INSTITUT DU
DÉVELOPPEMENT ET DES
RESSOURCES EN
INFORMATIQUE
SCIENTIFIQUE

MPI

Dimitri Lecas - Rémi Lacroix - Serge Van Criekingen - Myriam Peyrounette

CNRS — IDRIS

v5.3 06 juin 2023

Plan I

Plan II

Plan III

Plan IV

Introduction

A propos

Ce document est mis à jour régulièrement. La version la plus récente est disponible sur le site Web de l'IDRIS : http://www.idris.fr/formations/mpi/

- IDRIS
  Institut du développement et des ressources en informatique scientifique
  Rue John Von Neumann
  Bâtiment 506
  BP 167
  91403 ORSAY CEDEX
  France
  http://www.idris.fr

Introduction

Parallélisme

L'intérêt de faire de la programmation parallèle est :

- De réduire le temps de restitution ;
- D'effectuer de plus gros calculs ;
- D'exploiter le parallélisme des processeurs modernes (multi-coeurs, multithreading).

Mais pour travailler à plusieurs, la coordination est nécessaire. MPI est une bibliothèque permettant de coordonner des processus en utilisant le paradigme de l'échange de messages.

Modèle de programmation séquentiel

- le programme est exécuté par un et un seul processus ;
- toutes les variables et constantes du programme sont allouées dans la mémoire allouée au processus ;
- un processus s'exécute sur un processeur physique de la machine.

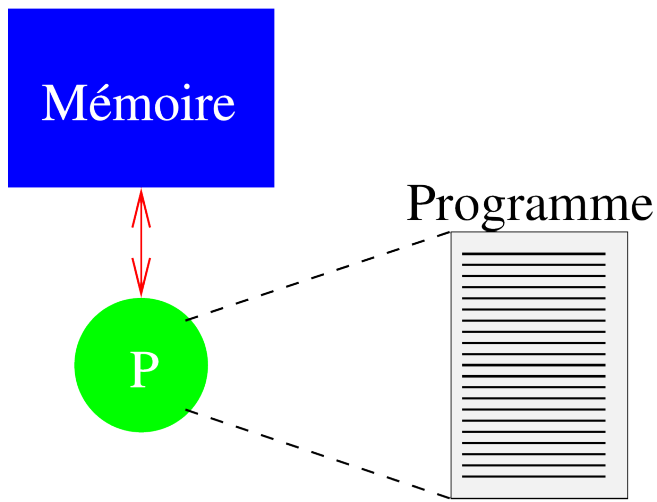


Figure 1 – Modèle de programmation séquentiel

Introduction

Modèle de programmation par échange de messages

- le programme est écrit dans un langage classique (Fortran, C, C++, etc.) ;
- toutes les variables du programme sont privées et résident dans la mémoire locale allouée à chaque processus ;
- chaque processus exécute éventuellement des parties différentes d'un programme ;
- une donnée est échangée entre deux ou plusieurs processus via un appel, dans le programme, à des sous-programmes particuliers.

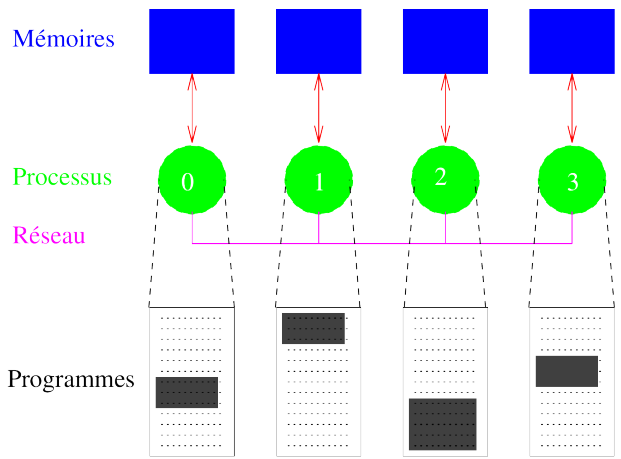


Figure 2 – Modèle de programmation par échange de messages

Concepts de l'échange de messages

Si un message est envoyé à un processus, celui-ci doit ensuite le recevoir

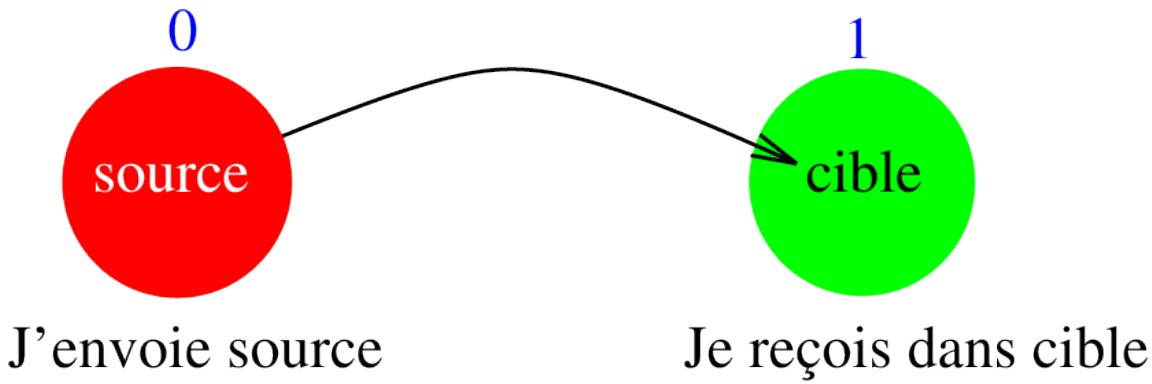


Figure 3 – échange d'un message

Introduction

Constitution d'un message

- Un message est constitué de paquets de données transitant du processus émetteur au(x) processus récepteur(s)
- En plus des données (variables scalaires, tableaux, etc.) à transmettre, un message doit contenir les informations suivantes :
  - l'identificateur du processus émetteur ;
  - le type de la donnée ;
  - sa longueur ;
  - l'identificateur du processus récepteur.

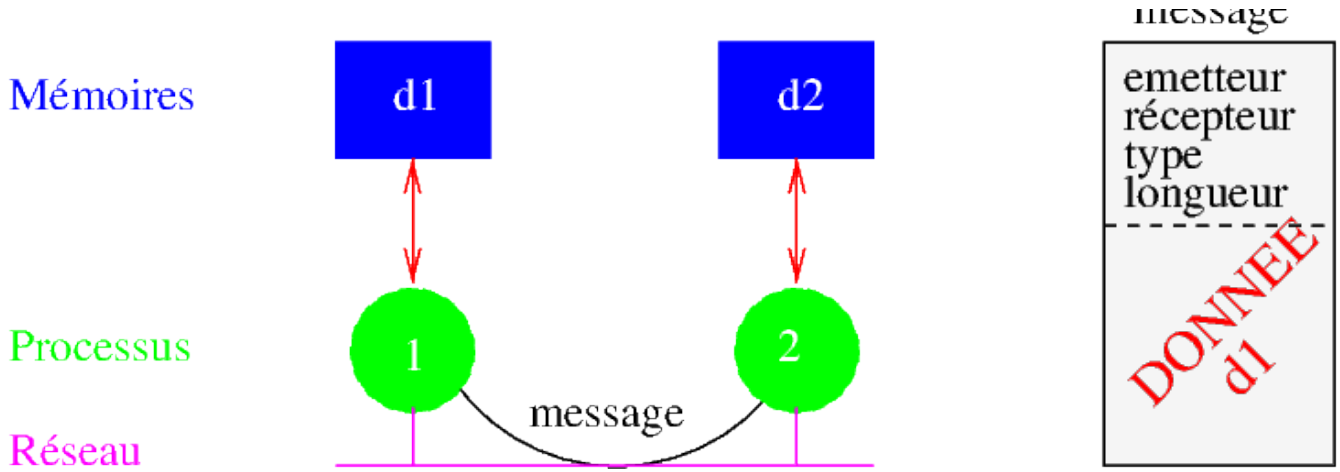


Figure 4 – Constitution d'un message

Introduction

Environnement

- Les messages échangés sont interprétés et gérés par un environnement qui peut être comparé à la téléphonie, au courrier postal, à la messagerie électronique, etc.
- Le message est envoyé à une adresse déterminée
- Le processus récepteur doit pouvoir classer et interpréter les messages qui lui ont été adressés
- L'environnement en question est MPI (Message Passing Interface). Une application MPI est un ensemble de processus autonomes exécutant chacun leur propre code et communiquant via des appels à des sous-programmes de la bibliothèque MPI

Architecture des supercalculateurs

La plupart des supercalculateurs sont des machines à mémoire distribuée. Ils sont composés d'un ensemble de nœud, à l'intérieur d'un nœud la mémoire est partagée.

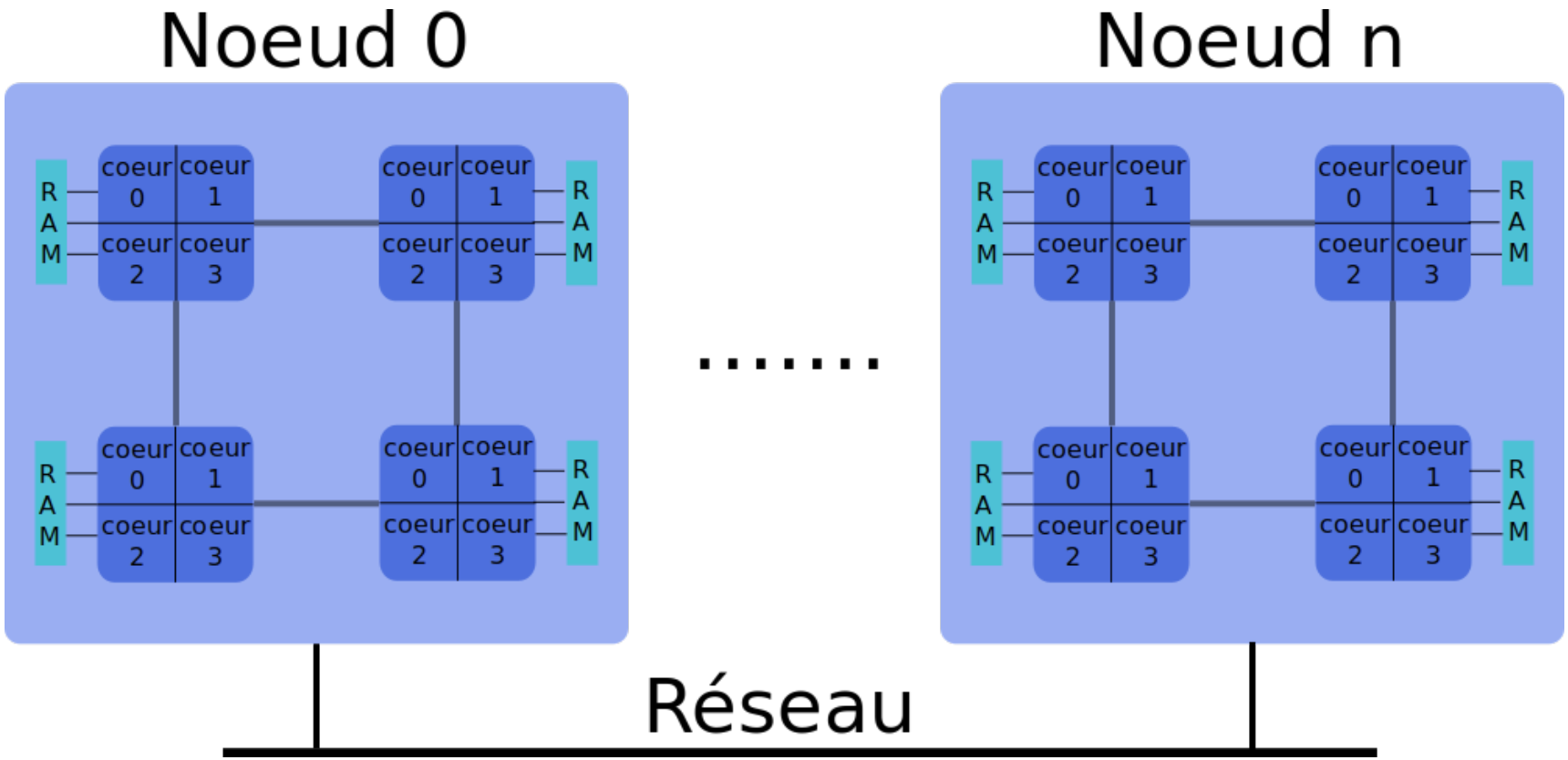


Figure 5 – Architecture des supercalculateurs

Introduction

Jean Zay

- 2 140 nœuds
- 2 processeurs Intel Cascade Lake (20 cœurs) à 2,5 Ghz par nœud
- 4 GPU Nvidia V100 par nœud (sur 612 nœud)
- 85 600 cœurs
- 410 To (192 Go par nœud)
- 26 Pflop/s crête
- 15,6 Pflop/s (linpack)

MPI vs OpenMP

OpenMP utilise un schéma à mémoire partagée, tandis que pour MPI la mémoire est distribuée.

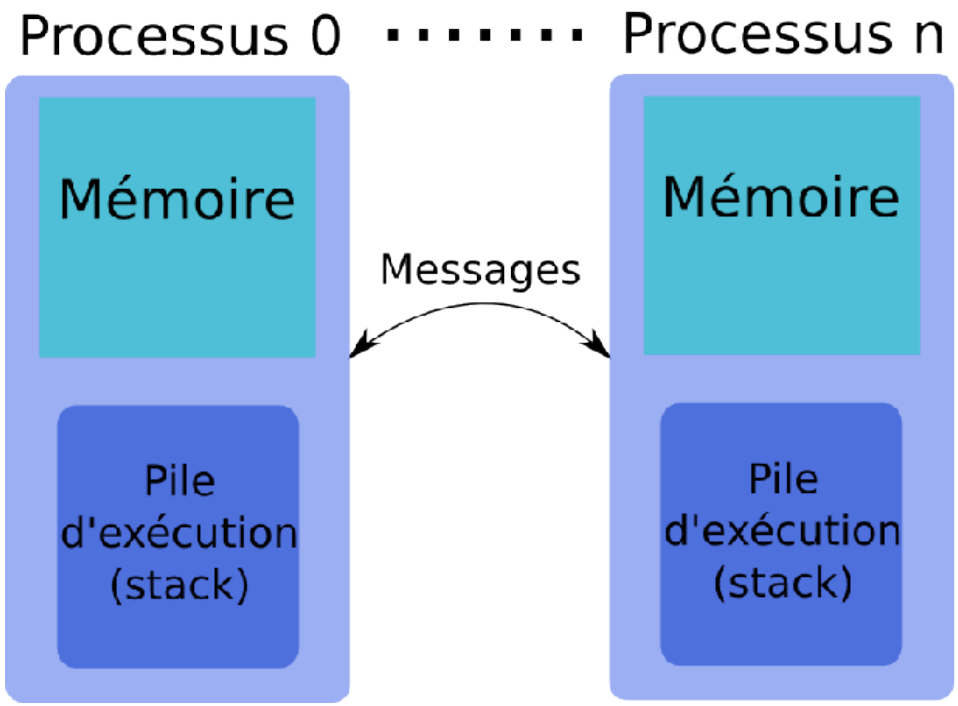


Figure 6 – Schéma MPI

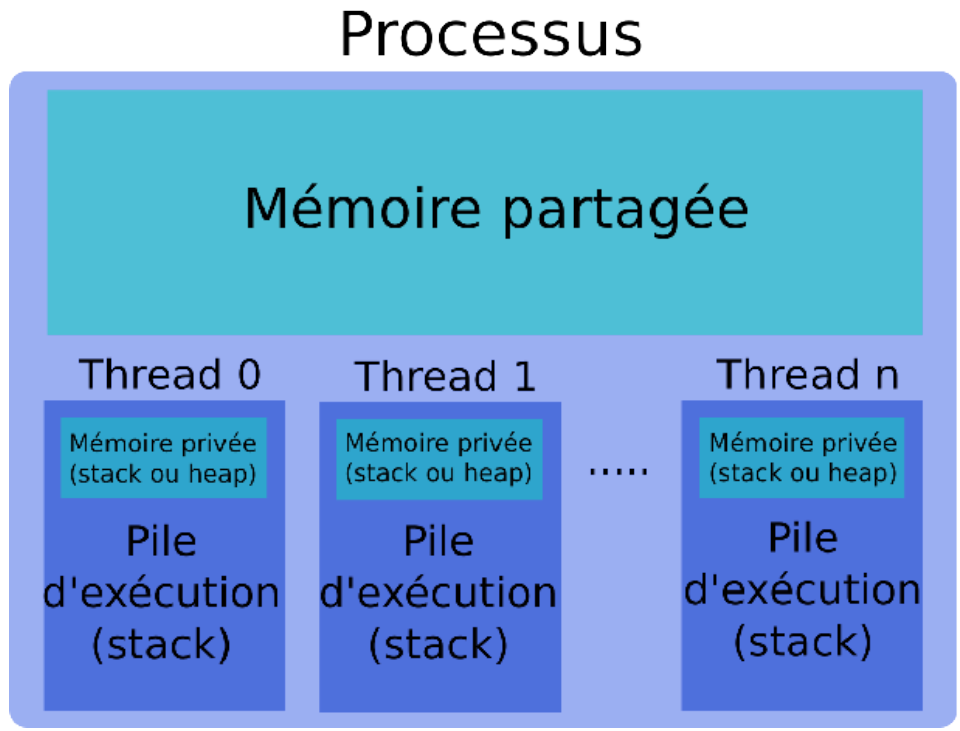


Figure 7 – Schéma OpenMP

Décomposition de domaine

Un schéma que l'on rencontre très souvent avec MPI est la décomposition de domaine. Chaque processus possède une partie du domaine global, et effectue principalement des échanges avec ses processus voisins.

