&
      MPI_INTEGER,statut,ier)

call MPI_FILE_CLOSE(descripteur,ier)

…
```

MPI-IO: définition d'une vue



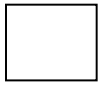
Motifs différents suivant les processus

```
program vues_multiples
...
! Ouverture du fichier
call MPI_FILE_OPEN(MPI_COMM_WORLD, ˝ donnees.dat ˝,&
     MPI_RDONLY,descripteur,ier)
! Definition des vues
if (rang == 0) then
   deplacement(1)=0
   deplacement(2)=4
   longueur(1)=2
   longueur(2)=0
else
   deplacement(1)=2
   deplacement(2)=0
   longueur(1)=2
   longueur(2)=0
endif
call MPI_TYPE_INDEXED(2,longueur,deplacement,MPI_INTEGER,&
        motif,ier)
call MPI_COMMIT(motif,ier)
```

```
deplacement_initial=0
call MPI_FILE_SET_VIEW(descripteur,deplacement_initial,&
    MPI_INTEGER,˝ native ˝,MPI_INFO_NULL, motif,ier)

call MPI_FILE_READ(descripteur,valeurs,nb_valeurs,&
      MPI_INTEGER,statut,ier)

call MPI_FILE_CLOSE(descripteur,ier)

…
```

0

Déplacement initial

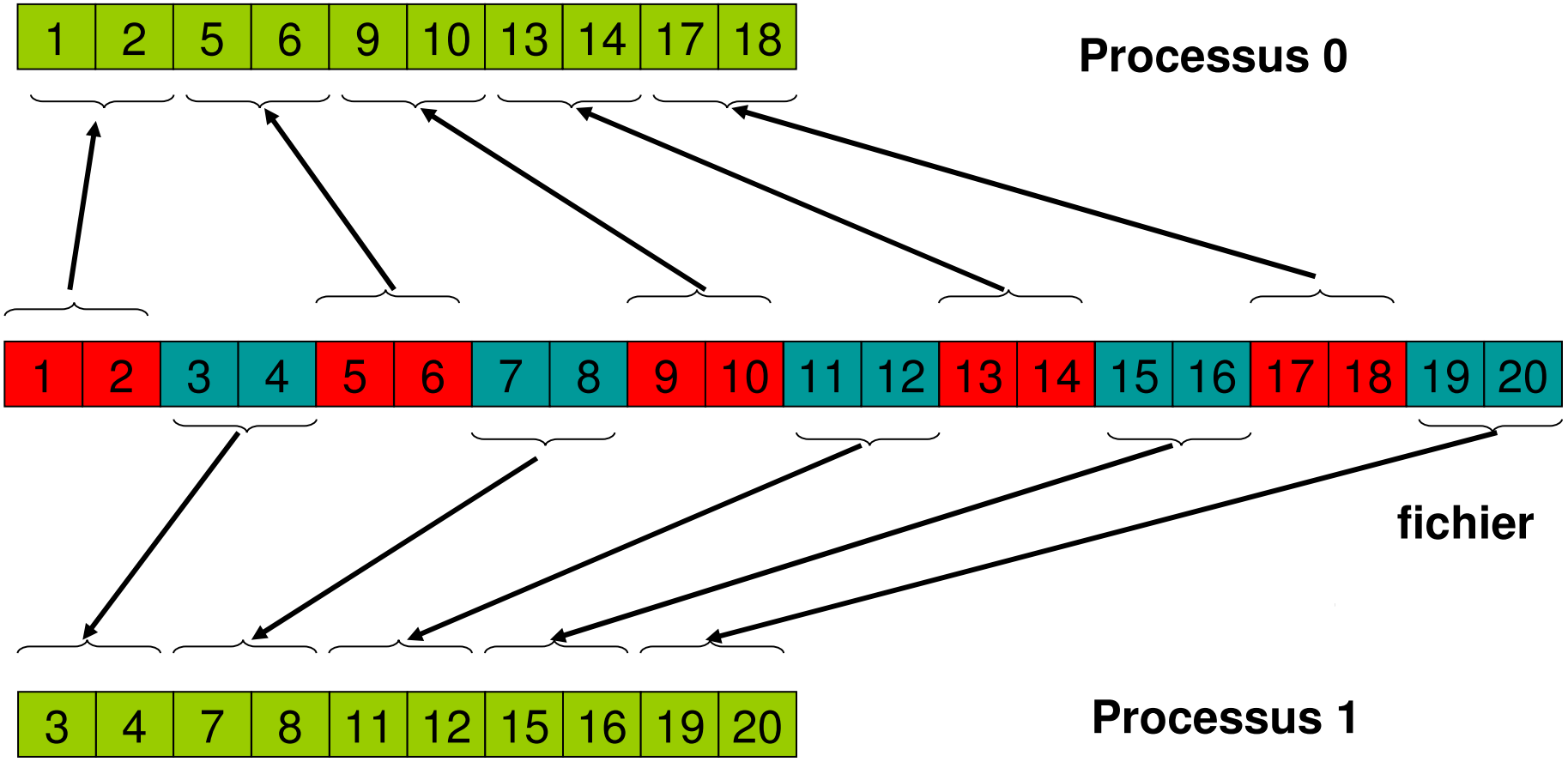


Type élémentaire : MPI_INTEGER

motif proc 0

motif proc 1

Motifs employés dans le programme ˝vues_multiples˝



%**mpirun –np 2 vues_multiples**

Lecture processus 0: 1 2 5 6 9 10 13 14 17 18

Lecture processus 1: 3 4 7 8 11 12 15 16 19 20

Positionnement des pointeurs dans un fichier

- **Connaître la valeur courante** du pointeur
  - **Individuel**: **MPI_FILE_GET_POSITION**()
  - **Partagé:** **MPI_FILE_GET_POSITION_SHARED**()
- **Positionner explicitement** le pointeur:
  - **Individuel**: **MPI_FILE_ SEEK**()
  - **Partagé:** **MPI_FILE_SEEK_SHARED**()
- **Trois modes** pour fixer la valeur d'un pointeur
  - **Une valeur absolue**: **MPI_SEEK_SET**
  - **Une valeur relative** : **MPI_SEEK_CUR**
  - **À la fin d'un fichier**: **MPI_SEEK_END**

E/S non bloquantes

- Implémentées **suivant le modèle utilisé pour les communications**
- Un **accès asynchrone** doit donner lieu à un **test de complétude explicite** ultérieur (MPI_TEST,MPI_WAIT, …)

MPI-2 : conclusion et conseils

- **L'interface MPI-IO** permet de masquer aux utilisateurs des opérations complexes et d'implémenter de façon transparente des optimisations particulières aux machines cibles
- **Quelques conseils**:
  - Il faut **privilégier la forme collective** des opérations
  - **L'utilisation implicite de pointeurs** individuels ou collectifs offre une interface de plus haut niveau
  - Comme pour les messages, **l'asynchronisme est une voie privilégiée d'optimisation** (s'assurer de la validation de l'algorithme en mode synchrone)
  - …

Quelques références

- http://ci-tutor.ncsa.uiuc.edu/login.php
- http://www.idris.fr