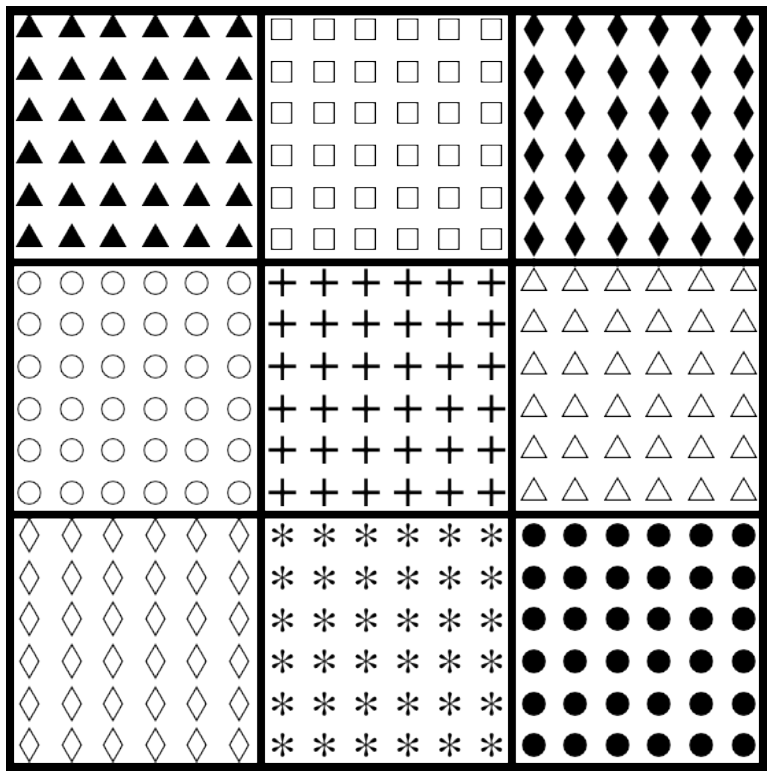


Figure 8 – Découpage en sous-domaines

Introduction

Historique

- Version 1.0 : en juin 1994, le forum MPI, avec la participation d'une quarantaine d'organisations, aboutit à la définition d'un ensemble de sous-programmes concernant la bibliothèque d'échanges de messages `MPI`
- Version 1.1 : juin 1995, avec seulement des changements mineurs
- Version 1.2 : en 1997, avec des changements mineurs pour une meilleure cohérence des dénominations de certains sous-programmes
- Version 1.3 : septembre 2008, avec des clarifications dans MPI 1.2, en fonction des clarifications elles-mêmes apportées par MPI-2.1
- Version 2.0 : apparue en juillet 1997, cette version apportait des compléments importants volontairement non intégrés dans MPI 1.0 (gestion dynamique de processus, copies mémoire à mémoire, entrées-sorties parallèles, etc.)
- Version 2.1 : juin 2008, avec seulement des clarifications dans MPI 2.0 mais aucun changement
- Version 2.2 : septembre 2009, avec seulement de petites additions

Introduction

MPI 3.0

- Version 3.0 : septembre 2012 changements et ajouts importants par rapport à la version 2.2 ;
  - communications collectives non bloquantes ;
  - révision de l'implémentation des copies mémoire à mémoire ;
  - Fortran (2003-2008) interface ;
  - suppression de l'interface C++ ;
  - interfaçage d'outils externes (pour le débogage et les mesures de performance) ;
  - etc.
- Version 3.1 : juin 2015
  - Correction de l'interface Fortran (2003-2008) ;
  - Nouvelles fonctions pour les écritures collectives non bloquantes ;

MPI 4.0

Version 4.0 : juin 2021

- Grand nombre
- Communication par morceaux
- MPI Session

Introduction

Bibliographie

- Site du MPI Forum http://www.mpi-forum.org
- Normes disponible en PDF sur http://www.mpi-forum.org/docs/
- William Gropp, Ewing Lusk et Anthony Skjellum : *Using MPI, third edition Portable Parallel Programming with the Message-Passing Interface*, MIT Press, 2014.
- William Gropp, Torsten Hoefler, Rajeev Thakur et Erwing Lusk : *Using Advanced MPI Modern Features of the Message-Passing Interface*, MIT Press, 2014.
- Victor Eijkhout : The Art of HPC http://theartofhpc.com

Implémentations MPI *open source*

Elles peuvent être installées sur un grand nombre d'architectures mais leurs performances sont en général en dessous de celles des implémentations constructeurs.

- MPICH : `http://www.mpich.org`
- Open MPI : `http://www.open-mpi.org`

Introduction

Outils

- Débogueurs
  - Totalview
    https://totalview.io
  - DDT
    https://www.linaroforge.com/linaroDdt/
- Outils de mesure de performances
  - FPMPI : *FPMPI*
    http://www.mcs.anl.gov/research/projects/fpmpi/WWW/
  - Scalasca : *Scalable Performance Analysis of Large-Scale Applications*
    http://www.scalasca.org
  - MUST : *MPI Runtime Correctness Analysis*
    https://itc.rwth-aachen.de/must/

Introduction

Bibliothèques scientifiques parallèles *open source*

- ScaLAPACK : résolution de problèmes d'algèbre linéaire par des méthodes directes.
  http://www.netlib.org/scalapack/
- PETSc : résolution de problèmes d'algèbre linéaire et non-linéaire par des méthodes itératives.
  https://petsc.org/release/
- PaStiX : résolution de grands systèmes linéaires creux.
  https://solverstack.gitlabpages.inria.fr/pastix/
- FFTW : transformées de Fourier rapides.
  http://www.fftw.org

Environnement

Environnement

Description

- Toute unité de programme appelant des routines MPI doit inclure un fichier d'en-têtes.
- En Fortran, il faut utiliser le *module* `mpi_f08` introduit dans MPI-3. Auparavant en MPI-2, il fallait utiliser le *module* `mpi`, et en MPI-1, il s'agissait du fichier `mpif.h`.
- Le sous-programme `MPI_Init()` permet d'initialiser l'environnement nécessaire :

```
MPI_INIT(code)

integer, optional, intent(out) :: code
```

- Réciproquement, le sous-programme `MPI_Finalize()` désactive cet environnement :

```
MPI_FINALIZE(code)

integer, optional, intent(out) :: code
```

Environnement

Différences entre le C/C++ et le Fortran

En C/C++ :

- il faut inclure le fichier `mpi.h` ;
- l'argument `code` est la valeur de retour de l'appel ;
- uniquement le préfixe MPI ainsi que la première lettre suivante sont en majuscules ;
- hormis `MPI_Init()`, les arguments des appels sont identiques au Fortran.

```
int MPI_Init(int *argc, char ***argv);
int MPI_Finalize(void);
```

Environnement

Communicateurs

- Toutes les opérations effectuées par MPI portent sur des communicateurs. Le communicateur par défaut est `MPI_COMM_WORLD` qui comprend tous les processus actifs.

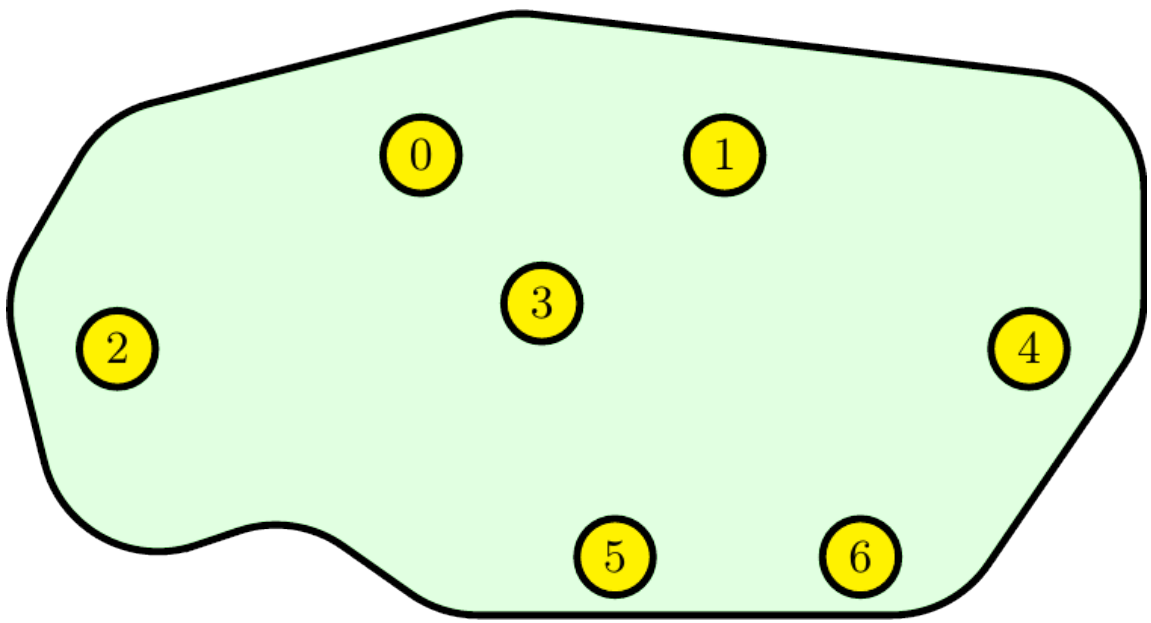


Figure 9 – Communicateur MPI_COMM_WORLD

Environnement

Arrêt d'un programme

Parfois un programme se trouve dans une situation où il doit s'arrêter sans attendre la fin normale. C'est typiquement le cas si un des processus ne peut pas allouer la mémoire nécessaire à son calcul. Dans ce cas il faut utiliser le sous-programme `MPI_Abort()` et non l'instruction Fortran stop (Ou *exit* in C).

```
MPI_ABORT(comm, erreur, code)

TYPE(MPI_Comm), intent(in)      :: comm
integer, intent(in)             :: erreur
integer, optional, intent(out) :: code
```

- `comm` : tous les processus appartenant à ce communicateur seront stoppés, il est donc conseillé d'utiliser `MPI_COMM_WORLD` ;
- `erreur` : numéro d'erreur retourné à l'environnement UNIX.

Code

Il n'est pas nécessaire de tester la valeur de `code` (valeur de retour en C) après des appels aux routines MPI. Par défaut, lorsque MPI rencontre un problème, le programme s'arrête comme lors d'un appel à `MPI_Abort()`.

Environnement

Rang et nombre de processus

- À tout instant, on peut connaître le nombre de processus gérés par un communicateur en appelant le sous-programme `MPI_Comm_size()` :

```
MPI_COMM_SIZE(comm,nb_procs,code)

TYPE(MPI_Comm)                  :: comm
integer, intent(out)            :: nb_procs
integer, optional, intent(out)  :: code
```

- De même, le sous-programme `MPI_Comm_rank()` permet d'obtenir le rang d'un processus (i.e. son numéro d'instance, qui est un nombre compris entre 0 et la valeur renvoyée par `MPI_COMM_SIZE()` – 1) :

```
MPI_COMM_RANK(comm,rang,code)

TYPE(MPI_Comm), intent(in)      :: comm
integer, intent(out)            :: rang
integer, optional, intent(out)  :: code
```

Environnement

Exemple

```
1  program qui_je_suis
2    use mpi_f08
3    implicit none
4    integer :: nb_procs,rang
5
6    call MPI_INIT()
7
8    call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs)
9    call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
10
11   print *,'Je suis le processus ',rang,' parmi ',nb_procs
12
13   call MPI_FINALIZE()
14 end program qui_je_suis
```

```
> mpiexec -n 7 qui_je_suis

Je suis le processus 3 parmi 7
Je suis le processus 0 parmi 7
Je suis le processus 4 parmi 7
Je suis le processus 1 parmi 7
Je suis le processus 5 parmi 7
Je suis le processus 2 parmi 7
Je suis le processus 6 parmi 7
```

Environnement

Compilation et exécution d'un code MPI

- Pour compiler un code MPI, on utilise un enrobeur (*wrapper*) de compilateur qui fait le lien avec la librairie MPI utilisée.
- Cet enrobeur diffère selon le langage de programmation, le compilateur et la librairie MPI utilisés. Par exemple : `mpif90`, `mpifort`, `mpicc`, ...

```
> mpif90 <options> -c source.f90
> mpif90 source.o -o mon_executable
```

- Pour exécuter un code MPI, on utilise un lanceur d'application MPI qui ordonne le lancement de l'exécution sur un nombre de processus choisi.
- Le lanceur défini par la norme MPI est `mpiexec`. Il existe également des lanceurs non standards, comme `mpirun`.

```
> mpiexec -n <nombre de processus> mon_executable
```

Travaux pratiques MPI – Exercice 1 : Environnement MPI

- Implémenter un programme MPI dans lequel chaque processus affiche un message indiquant si son rang est pair ou impair. Par exemple :

```
> mpiexec -n 4 ./pair_impair
Moi, processus 0, je suis de rang pair
Moi, processus 2, je suis de rang pair
Moi, processus 3, je suis de rang impair
Moi, processus 1, je suis de rang impair
```

- Pour tester la parité, la fonction intrinsèque Fortran correspondant à l'opération *modulo* est `mod` :

```
mod(a,b)
```

(en C, utilisez le symbole `%` : `a%b`)

- Pour compiler votre programme, utilisez la commande `make`
- Pour exécuter votre programme, utilisez la commande `make exe`

- Pour être reconnu par le Makefile, le programme doit se nommer `pair_impair.f90` (ou `pair_impair.c`)

Communications point à point

Communications point à point

Notions générales

Une communication dite point à point a lieu entre deux processus, l'un appelé processus émetteur et l'autre processus récepteur (ou destinataire).

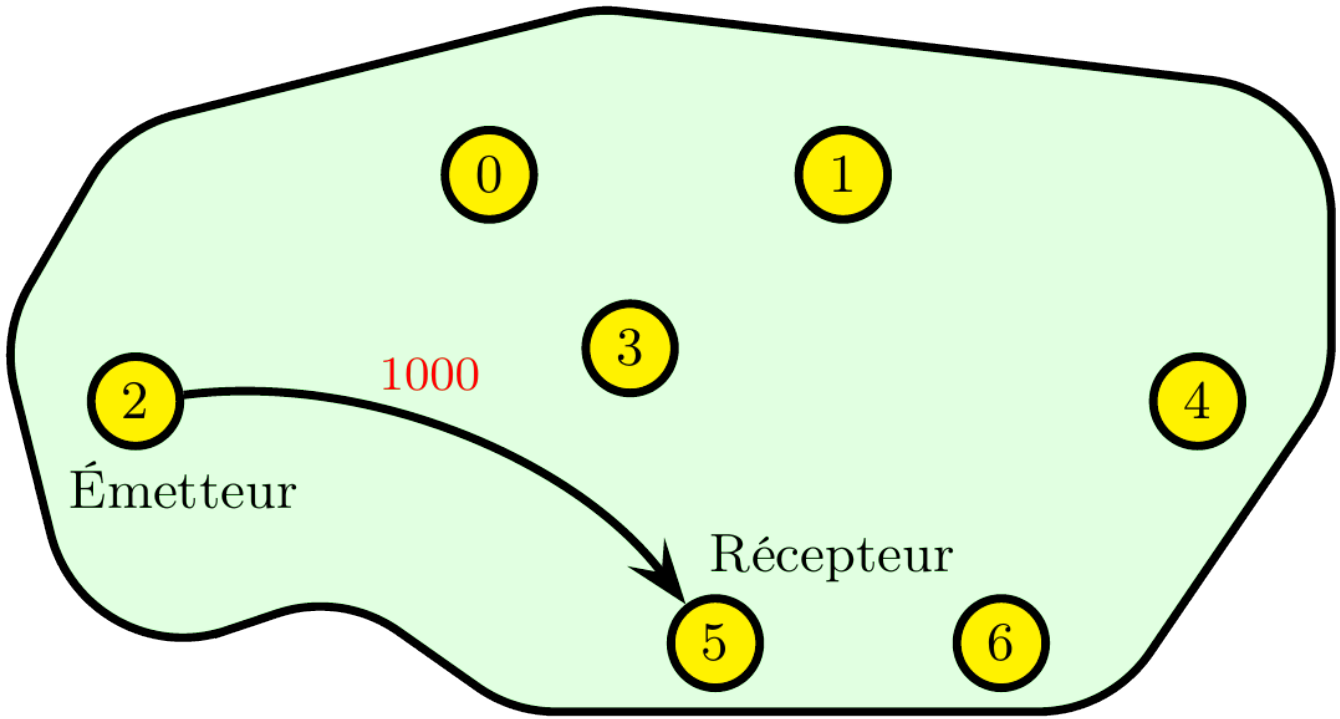


Figure 10 – Communication point à point

Communications point à point

Notions générales

- L'émetteur et le récepteur sont identifiés par leur rang dans le communicateur.
- L'entité transmise entre deux processus est appelée message.
- Un message est caractérisé par son enveloppe. Celle-ci est constituée :
  - du rang du processus émetteur ;
  - du rang du processus récepteur ;
  - de l'étiquette (*tag*) du message ;
  - du communicateur qui définit le groupe de processus et le contexte de communication.
- Les données échangées sont typées (entiers, réels, etc ou types dérivés personnels).
- Il existe dans chaque cas plusieurs modes de transfert, faisant appel à des protocoles différents.

Communications point à point

Opération d'envoi `MPI_Send`

```
MPI_SEND(message,longueur,type_message,rang_dest,etiquette,comm,code)

TYPE(*), dimension(..), intent(in) :: message
integer, intent(in)                :: longueur, rang_dest, etiquette
TYPE(MPI_Datatype), intent(in)     :: type_message
TYPE(MPI_Comm), intent(in)         :: comm
integer, optional, intent(out)     :: code
```

Envoi, à partir de l'adresse message, d'un message de taille longueur, de type type_message, étiqueté etiquette, au processus rang_dest dans le communicateur comm.

Remarque :
Cette opération est bloquante : l'exécution reste bloquée jusqu'à ce que le contenu de message puisse être réécrit sans risque d'écraser la valeur qui devait être envoyée.

Communications point à point

Opération de réception MPI_Recv

```
MPI_RECV(message,longueur,type_message,rang_source,etiquette,comm,statut,code)

TYPE(*), dimension(..), intent(in) :: message
integer, intent(in)                :: longueur,rang_source, etiquette
TYPE(MPI_Datatype),intent(in)      :: type_message
TYPE(MPI_Comm), intent(in)         :: comm
TYPE(MPI_Status)                   :: statut
integer, optional, intent(out)     :: code
```

Réception, à partir de l'adresse message, d'un message de taille longueur, de type type_message, étiqueté etiquette, du processus rang_source.

Remarques :

- statut stocke des informations sur la communication : rang_source, etiquette, code,...
- L'appel MPI_Recv ne pourra fonctionner avec une opération MPI_Send que si ces deux appels ont la même enveloppe (rang_source, rang_dest, etiquette, comm).
- Cette opération est bloquante : l'exécution reste bloquée jusqu'à ce que le contenu de message corresponde au message reçu.

Communications point à point

Exemple (voir Fig. 10)

```
1  program point_a_point
2    use mpi_f08
3    implicit none
4
5    TYPE(MPI_Status)     :: statut
6    integer, parameter   :: etiquette=100
7    integer              :: rang,valeur
8
9    call MPI_INIT()
10
11   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
12
13   if (rang == 2) then
14      valeur=1000
15      call MPI_SEND(valeur,1,MPI_INTEGER,5,etiquette,MPI_COMM_WORLD)
16   elseif (rang == 5) then
17      call MPI_RECV(valeur,1,MPI_INTEGER,2,etiquette,MPI_COMM_WORLD,statut)
18      print *,'Moi, processus 5, ai recu ',valeur,' du processus 2.'
19   end if
20
21   call MPI_FINALIZE()
22
23 end program point_a_point
```

```
> mpiexec -n 7 point_a_point

Moi, processus 5, ai recu 1000 du processus 2
```

Communications point à point

Types 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_BYTE	

Communications point à point

Types de données de base Fortran

- Le langage Fortran 95 introduit deux fonctions intrinsèques selected_int_kind() et selected_real_kind() qui permettent de définir la précision et/ou l'étendue d'un nombre entier, réel ou complexe
- MPI assure la portabilité de ces types de données avec MPI_TYPE_CREATE_F90_INTEGER(), MPI_TYPE_CREATE_F90_REAL() et MPI_TYPE_CREATE_F90_COMPLEX()

```
MPI_TYPE_CREATE_F90_INTEGER(r, newtype, code)

INTEGER, INTENT(IN)              :: r
TYPE(MPI_Datatype), INTENT(OUT) :: newtype
INTEGER, OPTIONAL, INTENT(OUT)   :: code

MPI_TYPE_CREATE_F90_REAL(p, r, newtype, code)

INTEGER, INTENT(IN)              :: p, r
TYPE(MPI_Datatype), INTENT(OUT) :: newtype
INTEGER, OPTIONAL, INTENT(OUT)   :: code
```

```
! Kind pour double precision
integer, parameter :: dp = selected_real_kind(15,307)
! Kind pour entier long
integer, parameter :: li = selected_int_kind(15)
integer(kind=li)   :: nbbloc
real(kind=dp)      :: largeur
call MPI_TYPE_CREATE_F90_INTEGER(15,typeli)
call MPI_TYPE_CREATE_F90_REAL(15,307,typedp)
```

Communications point à point

Autres possibilités

- À la réception d'un message, le rang de l'émetteur et l'étiquette peuvent être des « *jokers* », respectivement `MPI_ANY_SOURCE` et `MPI_ANY_TAG`.
- Une communication impliquant le processus « fictif » de rang `MPI_PROC_NULL` n'a aucun effet.
- `MPI_STATUS_IGNORE` est une constante prédéfinie qui peut être utilisée à la place de la variable `statut`.
- On peut communiquer des structures de données plus complexes en créant ses propres types dérivés.
- Il existe d'autres opérations qui effectuent simultanément un envoi et une réception : `MPI_Sendrecv()` et `MPI_Sendrecv_replace()`.

Communications point à point

Opération d'envoi et de réception simultanés MPI_Sendrecv

```
MPI_SENDRECV(message_emis,longueur_message_emis,type_message_emis,
             rang_dest,etiq_message_emis,
             message_recu, longueur_message_recu, type_message_recu,
             rang_source, etiq_message_recu, comm, statut, code)

TYPE(*), dimension(..), intent(in) :: message_emis
TYPE(*), dimension(..)             :: message_recu
integer, intent(in)                :: longueur_message_emis, longueur_message_recu
integer, intent(in)                :: rang_source, rang_dest, etiq_message_emis, etiq_message_recu
TYPE(MPI_Datatype), intent(in)     :: type_message_emis, type_message_recu
TYPE(MPI_Comm), intent(in)         :: comm
TYPE(MPI_Status)                   :: statut
integer, optional, intent(out)     :: code
```

- Envoi, à partir de l'adresse message_emis, d'un message de taille longueur_message_emis, de type type_message_emis, étiqueté etiq_message_emis, au processus rang_dest dans le communicateur comm ;
- Réception, à partir de l'adresse message_recu, d'un message de taille longueur_message_recu, de type type_message_recu, étiqueté etiq_message_recu, du processus rang_source dans le communicateur comm.

Remarque :

- La zone de réception message_recu doit différer de la zone d'envoi message_emis.

Communications point à point

Opération d'envoi et de réception simultanés `MPI_Sendrecv`

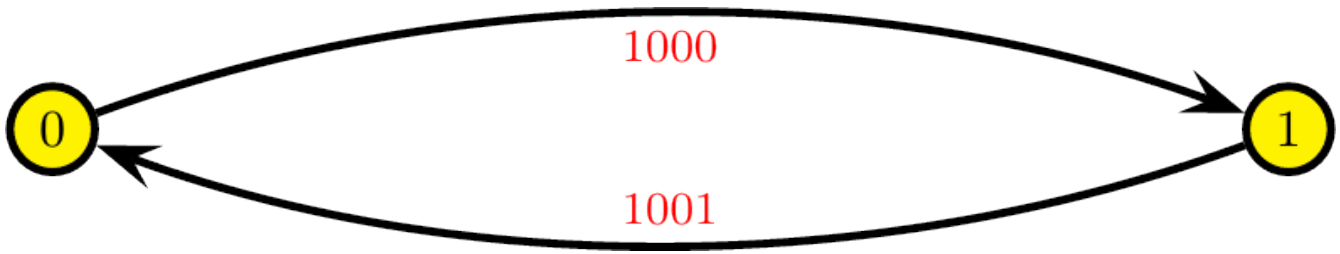


Figure 11 – Communication `sendrecv` entre les processus 0 et 1

Communications point à point

Exemple (voir Fig. 11)

```
1  program sendrecv
2    use mpi_f08
3    implicit none
4    integer                              :: rang,valeur,num_proc
5    integer,parameter                    :: etiquette=110
6
7    call MPI_INIT()
8    call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
9
10   ! On definit le rang du processus avec lequel on va communiquer ( on suppose avoir exactement 2 processus )
11   num_proc=mod(rang+1,2)
12
13   call MPI_SENDRECV(rang+1000,1,MPI_INTEGER,num_proc,etiquette,valeur,1,MPI_INTEGER, &
14                     num_proc,etiquette,MPI_COMM_WORLD,MPI_STATUS_IGNORE)
15
16   print *,'Moi, processus',rang,', ai recu',valeur,'du processus',num_proc
17
18   call MPI_FINALIZE()
19 end program sendrecv
```

```
> mpiexec -n 2 sendrecv

Moi, processus 1, ai recu 1000 du processus 0
Moi, processus 0, ai recu 1001 du processus 1
```

Communications point à point

Attention !
Dans le cas d'une implémentation synchrone de `MPI_Send()`, l'exemple précédent serait en situation de verrouillage si l'appel à `MPI_Sendrecv()` était remplacé par un `MPI_Send()` suivi d'un `MPI_Recv()`. En effet, chacun des deux processus attendrait un ordre de réception qui ne viendrait jamais, puisque les deux envois resteraient en suspens.

```
call MPI_SEND(rang+1000,1,MPI_INTEGER,num_proc,etiquette,MPI_COMM_WORLD)
call MPI_RECV(valeur,1,MPI_INTEGER,num_proc,etiquette,MPI_COMM_WORLD,statut)
```

Communications point à point

Opération d'envoi et de réception simultanés `MPI_Sendrecv_replace`

```
MPI_SENDRECV_REPLACE(message,longueur,type_message,
                     rang_dest, etiq_message_emis,
                     rang_source, etiq_message_recu, comm, statut, code)

TYPE(*),dimension(..)           :: message
integer, intent(in)             :: longueur, rang_source, rang_dest, etiq_message_emis, etiq_message_recu
TYPE(MPI_Datatype), intent(in)  :: type_message
TYPE(MPI_Comm), intent(in)      :: comm
TYPE(MPI_Status)                :: statut
integer, optional, intent(out)  :: code
```

- Envoi, à partir de l'adresse message, d'un message de taille longueur, de type type_message, étiqueté etiq_message_emis, au processus rang_dest dans le communicateur comm ;
- Réception d'un message à la même adresse, d'une taille et d'un type identique, étiqueté etiq_message_recu, du processus rang_source dans le communicateur comm.

Remarque :

- Contrairement à l'usage imposée par `MPI_Sendrecv()`, la zone de réception coincide ici avec la zone d'envoi message.

Communications point à point

Exemple

```
1  program joker
2    use mpi_f08
3    implicit none
4    integer, parameter       :: m=4,etiquette=11
5    integer, dimension(m,m) :: A
6    integer                  :: nb_procs,rang,i
7    TYPE(MPI_Status)         :: statut
8
9    call MPI_INIT ()
10   call MPI_COMM_SIZE ( MPI_COMM_WORLD ,nb_procs)
11   call MPI_COMM_RANK ( MPI_COMM_WORLD ,rang)
12   A(:,:) = 0
13
14   if (rang == 0) then
15     ! Initialisation de la matrice A sur le processus 0
16     A(:,:) = reshape((/ (i,i=1,m*m) /), (/ m,m /))
17     ! Envoi de 3 elements de la matrice A au processus 1
18     call MPI_SEND (A(1,1),3, MPI_INTEGER ,1,etiquette, MPI_COMM_WORLD)
19   else
20     ! On recoit le message
21     call MPI_RECV (A(1,2),3, MPI_INTEGER ,MPI_ANY_SOURCE,MPI_ANY_TAG, &
22                    MPI_COMM_WORLD ,statut)
23     print *,'Moi processus ',rang ,", ai recu 3 elements du processus ", &
24             statut%MPI_SOURCE , "avec comme etiquette", statut%MPI_TAG , &
25             " les elements sont ", A(1:3,2)
26   end if
27   call MPI_FINALIZE()
28 end program joker
```

Communications point à point

```
> mpiexec -n 2 joker
 Moi processus            1, ai recu 3 elements du processus            0
 avec comme etiquette          11  les elements sont            1           2           3
```

Travaux pratiques MPI – Exercice 2 : Ping-pong

- Communications point à point : *Ping-Pong* entre deux processus

- L'exercice 2 est décomposé en 3 étapes :

  1. *Ping* : compléter le script `ping_pong_1.f90` de manière à ce que le processus de rang 0 envoie un message contenant une série aléatoire de 1000 réels au rang 1.

  2. *Ping-Pong* : compléter le script `ping_pong_2.f90` de manière à ce que le processus de rang 1 renvoie le message vers le processus de rang 0, et mesurer le temps pris par la communication à l'aide de la fonction `MPI_Wtime()`.

  3. *Match de Ping-Pong* : compléter le script `ping_pong_3.f90` de manière à enchainer 9 *Ping-Pong*, en faisant varier la taille du message, et mesurer les temps pris par chaque échange. Les débits correspondants seront affichés.

Travaux pratiques MPI – Exercice 2 : Ping-pong

Remarques :

- Pour compiler la première étape : `make ping_pong_1`
- Pour exécuter la première étape : `make exe1`
- Pour compiler la seconde étape : `make ping_pong_2`
- Pour exécuter la seconde étape : `make exe2`
- Pour compiler la dernière étape : `make ping_pong_3`
- Pour exécuter la dernière étape : `make exe3`
- La génération de nombres réels pseudo-aléatoires uniformément répartis dans l'intervalle [0,1[se fait en Fortran par un appel au sous-programme `random_number` :

```
call random_number(variable)
```

  `variable` pouvant être un scalaire ou un tableau

- Les mesures de temps peuvent s'effectuer de la façon suivante :

```
temps_debut=MPI_WTIME()
.................................................
temps_fin=MPI_WTIME()
print ('("... en",f8.6," secondes.")'),temps_fin-temps_debut
```

Communications collectives

Communications collectives

Notions générales

- Les communications collectives permettent de faire en une seule opération une série de communications point à point.
- Une communication collective concerne toujours tous les processus du communicateur indiqué.
- Pour chacun des processus, l'appel se termine lorsque la participation de celui-ci à l'opération collective est achevée, au sens des communications point-à-point (donc quand la zone mémoire concernée peut être modifiée).
- La gestion des étiquettes dans ces communications est transparente et à la charge du système. Elles ne sont donc jamais définies explicitement lors de l'appel à ces sous-programmes. Cela a entre autres pour avantage que les communications collectives n'interfèrent jamais avec les communications point à point.

Communications collectives

Types de communications collectives

Il y a trois types de sous-programmes :

1. celui qui assure les synchronisations globales : `MPI_Barrier()`.
2. ceux qui ne font que transférer des données :
   - diffusion globale de données : `MPI_Bcast()` ;
   - diffusion sélective de données : `MPI_Scatter()` ;
   - collecte de données réparties : `MPI_Gather()` ;
   - collecte par tous les processus de données réparties : `MPI_Allgather()` ;
   - collecte et diffusion sélective, par tous les processus, de données réparties : `MPI_Alltoall()`.
3. ceux qui, en plus de la gestion des communications, effectuent des opérations sur les données transférées :
   - opérations de réduction (somme, produit, maximum, minimum, etc.), qu'elles soient d'un type prédéfini ou d'un type personnel : `MPI_Reduce()` ;
   - opérations de réduction avec diffusion du résultat (équivalent à un `MPI_Reduce()` suivi d'un `MPI_Bcast()`) : `MPI_Allreduce()`.

Communications collectives

Synchronisation globale : `MPI_Barrier()`

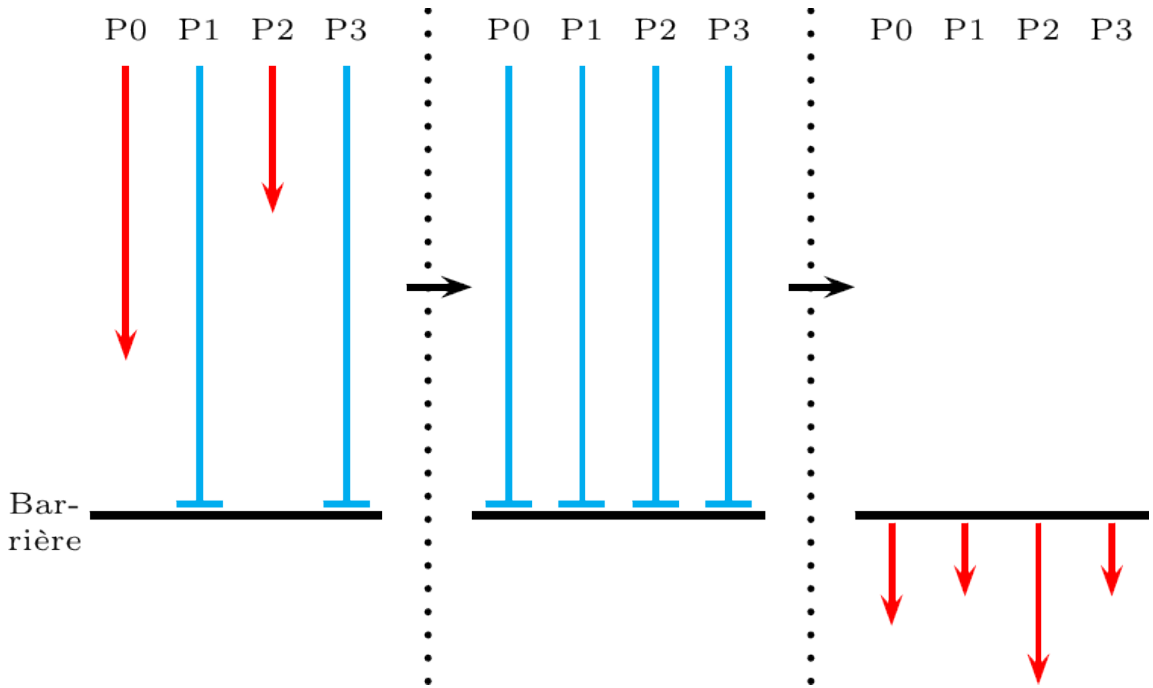


Figure 12 – Synchronisation globale : `MPI_Barrier()`

```
MPI_BARRIER(comm,code)

TYPE(MPI_Comm),intent(in)       :: comm
integer, optional, intent(out) :: code
```

Communications collectives

Diffusion générale : `MPI_Bcast()`

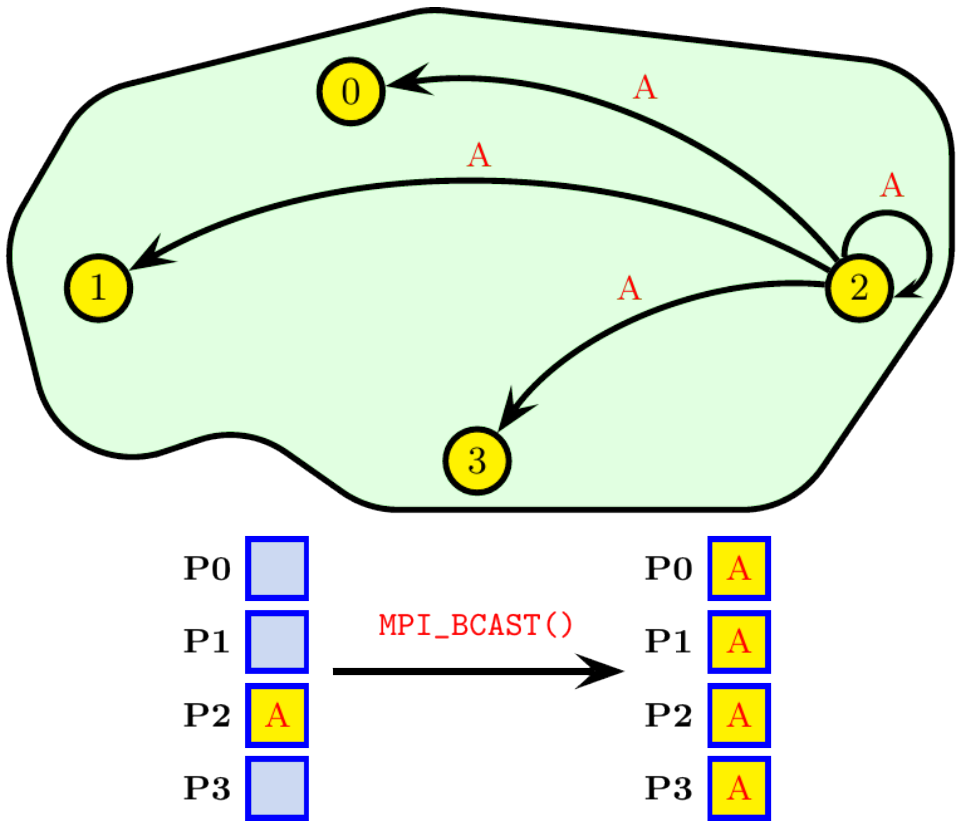


Figure 13 – Diffusion générale : `MPI_Bcast()`

Communications collectives

Diffusion générale : `MPI_Bcast()`

```
MPI_BCAST(message, longueur, type_message, rang_source, comm, code)

TYPE(*), dimension(..)          :: message
integer, intent(in)             :: longueur, rang_source
TYPE(MPI_Datatype), intent(in)  :: type_message
TYPE(MPI_Comm), intent(in)      :: comm
integer, optional, intent(out)  :: code
```

1. Envoi, à partir de l'adresse message, d'un message constitué de longueur élément de type type_message, par le processus rang_source, à tous les autres processus du communicateur comm.
2. Réception de ce message à l'adresse message pour les processus autre que rang_source.

Communications collectives

Exemple de `MPI_Bcast()`

```
1  program bcast
2    use mpi_f08
3    implicit none
4
5    integer :: rang,valeur
6
7    call MPI_INIT()
8    call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
9
10   if (rang == 2) valeur=rang+1000
11
12   call MPI_BCAST(valeur,1,MPI_INTEGER,2,MPI_COMM_WORLD)
13
14   print *,'Moi, processus ',rang,', ai recu ',valeur,' du processus 2'
15
16   call MPI_FINALIZE()
17
18 end program bcast
```

```
> mpiexec -n 4 bcast

Moi, processus 2, ai recu 1002 du processus 2
Moi, processus 0, ai recu 1002 du processus 2
Moi, processus 1, ai recu 1002 du processus 2
Moi, processus 3, ai recu 1002 du processus 2
```

Diffusion sélective : `MPI_Scatter()`

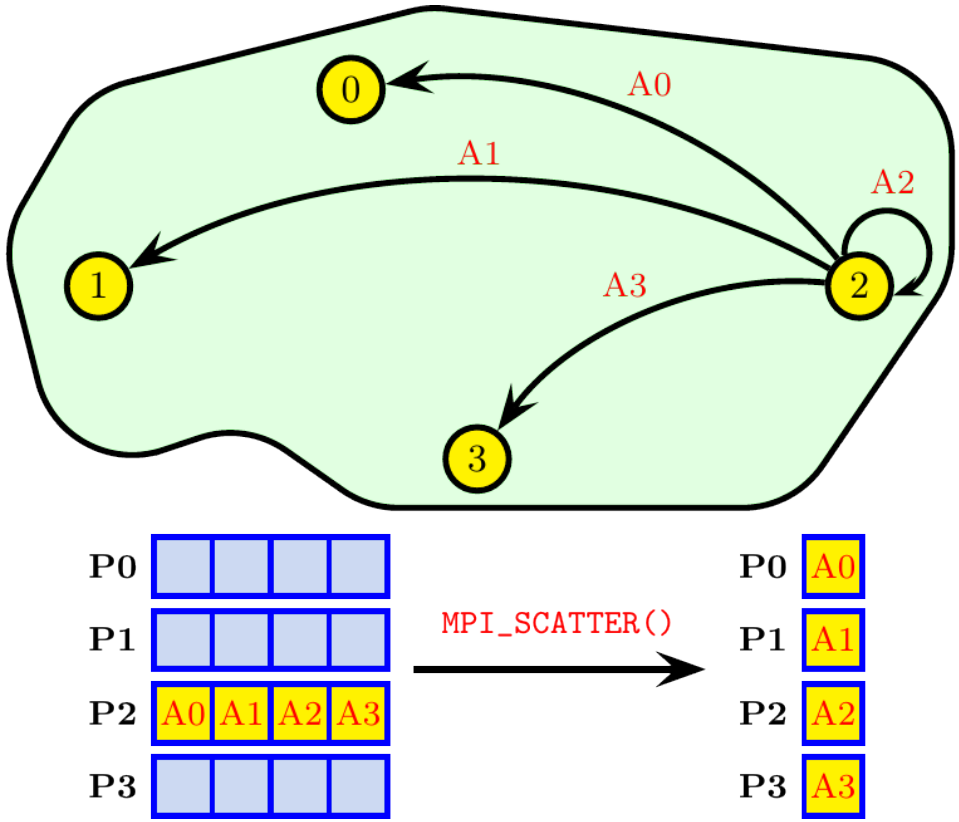


Figure 14 – Diffusion sélective : `MPI_Scatter()`

Communications collectives

Diffusion sélective : MPI_Scatter()

```
MPI_SCATTER(message_a_repartir,longueur_message_emis,type_message_emis,
            message_recu,longueur_message_recu,type_message_recu,rang_source,comm,code)

TYPE(*), dimension(..)          :: message_a_repartir, message_recu
integer, intent(in)             :: longueur_message_emis, longueur_message_recu, rang_source
TYPE(MPI_Datatype), intent(in)  :: type_message_emis, type_message_recu
TYPE(MPI_Comm), intent(in)      :: comm
integer, optional, intent(out)  :: code
```

1. Distribution, par le processus rang_source, à partir de l'adresse message_a_repartir, d'un message de taille longueur_message_emis, de type type_message_emis, à tous les processus du communicateur comm ;
2. réception du message à l'adresse message_recu, de longueur longueur_message_recu et de type type_message_recu par tous les processus du communicateur comm.

Remarques :

- Les couples (longueur_message_emis, type_message_emis) et (longueur_message_recu, type_message_recu) doivent être tels que les quantités de données envoyées et reçues soient égales.
- Les données sont distribuées en tranches égales, une tranche étant constituée de longueur_message_emis éléments du type type_message_emis.
- La ième tranche est envoyée au ième processus.

Communications collectives

Exemple de `MPI_Scatter()`

```
 1  program scatter
 2    use mpi_f08
 3    implicit none
 4
 5    integer, parameter                :: nb_valeurs=8
 6    integer                           :: nb_procs,rang,longueur_tranche,i
 7    real, allocatable, dimension(:) :: valeurs,donnees
 8
 9    call MPI_INIT()
10    call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs)
11    call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
12    longueur_tranche=nb_valeurs/nb_procs
13    allocate(donnees(longueur_tranche))
14
15    if (rang == 2) then
16       allocate(valeurs(nb_valeurs))
17       valeurs(:)=(/(1000.+i,i=1,nb_valeurs)/)
18       print *,'Moi, processus ',rang,' envoie mon tableau valeurs : ',&
19               valeurs(1:nb_valeurs)
20    end if
21
22    call MPI_SCATTER(valeurs,longueur_tranche,MPI_REAL,donnees,longueur_tranche, &
23                     MPI_REAL,2,MPI_COMM_WORLD)
24    print *,'Moi, processus ',rang,', ai recu ', donnees(1:longueur_tranche), &
25            ' du processus 2'
26    call MPI_FINALIZE()
27
28  end program scatter
```

```
> mpiexec -n 4 scatter
Moi, processus 2 envoie mon tableau valeurs :
1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.

Moi, processus 0, ai recu 1001. 1002. du processus 2
Moi, processus 1, ai recu 1003. 1004. du processus 2
Moi, processus 3, ai recu 1007. 1008. du processus 2
Moi, processus 2, ai recu 1005. 1006. du processus 2
```

Communications collectives

Collecte : `MPI_Gather()`

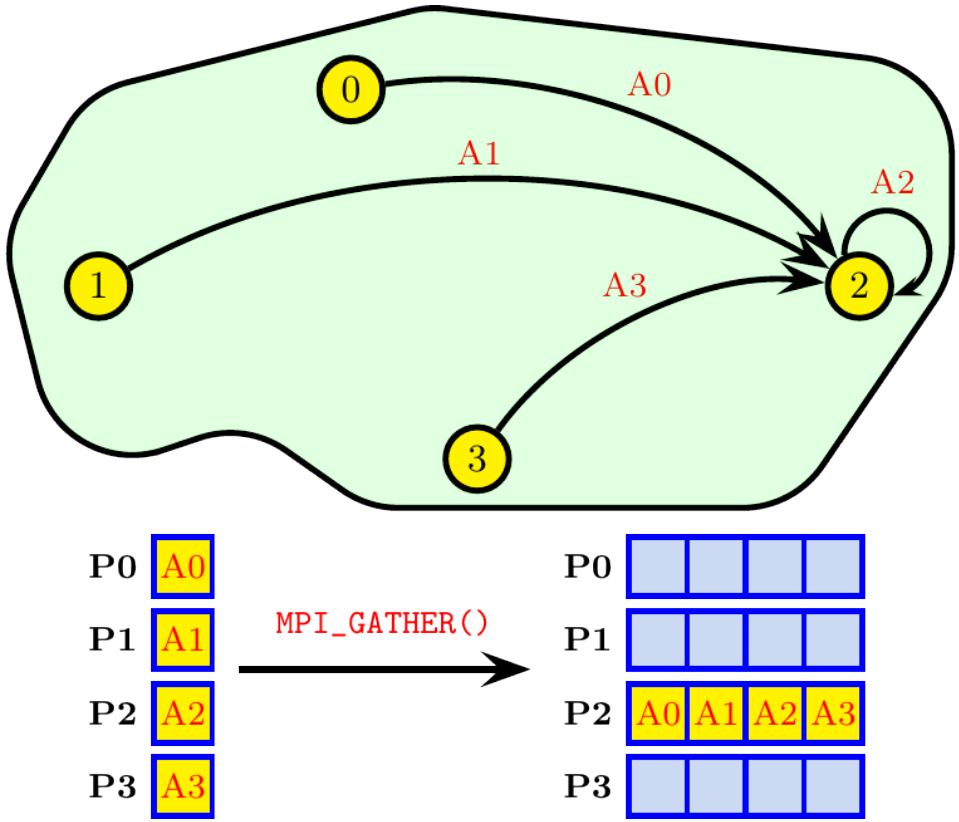


Figure 15 – Collecte : `MPI_Gather()`

Communications collectives

Collecte : `MPI_Gather()`

```
MPI_GATHER(message_emis,longueur_message_emis,type_message_emis,
           message_recu,longueur_message_recu,type_message_recu,rang_dest,comm,code)

TYPE(*), dimension(..), intent(in) :: message_emis
TYPE(*), dimension(..)             :: message_recu
integer, intent(in)                :: longueur_message_emis, longueur_message_recu, rang_dest
TYPE(MPI_Datatype), intent(in)     :: type_message_emis, type_message_recu
TYPE(MPI_Comm), intent(in)         :: comm
integer, optional, intent(out)     :: code
```

1. Envoi de chacun des processus du communicateur comm, d'un message message_emis, de taille longueur_message_emis et de type type_message_emis.
2. Collecte de chacun de ces messages, par le processus rang_dest, à partir l'adresse message_recu, sur une longueur longueur_message_recu et avec le type type_message_recu.

Remarques :

- Les couples (longueur_message_emis, type_message_emis) et (longueur_message_recu, type_message_recu) doivent être tels que les quantités de données envoyées et reçues soient égales.
- Les données sont collectées dans l'ordre des rangs des processus.

Communications collectives

Collecte : MPI_Gather()

```
1  program gather
2    use mpi_f08
3    implicit none
4    integer, parameter                :: nb_valeurs=8
5    integer                           :: nb_procs,rang,longueur_tranche,i
6    real, dimension(nb_valeurs)       :: donnees
7    real, allocatable, dimension(:) :: valeurs
8
9    call MPI_INIT()
10   call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs)
11   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
12
13   longueur_tranche=nb_valeurs/nb_procs
14
15   allocate(valeurs(longueur_tranche))
16
17   valeurs(:)=(/(1000.+rang*longueur_tranche+i,i=1,longueur_tranche)/)
18   print *,'Moi, processus ',rang,'envoie mon tableau valeurs : ',&
19               valeurs(1:longueur_tranche)
20
21   call MPI_GATHER(valeurs,longueur_tranche,MPI_REAL,donnees,longueur_tranche, &
22                   MPI_REAL,2,MPI_COMM_WORLD)
23
24   if (rang == 2) print *,'Moi, processus 2', ' ai recu ',donnees(1:nb_valeurs)
25
26   call MPI_FINALIZE()
27
28 end program gather
```

```
> mpiexec -n 4 gather
Moi, processus 1 envoie mon tableau valeurs :1003. 1004.
Moi, processus 0 envoie mon tableau valeurs :1001. 1002.
Moi, processus 2 envoie mon tableau valeurs :1005. 1006.
Moi, processus 3 envoie mon tableau valeurs :1007. 1008.

Moi, processus 2, ai recu 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
```

Communications collectives

Collecte générale : `MPI_Allgather()`

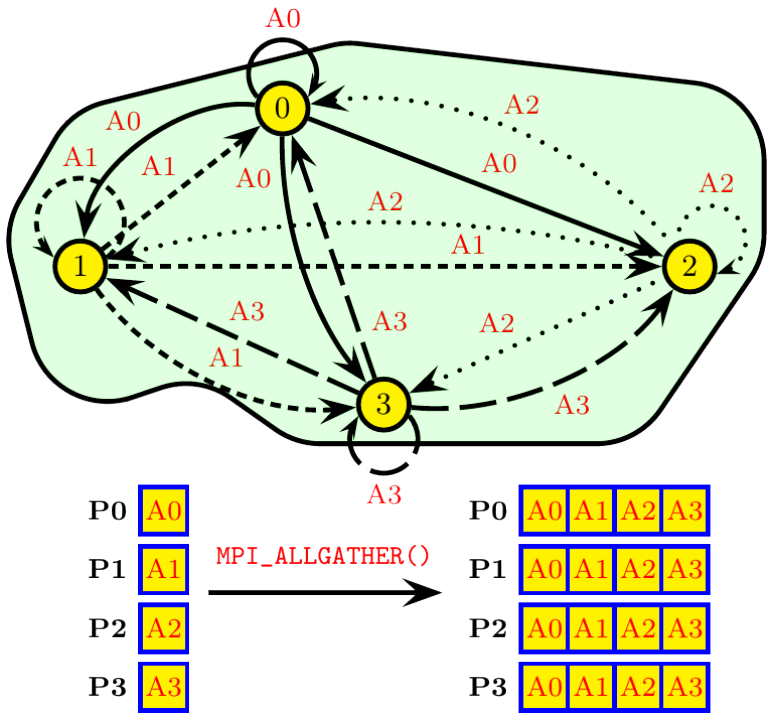


Figure 16 – Collecte générale : `MPI_Allgather()`

Communications collectives

Collecte générale : `MPI_Allgather()`

Correspond à un `MPI_Gather()` suivi d'un `MPI_Bcast()` :

```
MPI_ALLGATHER(message_emis,longueur_message_emis,type_message_emis,
              message_recu,longueur_message_recu,type_message_recu,comm,code)

TYPE(*), dimension(..), intent(in) :: message_emis
TYPE(*), dimension(..)             :: message_recu
integer, intent(in)                :: longueur_message_emis, longueur_message_recu
TYPE(MPI_Datatype), intent(in)     :: type_message_emis, type_message_recu
TYPE(MPI_Comm), intent(in)         :: comm
integer, optional, intent(out)     :: code
```

1. Envoi de chacun des processus du communicateur comm, d'un message message_emis, de taille longueur_message_emis et de type type_message_emis.
2. Collecte de chacun de ces messages, par tous les processus, à partir l'adresse message_recu, sur une longueur longueur_message_recu et avec le type type_message_recu.

Remarques :

- Les couples (longueur_message_emis, type_message_emis) et (longueur_message_recu, type_message_recu) doivent être tels que les quantités de données envoyées et reçues soient égales.
- Les données sont collectées dans l'ordre des rangs des processus.

Communications collectives

Exemple de `MPI_Allgather()`

```
 1  program allgather
 2    use mpi_f08
 3    implicit none
 4
 5    integer, parameter                :: nb_valeurs=8
 6    integer                           :: nb_procs,rang,longueur_tranche,i
 7    real, dimension(nb_valeurs)       :: donnees
 8    real, allocatable, dimension(:)   :: valeurs
 9
10    call MPI_INIT()
11
12    call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs)
13    call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
14
15    longueur_tranche=nb_valeurs/nb_procs
16    allocate(valeurs(longueur_tranche))
17
18    valeurs(:)=(/(1000.+rang*longueur_tranche+i,i=1,longueur_tranche)/)
19
20    call MPI_ALLGATHER(valeurs,longueur_tranche,MPI_REAL,donnees,longueur_tranche, &
21                       MPI_REAL,MPI_COMM_WORLD)
22
23    print *,'Moi, processus ',rang,', ai recu ',donnees(1:nb_valeurs)
24
25    call MPI_FINALIZE()
26
27  end program allgather
```

```
> mpiexec -n 4 allgather

Moi, processus 1, ai recu 1001. 1002.  1003. 1004.  1005. 1006.  1007. 1008.
Moi, processus 3, ai recu 1001. 1002.  1003. 1004.  1005. 1006.  1007. 1008.
Moi, processus 2, ai recu 1001. 1002.  1003. 1004.  1005. 1006.  1007. 1008.
Moi, processus 0, ai recu 1001. 1002.  1003. 1004.  1005. 1006.  1007. 1008.
```

Collecte "variable" : `MPI_Gatherv()`

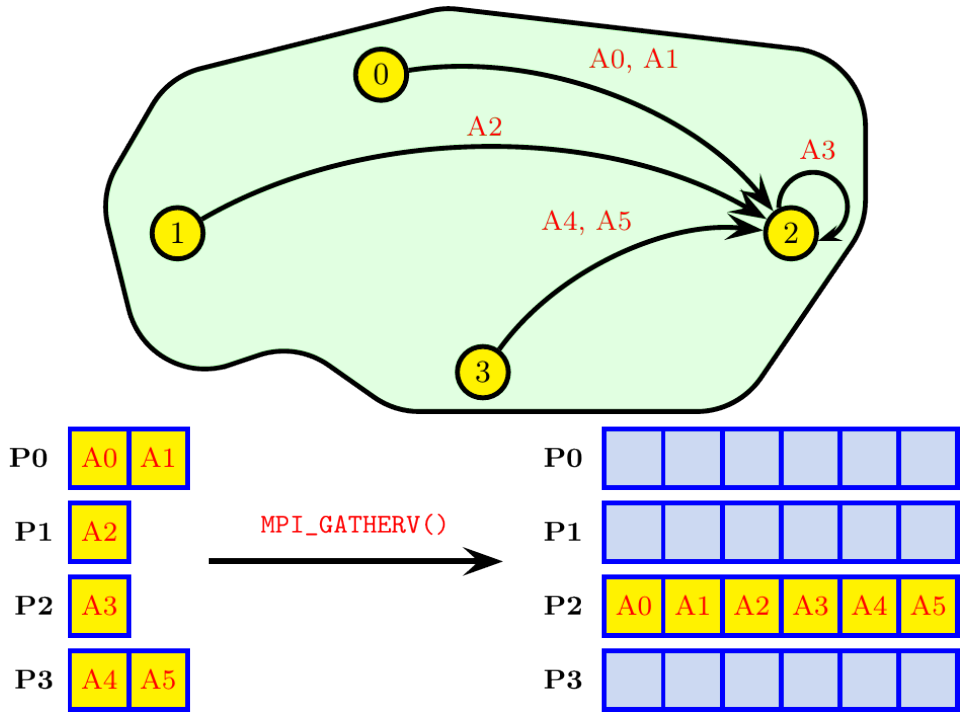


Figure 17 – Collecte : `MPI_Gatherv()`

Communications collectives

Collecte "variable" : `MPI_Gatherv()`

Correspond à un `MPI_Gather()` pour lequel la taille des messages varie :

```
MPI_GATHERV(message_emis,longueur_message_emis,type_message_emis,
            message_recu,nb_elts_recus,deplts,type_message_recu,
            rang_dest,comm,code)

TYPE(*), dimension(..), intent(in) :: message_emis
TYPE(*), dimension(..)             :: message_recu
integer, intent(in)                :: longueur_message_emis, rang_dest
TYPE(MPI_Datatype), intent(in)     :: type_message_emis, type_message_recu
integer, dimension(:), intent(in)  :: nb_elts_recus, deplts
TYPE(MPI_Comm), intent(in)         :: comm
integer, optional, intent(out)     :: code
```

Le ième processus du communicateur comm envoie au processus rang_dest, un message depuis l'adresse message_emis, de taille longueur_message_emis, de type type_message_emis, avec réception du message à l'adresse message_recu, de type type_message_recu, de taille nb_elts_recus(i) avec un déplacement de deplts(i).

Remarques :

- Les couples (longueur_message_emis, type_message_emis) du ième processus et (nb_elts_recus(i), type_message_recu) du processus rang_dest doivent être tels que les quantités de données envoyées et reçues soient égales.

Communications collectives

Exemple de MPI_Gatherv()

```
 1  program gatherv
 2    use mpi_f08
 3    implicit none
 4    integer, parameter                  :: nb_valeurs=10
 5    integer                             :: reste, nb_procs, rang, longueur_tranche, i
 6    real, dimension(nb_valeurs)         :: donnees
 7    real, allocatable, dimension(:)     :: valeurs
 8    integer, allocatable, dimension(:) :: nb_elements_recus,deplacements
 9
10    call MPI_INIT()
11    call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs)
12    call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
13
14    longueur_tranche=nb_valeurs/nb_procs
15    reste = mod(nb_valeurs,nb_procs)
16    if (rang < reste) longueur_tranche = longueur_tranche+1
17
18    ALLOCATE(valeurs(longueur_tranche))
19    valeurs(:) = (/(1000.+(rang*(nb_valeurs/nb_procs))+min(rang,reste)+i, &
20                    i=1,longueur_tranche)/)
21
22    PRINT *, 'Moi, processus ', rang,'envoie mon tableau valeurs : ',&
23               valeurs(1:longueur_tranche)
24
25    IF (rang == 2) THEN
26      ALLOCATE(nb_elements_recus(nb_procs),deplacements(nb_procs))
27      nb_elements_recus(1) = nb_valeurs/nb_procs
28      if (reste > 0) nb_elements_recus(1) = nb_elements_recus(1)+1
29      deplacements(1) = 0
30      DO i=2,nb_procs
31        deplacements(i) = deplacements(i-1)+nb_elements_recus(i-1)
32        nb_elements_recus(i) = nb_valeurs/nb_procs
33        if (i-1 < reste) nb_elements_recus(i) = nb_elements_recus(i)+1
34      END DO
35    END IF
```

Communications collectives

Exemple de `MPI_Gatherv()` (suite)

```
  CALL MPI_GATHERV (valeurs,longueur_tranche, MPI_REAL ,donnees,nb_elements_recus,&
                    deplacements, MPI_REAL ,2, MPI_COMM_WORLD)
  IF (rang == 2) PRINT *, 'Moi, processus 2 ai recu', donnees(1:nb_valeurs)
  CALL MPI_FINALIZE()
end program gatherv
```

```
> mpiexec -n 4 gatherv

 Moi, processus  0 envoie mon tableau valeurs :  1001. 1002. 1003.
 Moi, processus  2 envoie mon tableau valeurs :  1007. 1008.
 Moi, processus  3 envoie mon tableau valeurs :  1009. 1010.
 Moi, processus  1 envoie mon tableau valeurs :  1004. 1005. 1006.

 Moi, processus 2 ai recu  1001. 1002. 1003.  1004. 1005. 1006.  1007. 1008.  1009. 1010.
```

Communications collectives

Collectes et diffusions sélectives : `MPI_Alltoall()`

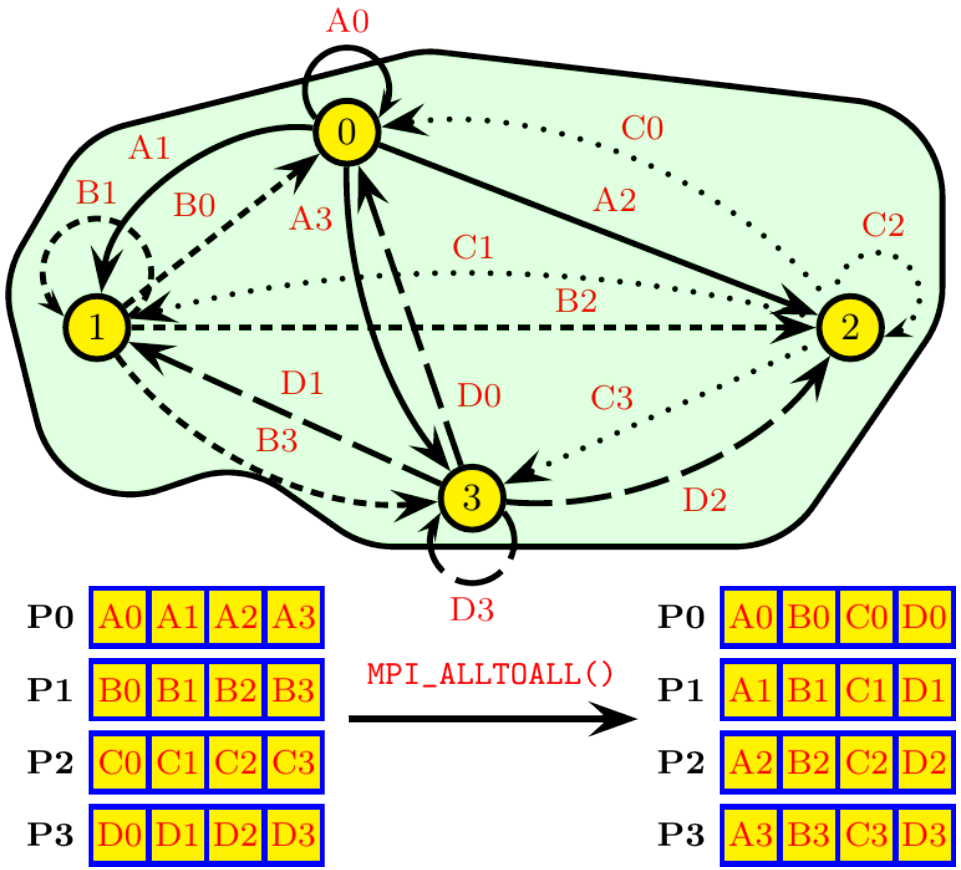


Figure 18 – Collecte et diffusion sélectives : `MPI_Alltoall()`

Communications collectives

Collectes et diffusions sélectives : `MPI_Alltoall()`

```
MPI_ALLTOALL(message_emis,longueur_message_emis,type_message_emis,
             message_recu,longueur_message_recu,type_message_recu,comm,code)

TYPE(*), dimension(..), intent(in) :: message_emis
TYPE(*), dimension(..)             :: message_recu
integer, intent(in)                :: longueur_message_emis, longueur_message_recu
TYPE(MPI_Datatype), intent(in)     :: type_message_emis, type_message_recu
TYPE(MPI_Comm), intent(in)         :: comm
integer, optional, intent(out)     :: code
```

Ici, le ième processus envoie la jème tranche au jème processus qui le place à l'emplacement de la ième tranche.

Remarque :

- Les couples (longueur_message_emis, type_message_emis) et (longueur_message_recu, type_message_recu) doivent être tels que les quantités de données envoyées et reçues soient égales.

Communications collectives

Exemple de `MPI_Alltoall()`

```
1  program alltoall
2    use mpi_f08
3    implicit none
4
5    integer, parameter               :: nb_valeurs=8
6    integer                          :: nb_procs,rang,longueur_tranche,i
7    real, dimension(nb_valeurs)      :: valeurs,donnees
8
9    call MPI_INIT()
10   call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs)
11   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
12
13   valeurs(:)=(/(1000.+rang*nb_valeurs+i,i=1,nb_valeurs)/)
14   longueur_tranche=nb_valeurs/nb_procs
15
16   print *,'Moi, processus ',rang," envoie mon tableau valeurs : ", &
17           valeurs(1:nb_valeurs)
18
19   call MPI_ALLTOALL(valeurs,longueur_tranche,MPI_REAL,donnees,longueur_tranche, &
20                     MPI_REAL,MPI_COMM_WORLD)
21
22   print *,'Moi, processus ',rang,', ai recu ',donnees(1:nb_valeurs)
23
24   call MPI_FINALIZE()
25 end program alltoall
```

Communications collectives

Exemple de `MPI_Alltoall()` (suite)

```
> mpiexec -n 4 alltoall
Moi, processus 1 envoie mon tableau valeurs :
1009. 1010. 1011. 1012. 1013. 1014. 1015. 1016.
Moi, processus 0 envoie mon tableau valeurs :
1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
Moi, processus 2 envoie mon tableau valeurs :
1017. 1018. 1019. 1020. 1021. 1022. 1023. 1024.
Moi, processus 3 envoie mon tableau valeurs :
1025. 1026. 1027. 1028. 1029. 1030. 1031. 1032.

Moi, processus 0, ai recu 1001. 1002.  1009. 1010.  1017. 1018.  1025. 1026.
Moi, processus 2, ai recu 1005. 1006.  1013. 1014.  1021. 1022.  1029. 1030.
Moi, processus 1, ai recu 1003. 1004.  1011. 1012.  1019. 1020.  1027. 1028.
Moi, processus 3, ai recu 1007. 1008.  1015. 1016.  1023. 1024.  1031. 1032.
```

Communications collectives

Réductions réparties

- Une réduction est une opération appliquée à un ensemble d'éléments pour en obtenir une seule valeur. Des exemples typiques sont la somme des éléments d'un vecteur `SUM(A(:))` ou la recherche de l'élément de valeur maximum dans un vecteur `MAX(V(:))`.
- MPI propose des sous-programmes de haut-niveau pour opérer des réductions sur des données réparties sur un ensemble de processus. Le résultat est obtenu sur un seul processus (`MPI_Reduce()`) ou bien sur tous (`MPI_Allreduce()`, qui est en fait équivalent à un `MPI_Reduce()` suivi d'un `MPI_Bcast()`).
- Si plusieurs éléments sont concernés par processus, la fonction de réduction est appliquée à chacun d'entre eux (par exemple à tous les éléments d'un vecteur).

Réductions réparties : `MPI_Reduce()`

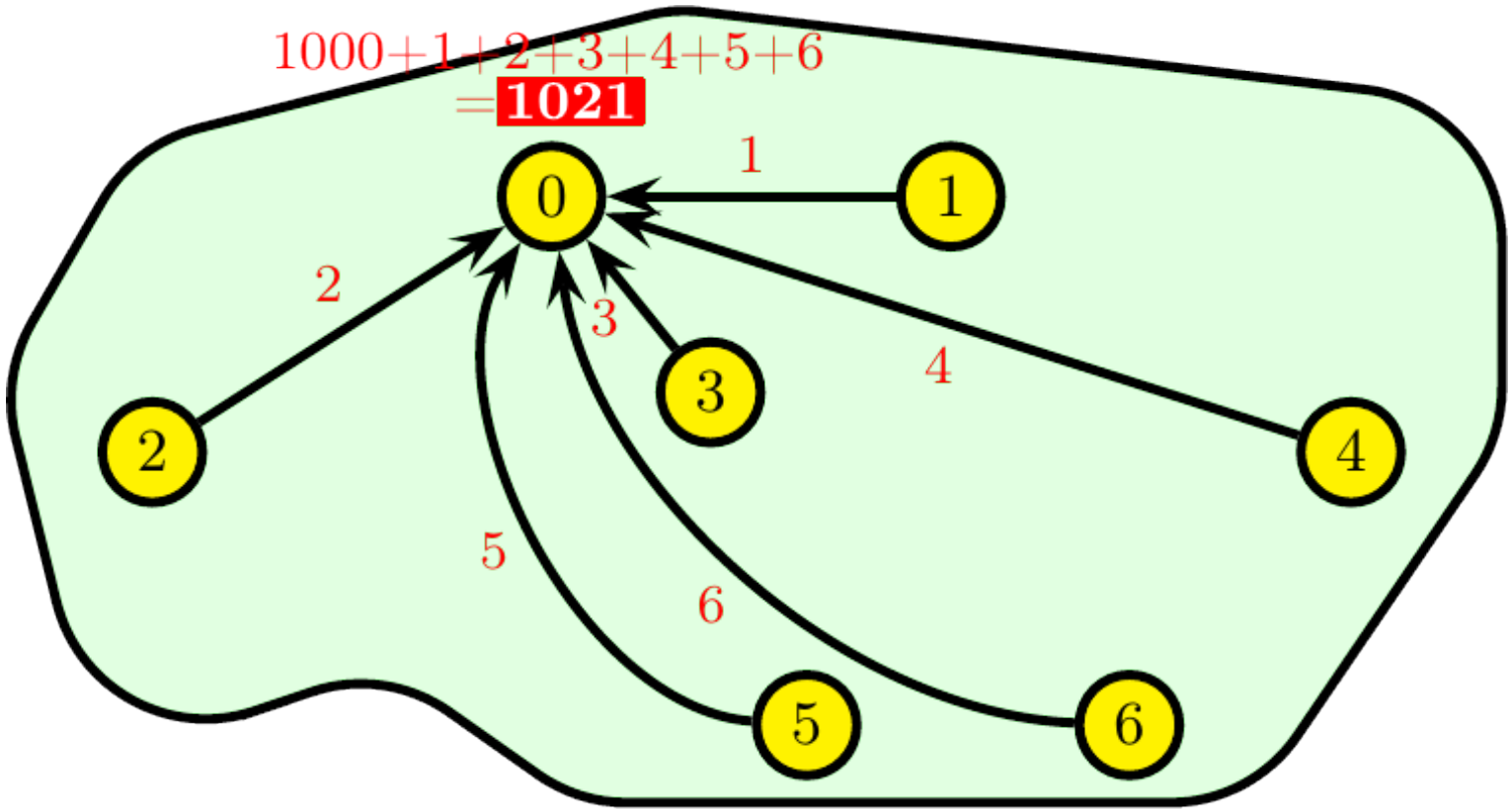


Figure 19 – Réduction répartie : `MPI_Reduce()` avec l'opérateur somme

Communications collectives

Opérations pour réductions réparties

Nom	Opération
MPI_SUM	Somme des éléments
MPI_PROD	Produit des éléments
MPI_MAX	Recherche du maximum
MPI_MIN	Recherche du minimum
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

Communications collectives

Réductions réparties : MPI_Reduce()

```
MPI_REDUCE(message_emis,message_recu,longueur,type_message,operation,rang_dest,comm,code)

TYPE(*), dimension(..), intent(in ) :: message_emis
TYPE(*), dimension(..)              ::  message_recu
integer,integer(in)                 :: longueur, rang_dest
TYPE(MPI_Datatype), intent(in)      :: type_message
TYPE(MPI_Op), intent(in)            :: operation
TYPE(MPI_Comm), intent(in)          :: comm
integer, optional, intent(out)      :: code
```

1. Réduction répartie des éléments situés à partir de l'adresse message_emis, de taille longueur, de type type_message, pour les processus du communicateur comm,
2. Ecrit le résultat à l'adresse message_recu pour le processus de rang rang_dest.

Communications collectives

Exemple de `MPI_Reduce()` (voir Fig.19)

```
1  program reduce
2    use mpi_f08
3    implicit none
4    integer :: nb_procs,rang,valeur,somme
5
6    call MPI_INIT()
7    call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs)
8    call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
9
10   if (rang == 0) then
11      valeur=1000
12   else
13      valeur=rang
14   endif
15
16   call MPI_REDUCE(valeur,somme,1,MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORLD)
17
18   if (rang == 0) then
19      print *,'Moi, processus 0, ma valeur de la somme globale est ',somme
20   end if
21
22   call MPI_FINALIZE()
23 end program reduce
```

```
> mpiexec -n 7 reduce

Moi, processus 0, ma valeur de la somme globale est 1021
```

Communications collectives

Réductions réparties avec diffusion du résultat : `MPI_Allreduce()`

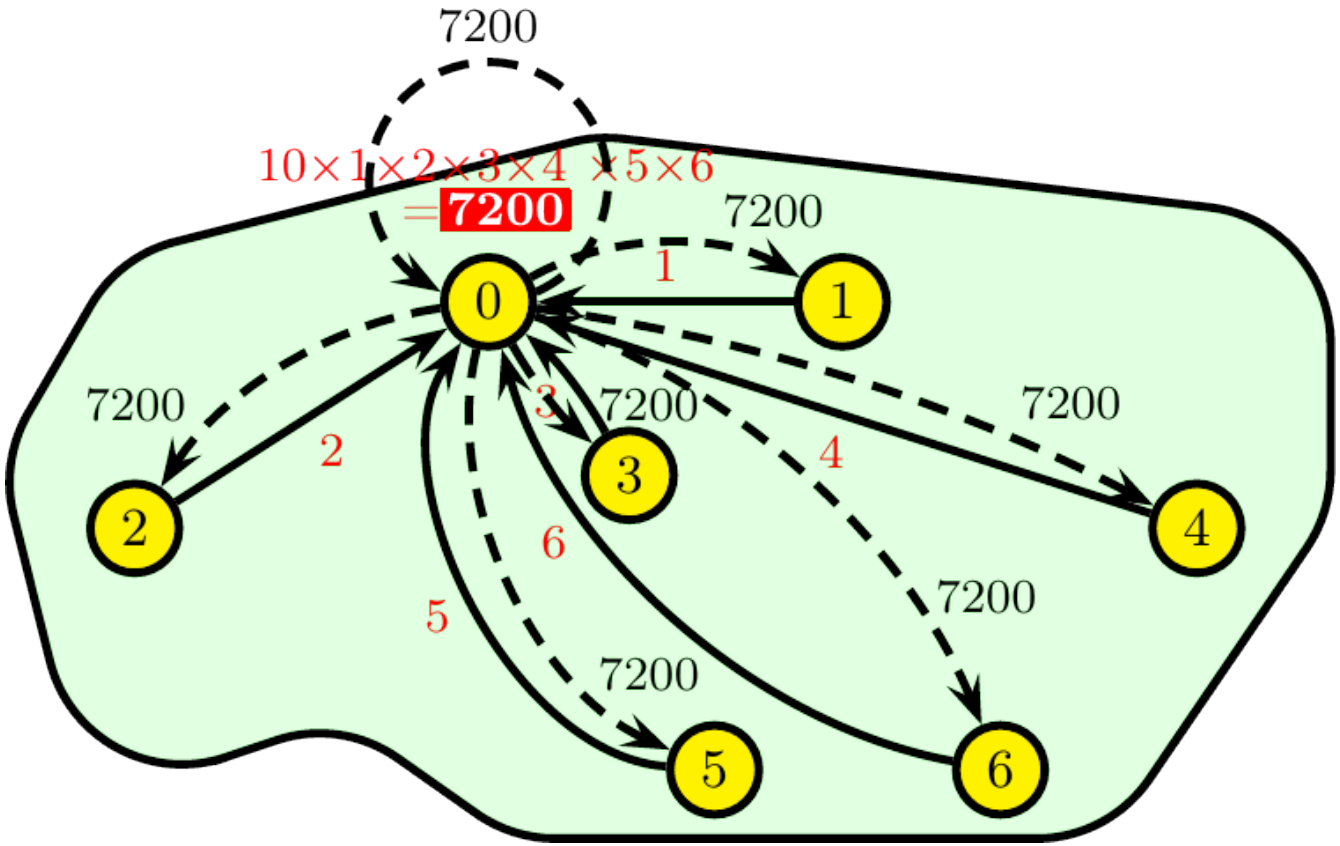


Figure 20 – Réduction répartie avec diffusion du résultat : `MPI_Allreduce` (utilisation de l'opérateur produit)

Communications collectives

Réductions réparties avec diffusion du résultat : `MPI_Allreduce()`

```
MPI_ALLREDUCE(message_emis,message_recu,longueur,type_message,operation,comm,code)

TYPE(*), dimension(..), intent(in) :: message_emis
TYPE(*), dimension(..)             :: message_recu
integer, intent(in)                :: longueur
TYPE(MPI_Datatype), intent(in)     :: type_message
TYPE(MPI_Op), intent(in)           :: operation
TYPE(MPI_Comm), intent(in)         :: comm
integer, optional, intent(out)     :: code
```

1. Réduction répartie des éléments situés à partir de l'adresse message_emis, de taille longueur, de type type_message, pour les processus du communicateur comm,
2. Ecrit le résultat à l'adresse message_recu pour tous les processus du communicateur comm.

Communications collectives

Exemple de `MPI_Allreduce()` (voir Fig.20)

```
1  program allreduce
2
3    use mpi_f08
4    implicit none
5
6    integer :: nb_procs,rang,valeur,produit
7
8    call MPI_INIT()
9    call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs)
10   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
11
12   if (rang == 0) then
13      valeur=10
14   else
15      valeur=rang
16   endif
17
18   call MPI_ALLREDUCE(valeur,produit,1,MPI_INTEGER,MPI_PROD,MPI_COMM_WORLD)
19
20   print *,'Moi, processus ',rang,', ai recu la valeur du produit global ',produit
21
22   call MPI_FINALIZE()
23
24 end program allreduce
```

Communications collectives

Exemple de `MPI_Allreduce()` (voir Fig.20) (suite)

```
> mpiexec -n 7 allreduce

Moi, processus 6, ai recu la valeur du produit global 7200
Moi, processus 2, ai recu la valeur du produit global 7200
Moi, processus 0, ai recu la valeur du produit global 7200
Moi, processus 4, ai recu la valeur du produit global 7200
Moi, processus 5, ai recu la valeur du produit global 7200
Moi, processus 3, ai recu la valeur du produit global 7200
Moi, processus 1, ai recu la valeur du produit global 7200
```

Communications collectives

Compléments

- Le sous-programme `MPI_Scan()` permet d'effectuer des réductions partielles en considérant, pour chaque processus, les processus précédents du communicateur et lui-même. `MPI_Exscan()` est la version *exclusive* de `MPI_Scan()`, qui elle est inclusive.
- Les sous-programmes `MPI_Op_create()` et `MPI_Op_free()` permettent de définir des opérations de réduction personnelles.
- Pour toutes les opérations de reduction, le mot-clé `MPI_IN_PLACE` peut être utilisé pour que les données et résultats de l'opération soient stockés au même endroit (mais uniquement pour le ou les processus qui reçoivent les résultats).
  Exemple : `call MPI_Allreduce(MPI_IN_PLACE,message_emis_et_recu,...).`

Compléments

- De même que ce qui a été vu pour `MPI_Gatherv()` vis-à-vis de `MPI_Gather()`, les sous-programmes `MPI_Scatterv()`, `MPI_Allgatherv()` et `MPI_Alltoallv()` étendent `MPI_Scatter()`, `MPI_Allgather()` et `MPI_Alltoall()` au cas où le nombre d'éléments à diffuser ou collecter est différent suivant les processus.
- `MPI_Alltoallw()` est la version de `MPI_Alltoallv()` permettant de traiter des éléments hétérogènes (en exprimant les déplacements en octets et non en éléments).

T.P. MPI – Exercice 3 : Communications collectives et réductions

- Il s'agit de calculer π par intégration numérique $\pi = \int_0^1 \frac{4}{1+x^2} dx$.
- On utilise la méthode des rectangles (point milieu).
- La fonction à intégrer est $f(x) = \frac{4}{1+x^2}$.
- *nbbloc* est le nombre de points.
- $largeur = \frac{1}{nbbloc}$ est le pas de discrétisation et la largeur de chaque rectangle.
- La version séquentielle est disponible dans le fichier `pi.f90`.
- Il vous faut écrire la version parallélisée avec MPI dans ce fichier.

Modèles de communication

Modèles de communication

Modes d'envoi point à point

Mode	*Bloquant*	*Non bloquant*
Envoi standard	`MPI_Send()`	`MPI_Isend()`
Envoi synchrone	`MPI_Ssend()`	`MPI_Issend()`
Envoi *bufferisé*	`MPI_Bsend()`	`MPI_Ibsend()`
Réception	`MPI_Recv()`	`MPI_Irecv()`

Modèles de communication

Appels bloquants

- Un appel est bloquant si l'espace mémoire servant à la communication peut être réutilisé immédiatement après la sortie de l'appel.
- Les données envoyées peuvent être modifiées après l'appel bloquant.
- Les données reçues peuvent être lues après l'appel bloquant.

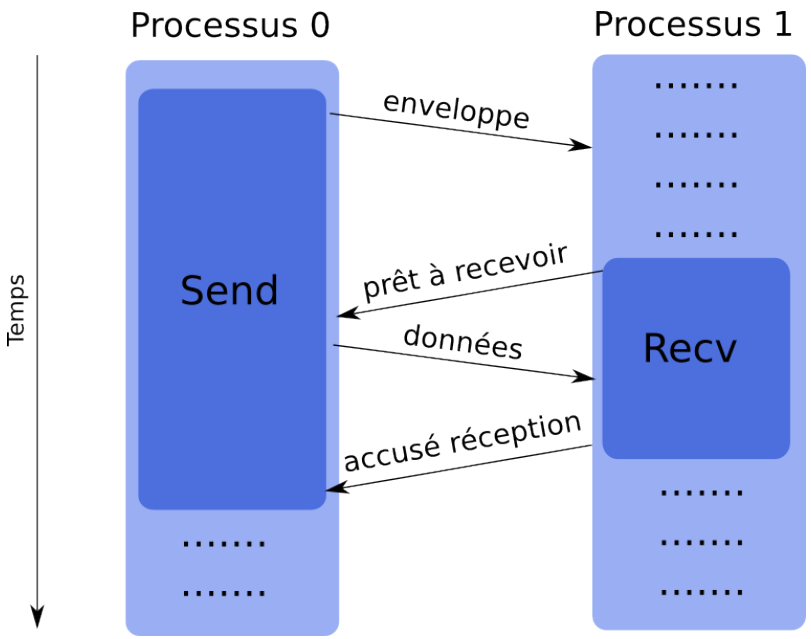
Modèles de communication

Envois synchrones

Un envoi synchrone implique une synchronisation entre les processus concernés. Un envoi ne pourra commencer que lorsque sa réception sera postée. Il ne peut y avoir de communication que si les deux processus sont prêts à communiquer.

Protocole de *rendez-vous*

Le protocole de *rendez-vous* est généralement celui employé pour les envois en mode synchrone (dépend de l'implémentation). L'accusé de réception est optionnel.



Modèles de communication

Interface de `MPI_Ssend()`

```
MPI_SSEND(valeurs, taille, type_message, dest, etiquette, comm, code)

TYPE(*), dimension(..), intent(in) :: valeurs
integer, intent(in)                :: taille, dest, etiquette
TYPE(MPI_Datatype), intent(in)     :: type_message
TYPE(MPI_Comm), intent(in)         :: comm
integer, optional, intent(out)     :: code
```

Avantages

- Consomment peu de ressources (pas de *buffer*)
- Rapides si le récepteur est prêt (pas de recopie dans un *buffer*)
- Connaissance de la réception grâce à la synchronisation

Inconvénients

- Temps d'attente si le récepteur n'est pas là/pas prêt
- Risques d' inter-blocage

Modèles de communication

Exemple d'inter-blocage

Dans l'exemple suivant, on a un inter-blocage, car on est en mode synchrone, les deux processus sont bloqués sur le `MPI_Ssend()` car ils attendent le `MPI_Recv()` de l'autre processus. Or ce `MPI_Recv()` ne pourra se faire qu'après le déblocage du `MPI_Ssend()`.

```
1   program ssendrecv
2     use mpi_f08
3     implicit none
4     integer                              :: rang,valeur,num_proc
5     integer,parameter                    :: etiquette=110
6
7     call MPI_INIT()
8     call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
9
10    ! On suppose avoir exactement 2 processus
11    num_proc=mod(rang+1,2)
12
13    call MPI_SSEND(rang+1000,1,MPI_INTEGER,num_proc,etiquette,MPI_COMM_WORLD)
14    call MPI_RECV(valeur,1,MPI_INTEGER, num_proc,etiquette,MPI_COMM_WORLD, &
15                  MPI_STATUS_IGNORE)
16
17    print *,'Moi, processus',rang,', ai recu',valeur,'du processus',num_proc
18
19    call MPI_FINALIZE()
20  end program ssendrecv
```

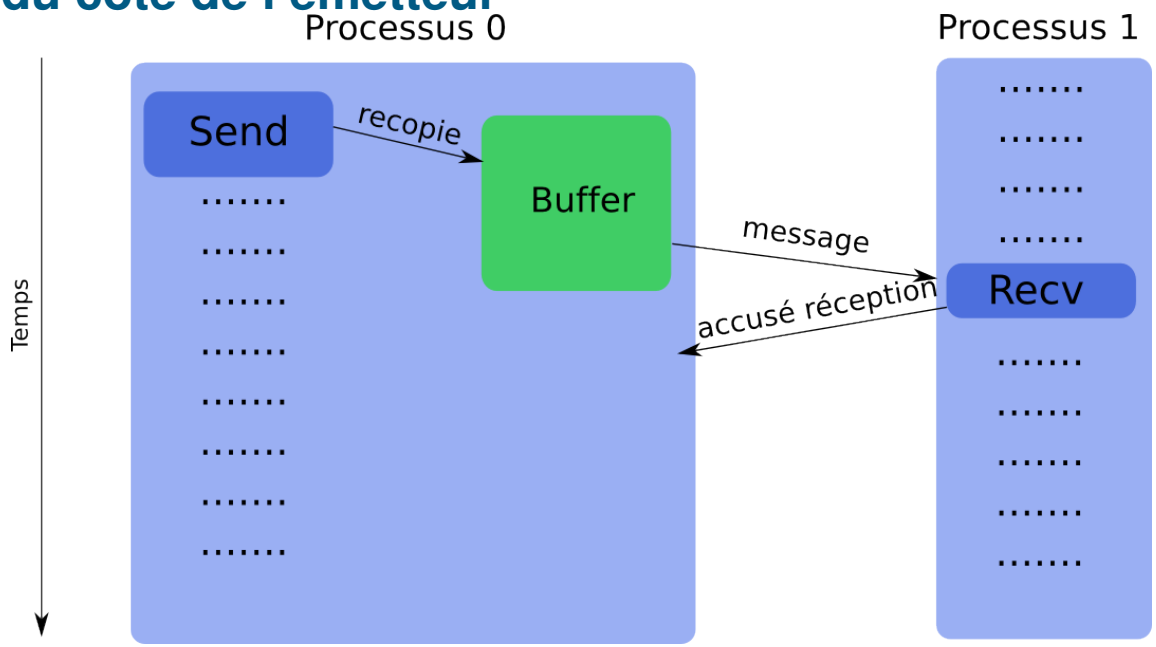
Modèles de communication

Envois *bufferisés*

Un envoi bufferisé implique la recopie des données dans un espace mémoire intermédiaire. Il n'y a alors pas de couplage entre les deux processus de la communication. La sortie de ce type d'envoi ne signifie donc pas que la réception a eu lieu.

Protocole avec *buffer* utilisateur du côté de l'émetteur

Dans cette approche, le *buffer* se trouve du côté de l'émetteur et est géré explicitement par l'application. Un *buffer* géré par MPI peut exister du côté du récepteur. De nombreuses variantes sont possibles. L'accusé de réception est optionnel.



Modèles de communication

Buffers

Les *buffers* doivent être gérés manuellement (avec appels à `MPI_Buffer_attach()` et `MPI_Buffer_detach()`). Ils doivent être alloués en tenant compte des surcoûts mémoire des messages (en ajoutant la constante `MPI_BSEND_OVERHEAD` pour chaque instance de message).

Interfaces

```
MPI_BUFFER_ATTACH(buf, taille_buf, code)
MPI_BUFFER_DETACH(buf_adr, taille_buf, code)

TYPE(*), dimension(..), asynchronous :: buf
TYPE(C_PTR), intent(out)           :: buf_adr
integer                               :: taille_buf
integer, optional, intent(out)        :: code

MPI_BSEND(valeurs, taille, type_message, dest, etiquette, comm, code)

TYPE(*), dimension(..), intent(in)    :: valeurs
integer, intent(in)                   :: taille, dest, etiquette
TYPE(MPI_Datatype), intent(in)        :: type_message
TYPE(MPI_Comm), intent(in)            :: comm
integer, optional, intent(out)        :: code
```

Modèles de communication

Avantages du mode bufferisé

- Pas besoin d'attendre le récepteur (recopie dans un *buffer*)
- Pas de risque de blocage (*deadlocks*)

Inconvénients du mode bufferisé

- Consomment plus de ressources (occupation mémoire par les *buffers* avec risques de saturation)
- Les *buffers* d'envoi doivent être gérés manuellement (souvent délicat de choisir une taille adaptée)
- Un peu plus lent que les envois synchrones si le récepteur est prêt
- Pas de connaissance de la réception (découplage envoi-réception)
- Risque de gaspillage d'espace mémoire si les *buffers* sont trop sur-dimensionnés
- L'application plante si les *buffers* sont trop petits
- Il y a aussi souvent des *buffers* cachés géré par l'implémentation MPI du côté de l'expéditeur et/ou du récepteur (et consommant des ressources mémoires)

Modèles de communication

Absence d'inter-blocage

Dans l'exemple suivant, on a pas d'inter-blocage, car on est en mode bufferisé. Une fois la copie faite dans le *buffer*, l'appel `MPI_Bsend()` retourne et on passe à l'appel `MPI_Recv()`.

```
1  program bsendrecv
2    use mpi_f08
3    use, INTRINSIC :: ISO_C_BINDING
4    implicit none
5    integer                                :: rang,valeur,num_proc,taille,surcout,&
6                                              taille_buf
7    integer,parameter                      :: etiquette=110, nb_elt=1, nb_msg=1
8    integer,dimension(:), allocatable      :: buffer
9    TYPE(C_PTR)                            :: p
10
11   call MPI_INIT()
12   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
13
14   call MPI_TYPE_SIZE(MPI_INTEGER,taille)
15   ! Convertir taille MPI_BSEND_OVERHEAD (octets) en nombre d'integer
16   surcout = int(1+(MPI_BSEND_OVERHEAD*1.)/taille)
17   allocate(buffer(nb_msg*(nb_elt+surcout)))
18   taille_buf = taille * nb_msg * (nb_elt+surcout)
19   call MPI_BUFFER_ATTACH(buffer,taille_buf)
20   ! On suppose avoir exactement 2 processus
21   num_proc=mod(rang+1,2)
22   call MPI_BSEND(rang+1000,nb_elt,MPI_INTEGER,num_proc,etiquette,MPI_COMM_WORLD)
23   call MPI_RECV(valeur,nb_elt,MPI_INTEGER, num_proc,etiquette,MPI_COMM_WORLD, &
24                 MPI_STATUS_IGNORE)
25
26   print *,'Moi, processus',rang,', ai recu',valeur,'du processus',num_proc
27   call MPI_BUFFER_DETACH(p,taille_buf)
28   call MPI_FINALIZE()
29 end program bsendrecv
```

Modèles de communication

Envois standards

Un envoi standard se fait en appelant le sous-programme `MPI_Send()`. Dans la plupart des implémentations, ce mode passe d'un mode *bufferisé* (*eager*) à un mode synchrone lorsque la taille des messages croît.

Interfaces

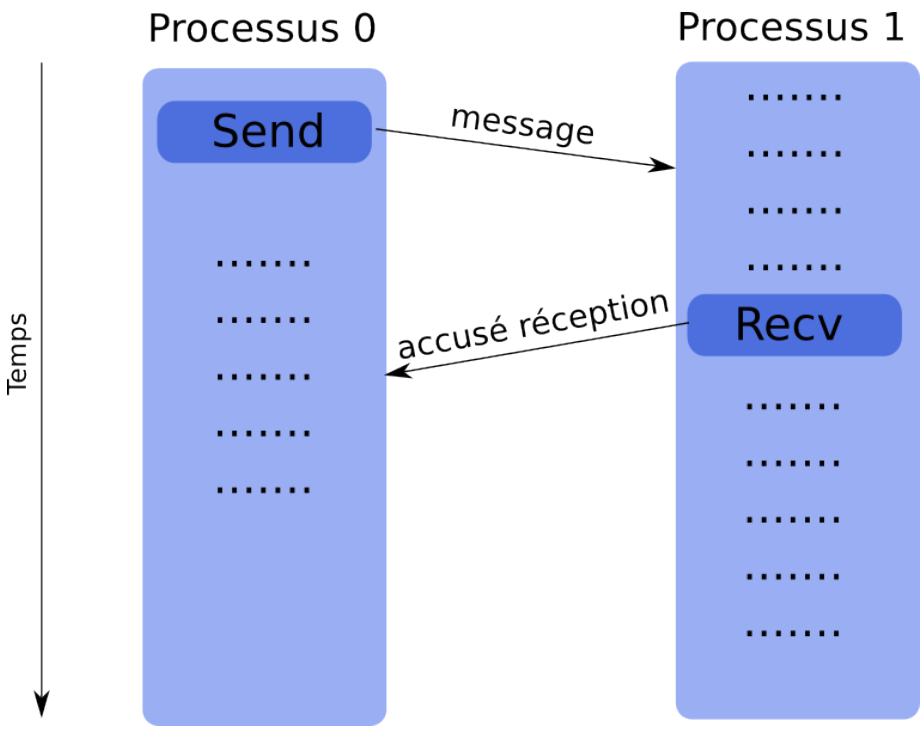
```
MPI_SEND(valeurs, taille, type_message, dest, etiquette, comm, code)

TYPE(*), dimension(..), intent(in) :: valeurs
integer, intent(in)                :: taille, dest, etiquette
TYPE(MPI_Datatype), intent(in)     :: type_message
TYPE(MPI_Comm), intent(in)         :: comm
integer, optional, intent(out)     :: code
```

Modèles de communication

Protocole *eager*

Le protocole *eager* est souvent employé pour les envois en mode standard (`MPI_Send()`) pour les messages de petites tailles. Il peut aussi être utilisé pour les envois avec `MPI_Bsend()` avec des petits messages (dépend de l'implémentation) et en court-circuitant le *buffer* utilisateur du côté de l'émetteur. Dans cette approche, le *buffer* se trouve du côté du récepteur. L'accusé de réception est optionnel.



Modèles de communication

Avantages du mode standard

- Souvent le plus performant (choix du mode le plus adapté par le constructeur)

Inconvénients du mode standard

- Peu de contrôle sur le mode réellement utilisé (souvent accessible via des variables d'environnement)
- Risque de *deadlock* selon le mode réel
- Comportement pouvant varier selon l'architecture et la taille du problème

Modèles de communication

Nombre d'éléments reçus

```
MPI_RECV(message,longueur,type_message,rang_source,etiquette,comm,statut,code)

TYPE(*), dimension(..)          :: message
integer                         :: longueur, rang_source, etiquette
TYPE(MPI_Datatype), intent(in)  :: type_message
TYPE(MPI_Comm), intent(in)      :: comm
TYPE(MPI_Status)                :: statut
integer, optional, intent(out)  :: code
```

- Dans l'appel à `MPI_Recv()`, l'argument longueur correspond dans la norme au nombre d'éléments dans le buffer message.
- Ce nombre doit être supérieur au nombre d'éléments à recevoir.
- Quand c'est possible, pour des raisons de lisibilité, il est conseillé de mettre le nombre d'éléments à recevoir.
- On peut connaitre le nombre d'éléments reçus avec `MPI_Get_count()` et à l'aide de l'argument statut retourné par l'appel à `MPI_Recv()`.

```
MPI_GET_COUNT(statut,type_message,longueur,code)

TYPE(MPI_Status), intent(in)    :: statut
TYPE(MPI_Datatype), intent(in)  :: type_message
integer, intent(out)            :: longueur
integer, optional, intent(out)  :: code
```

Modèles de communication

Nombre d'éléments reçus

MPI_Probe permet de vérifier les messages entrants sans les recevoir.

```
MPI_PROBE(source,tag,comm,statut,code)

integer, intent(in)             :: source, tag
TYPE(MPI_Comm), intent(in)      :: comm
TYPE(MPI_Status)                :: statut
integer, optional, intent(out) :: code
```

Une utilisation courante de MPI_Probe consiste à allouer de l'espace pour un message avant de le recevoir.

```
call MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm, statut)
call MPI_Get_count(statut, MPI_INTEGER, msgsize)
allocate(buf(msgsize))
call MPI_Recv(buf, msgsize, MPI_INTEGER, statut%MPI_SOURCE,
              statut%MPI_TAG, comm, MPI_STATUS_IGNORE)
```

Modèles de communication

Présentation

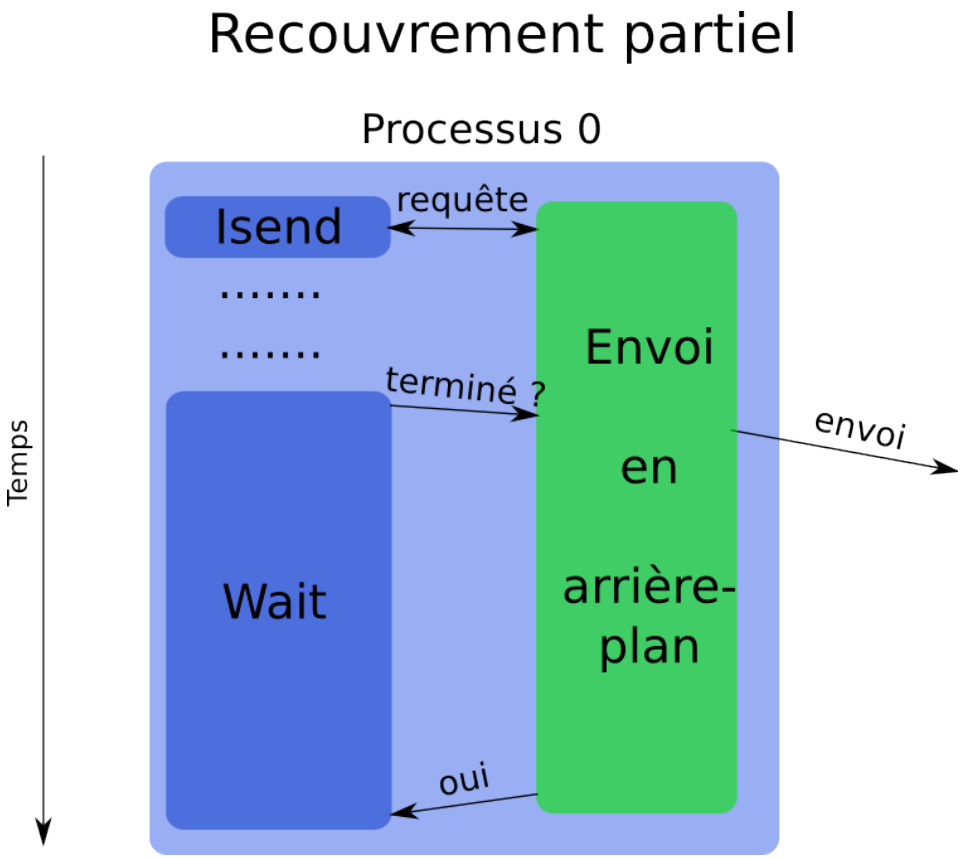
Le recouvrement des communications par des calculs est une méthode permettant de réaliser des opérations de communications en arrière-plan pendant que le programme continue de s'exécuter. Sur Jean Zay, la latence d'une communication inter-nœud est de 1μs soit 2500 cycles processeur.

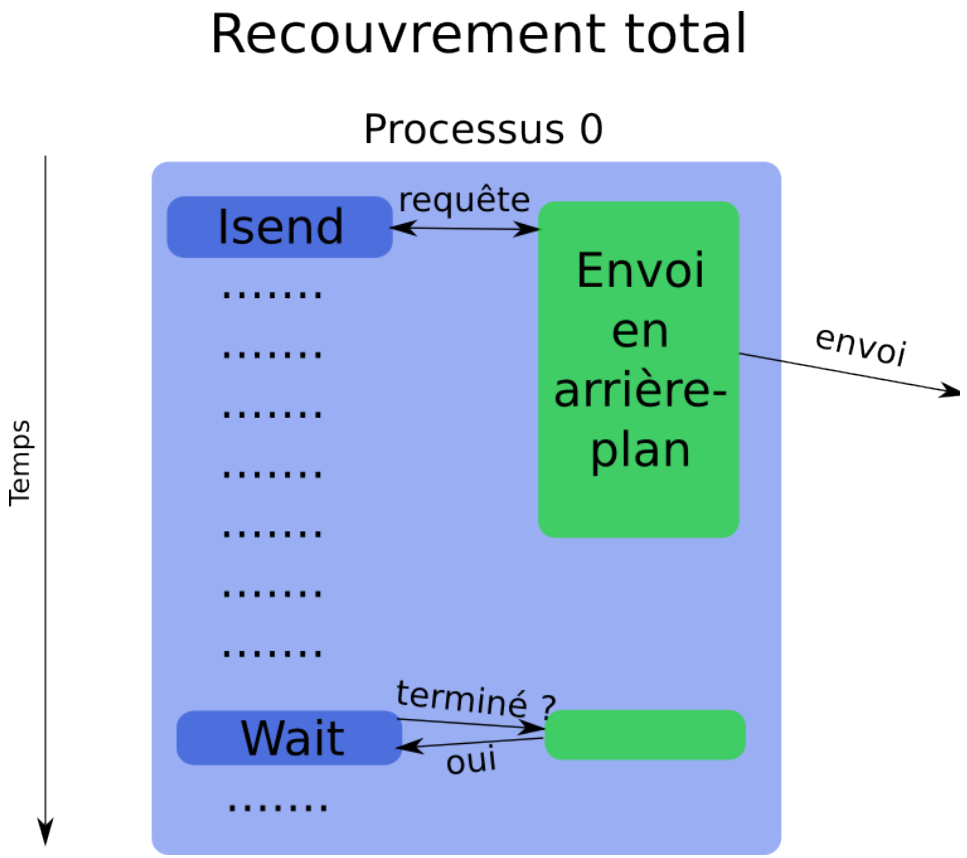
- Il est ainsi possible, si l'architecture matérielle et logicielle le permet, de masquer tout ou une partie des coûts de communications.
- Le recouvrement calculs-communications peut être vu comme un niveau supplémentaire de parallélisme.
- Cette approche s'utilise dans MPI par l'utilisation de sous-programmes non-bloquants (i.e. `MPI_Isend()`, `MPI_Irecv()` et `MPI_Wait()`).

Appels non bloquants

Un appel non bloquant rend la main très rapidement, mais n'autorise pas la réutilisation immédiate de l'espace mémoire utilisé dans la communication. Il est nécessaire de s'assurer que la communication est bien terminée (avec `MPI_Wait()` par exemple) avant de l'utiliser à nouveau.

Modèles de communication





Modèles de communication

Avantages des appels non bloquants

- Possibilité de masquer tout ou une partie des coûts des communications (si l'architecture le permet)
- Pas de risques de *deadlock*

Inconvénients des appels non bloquants

- Surcoûts plus importants (plusieurs appels pour un seul envoi ou réception, gestion des requêtes)
- Complexité plus élevée et maintenance plus compliquée
- Peu performant sur certaines machines (par exemple avec transfert commençant seulement à l'appel de `MPI_Wait()`)
- Risque de perte de performance sur les noyaux de calcul (par exemple gestion différenciée entre la zone proche de la frontière d'un domaine et la zone intérieure entraînant une moins bonne utilisation des caches mémoires)
- Limité aux communications point à point (a été étendu aux collectives dans MPI 3.0)

Modèles de communication

Interfaces

`MPI_Isend()` `MPI_Issend()` et `MPI_Ibsend()` pour les envois non bloquants

```
MPI_ISEND(valeurs, taille, type_message, dest, etiquette, comm, req, code)
MPI_ISSEND(valeurs, taille, type_message, dest, etiquette, comm, req, code)
MPI_IBSEND(valeurs, taille, type_message, dest, etiquette, comm, req, code)

TYPE(*), dimension(..), intent(in), asynchronous  :: valeurs
integer, intent(in)                               :: taille, dest, etiquette
TYPE(MPI_Datatype), intent(in)                    :: type_message
TYPE(MPI_Comm), intent(in)                        :: comm
TYPE(MPI_Request), intent(out)                    :: req
integer, optional, intent(out)                    :: code
```

`MPI_Irecv()` pour les réceptions non bloquantes.

```
MPI_IRECV(valeurs, taille, type_message, source, etiquette, comm, req, code)

TYPE(*), dimension(..), intent(in), asynchronous  :: valeurs
integer, intent(in)                               :: taille, source, etiquette
TYPE(MPI_Datatype), intent(in)                    :: type_message
TYPE(MPI_Comm), intent(in)                        :: comm
TYPE(MPI_Request), intent(out)                    :: req
integer, optional, intent(out)                    :: code
```

Modèles de communication

Interfaces

`MPI_Wait()` attend la fin d'une communication. `MPI_Test()` est la version non bloquante.

```
MPI_WAIT(req, statut, code)
MPI_TEST(req, flag, statut, code)

TYPE(MPI_Request), intent(inout) :: req
logical, intent(out)             :: flag
TYPE(MPI_Status)                 :: statut
integer, optional, intent(out)   :: code
```

`MPI_Waitall()` attend la fin de toutes les communications. `MPI_Testall()` est la version non bloquante.

```
MPI_WAITALL(taille, reqs, statuts, code)
MPI_TESTALL(taille, reqs, flag, statuts, code)

integer, intent(in)                  :: taille
TYPE(MPI_Request), dimension(taille) :: reqs
logical, intent(out)                 :: flag
TYPE(MPI_Status), dimension(taille)  :: statuts
integer, optional, intent(out)       :: code
```

Modèles de communication

Interfaces

`MPI_Waitany()` attend la fin d'une communication parmi plusieurs. `MPI_Testany()` est la version non bloquante.

```
MPI_WAITANY(taille, reqs, indice, statut, code)
MPI_TESTANY(taille, reqs, indice, flag, statut, code)

integer, intent(in)                                  :: taille
TYPE(MPI_Request), dimension(taille), intent(inout)  :: reqs
integer, intent(out)                                 :: indice
logical, intent(out)                                 :: flag
TYPE(MPI_Status)                                     :: statut
integer, optional, intent(out)                       :: code
```

`MPI_Waitsome()` attend la fin d'une ou plusieurs communications.
`MPI_Testsome()` est la version non bloquante.

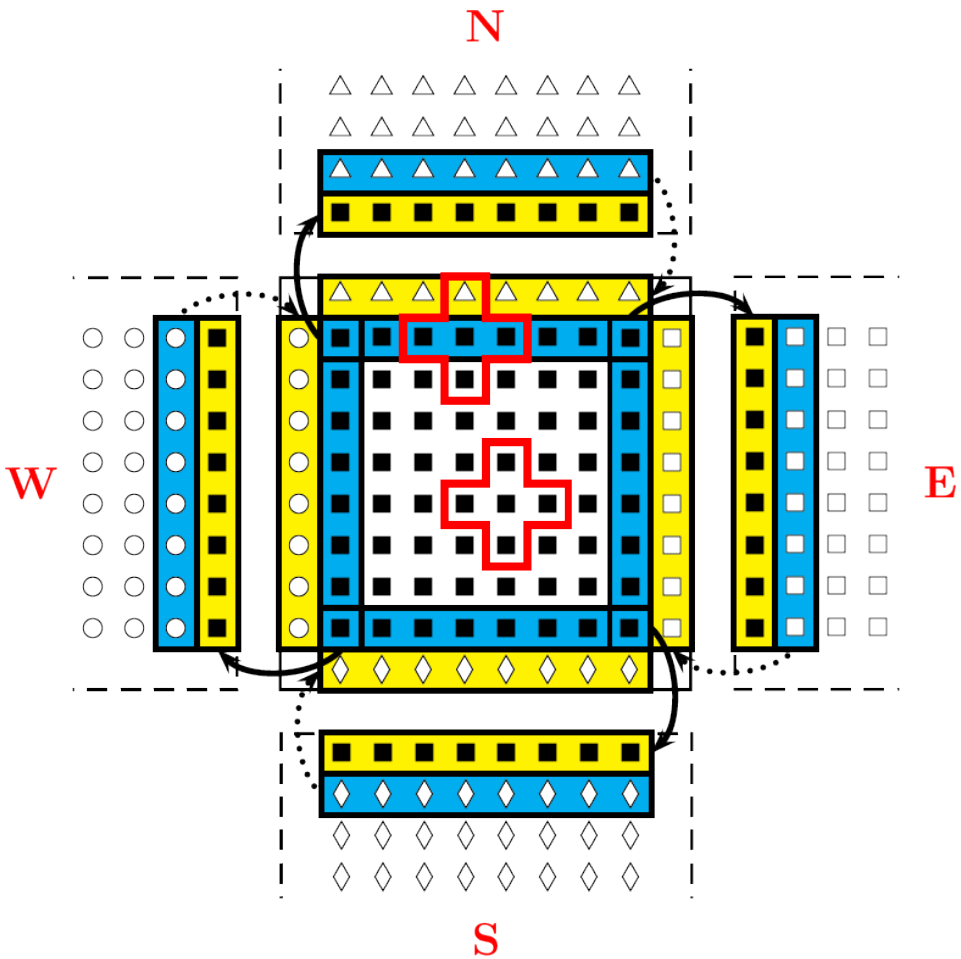
```
MPI_WAITSOME(taille, reqs, nbfin, indices, statuts, code)
MPI_TESTSOME(taille, reqs, nbfin, indices, statuts, code)

integer, intent(in)                                  :: taille
TYPE(MPI_Request), dimension(taille), intent(inout)  :: reqs
integer, intent(out)                                 :: nbfin
integer, dimension(taille), intent(out)              :: indices
TYPE(MPI_Status), dimension(taille), intent(out)     :: statuts
integer, optional, intent(out)                       :: code
```

Gestion des requêtes

- Après un appel aux fonctions bloquantes d'attente (`MPI_Wait()`, `MPI_Waitall()`,...), la requête vaut `MPI_REQUEST_NULL`.
- De même après un appel aux fonctions non bloquantes d'attente lorsque le *flag* est à vrai.
- Une attente avec une requête qui vaut `MPI_REQUEST_NULL` ne fait rien.

Modèles de communication



Modèles de communication

```
1   SUBROUTINE debut_communication(u)
2     !Envoi au voisin N et reception du voisin S
3     CALL MPI_IRECV( u(,), 1, type_ligne, voisin(S), &
4          etiquette, comm2d, requete(1))
5     CALL MPI_ISEND( u(,), 1, type_ligne, voisin(N), &
6          etiquette, comm2d, requete(2))
7
8     !Envoi au voisin S et reception du voisin N
9     CALL MPI_IRECV( u(,), 1,type_ligne,voisin(N), &
10         etiquette, comm2d, requete(3))
11    CALL MPI_ISEND(  u(,), 1,type_ligne,voisin(S), &
12         etiquette,comm2d, requete(4))
13
14    !Envoi au voisin W et  reception du voisin E
15    CALL MPI_IRECV( u(,), 1, type_colonne , voisin(E), &
16         etiquette, comm2d, requete(5))
17    CALL MPI_ISEND( u(,), 1, type_colonne , voisin(W), &
18         etiquette,comm2d, requete(6))
19
20    !Envoi au voisin E et  reception du voisin W
21    CALL MPI_IRECV( u(,), 1, type_colonne, voisin(W), &
22         etiquette, comm2d, requete(7))
23    CALL MPI_ISEND( u(,), 1, type_colonne, voisin(E), &
24         etiquette, comm2d, requete(8))
25  END SUBROUTINE debut_communication
26  SUBROUTINE fin_communication(u)
27    CALL MPI_WAITALL(2*NB_VOISINS,requete,tab_statut)
28    if (.not. MPI_ASYNC_PROTECTS_NONBLOCKING) call MPI_F_SYNC_REG(u)
29  END SUBROUTINE fin_communication
```

Modèles de communication

```
1   DO WHILE ((.NOT. convergence) .AND. (it < it_max))
2     it = it +1
3     u(sx:ex,sy:ey) = u_nouveau(sx:ex,sy:ey)
4
5     !Echange des points aux interfaces pour u a l'iteration n
6     CALL debut_communication( u )
7
8     !Calcul de u a l'iteration n+1
9     CALL calcul( u, u_nouveau, sx+1, ex-1, sy+1, ey-1)
10
11    CALL fin_communication( u )
12
13    ! Nord
14    CALL calcul( u, u_nouveau, sx, sx, sy, ey)
15    ! Sud
16    CALL calcul( u, u_nouveau, ex, ex, sy, ey)
17    ! Ouest
18    CALL calcul( u, u_nouveau, sx, ex, sy, sy)
19    ! Est
20    CALL calcul( u, u_nouveau, sx, ex, ey, ey)
21
22    !Calcul de l'erreur globale
23    diffnorm =  erreur_globale (u, u_nouveau)
24    !Arret du programme si on a atteint la precision machine obtenu
25    !par la fonction F90 EPSILON
26    convergence = ( diffnorm < eps )
27
28  END DO
```

Modèles de communication

Niveau de recouvrement sur différentes machines

Machine	*Niveau*
Zay(IntelMPI)	43%
Zay(IntelMPI) I_MPI_ASYNC_PROGRESS=yes	95%

Mesures faites en recouvrant un noyau de calcul et un noyau de communication de mêmes durées.

Un recouvrement de 0% signifie que la durée totale d'exécution vaut 2x la durée d'un noyau de calcul (ou communication).
Un recouvrement de 100% signifie que la durée totale vaut 1x la durée d'un noyau de calcul (ou communication).

Modèles de communication

Communications collectives non bloquantes

- Version non bloquante des communications collectives
- Avec un I (immediate) devant : `MPI_Ireduce()`, `MPI_Ibcast()`, ...
- Attente avec les appels `MPI_Wait()`, `MPI_Test()` et leurs variantes
- Pas de correspondance bloquant et non bloquant
- Le *status* récupéré par `MPI_Wait()` contient une valeur non définie pour `MPI_SOURCE` et `MPI_TAG`
- Pour les processus d'un communicateur donné, l'ordre des appels doit être le même (comme en version bloquante)

```
MPI_IBARRIER(comm, request, code)

TYPE(MPI_Comm), intent(in)      :: comm,
TYPE(MPI_Request), intent(out) :: request
integer, optional, intent(out) :: code
```

Modèles de communication

Exemple d'utilisation du `MPI_Ibarrier`
Comment gérer les communications quand on ne sait pas à chaque itération si nos voisins vont envoyer un message.

```
logical isAllFinish=.false.
logical isMySendFinish=.false.
do i=1,m
  ! Envoi synchrone
  call MPI_ISSEND(sbuf(i),ssize(i),datatype,dst(i),tag,comm,reqs(i))
end do

do while (.not. isAllFinish)
  ! Avons nous un message pret a etre recu
  call MPI_IPROBE(MPI_ANY_SOURCE,tag,comm,flag,astat)
  if (flag) then
    ! Recoit le message
    call MPI_RECV(rbuf,rsize,datatype,astat%MPI_SOURCE,tag,comm,rstat)
  end if
  if (.not. isMySendFinish) then
    ! Verifie si tous nos ssend sont termines
    call MPI_TESTALL(m,reqs,flag,MPI_STATUSES_IGNORE)
    if (flag) then
      ! Si c'est le cas on demarre la ibarrier
      call MPI_IBARRIER(comm,reqb)
      isMySendFinish=.true.
    end if
  else
    ! Test si tout le monde a fait la ibarrier
    call MPI_TEST(reqb,isAllFinish,MPI_STATUS_IGNORE)
  end if
end do
```

Modèles de communication

MPI-3 : fonctionnalités ajoutées

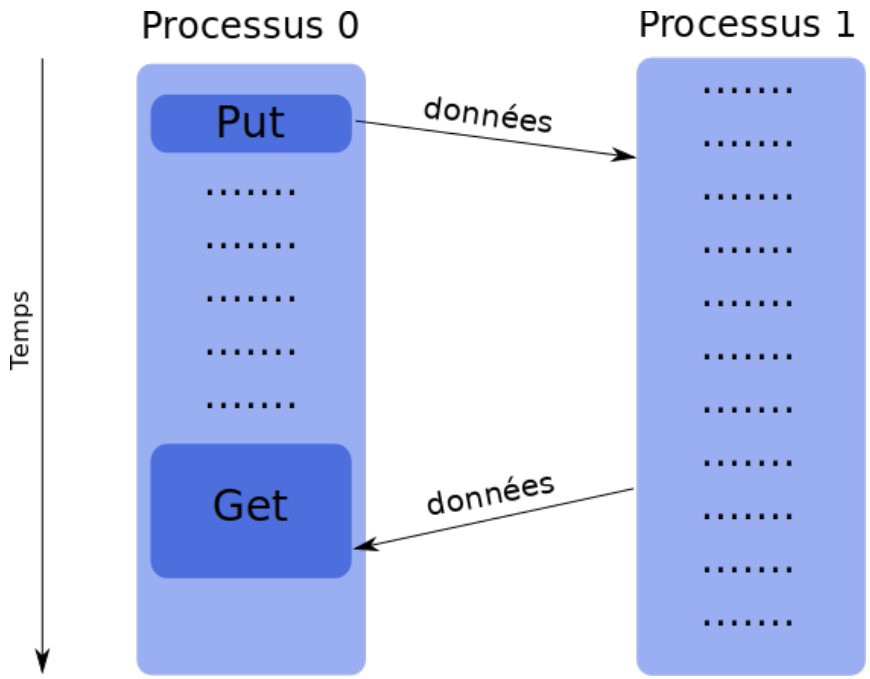
- Si MPI_SUBARRAYS_SUPPORTED est à *true*, il est possible d'utiliser des sections de tableaux pour les appels non bloquants.
- Si MPI_ASYNC_PROTECTS_NONBLOCKING est à *true*, les arguments d'envoi et/ou de réception sont *asynchronous* pour les interfaces des appels non bloquants.

```
call MPI_ISEND(buf,...,req)
...
call MPI_WAIT(req,...)
if (.not. MPI_ASYNC_PROTECTS_NONBLOCKING) call MPI_F_SYNC_REG(buf)
buf = val2
```

Modèles de communication

Communications mémoire à mémoire (RMA)

Les communications mémoire à mémoire (ou RMA pour *Remote Memory Access* ou *one sided communications*) consistent à accéder en écriture ou en lecture à la mémoire d'un processus distant sans que ce dernier doive gérer cet accès explicitement. Le processus cible n'intervient donc pas lors du transfert.



Modèles de communication

RMA - Approche générale

- Création d'une fenêtre mémoire avec `MPI_Win_create()` pour autoriser les transferts RMA dans cette zone.
- Accès distants en lecture ou écriture en appellant `MPI_Put()`, `MPI_Get()`, `MPI_Accumulate()`, `MPI_Fetch_and_op()`, `MPI_Get_accumulate()` et `MPI_Compare_and_swap()` .
- Libération de la fenêtre mémoire avec `MPI_Win_free()`.

RMA - Méthodes de synchronisation

Pour s'assurer d'un fonctionnement correct, il est obligatoire de réaliser certaines synchronisations. 3 méthodes sont disponibles :

- Communication à cible active avec synchronisation globale (`MPI_Win_fence()`) ;
- Communication à cible active avec synchronisation par paire (`MPI_Win_start()` et `MPI_Win_complete()` pour le processus origine ; `MPI_Win_post()` et `MPI_Win_wait()` pour le processus cible) ;
- Communication à cible passive sans intervention de la cible (`MPI_Win_lock()` et `MPI_Win_unlock()`).

Modèles de communication

```
1   program ex_fence
2     use mpi_f08
3     implicit none
4
5     integer, parameter :: assert=0
6     integer :: code, rang, taille_reel, i, nb_elements, cible, m=4, n=4
7     TYPE(MPI_Win) :: win
8     integer (kind=MPI_ADDRESS_KIND) :: deplacement, dim_win
9     real(kind=kind(1.d0)), dimension(:), allocatable :: win_local, tab
10
11    call MPI_INIT()
12    call MPI_COMM_RANK(MPI_COMM_WORLD, rang)
13    call MPI_TYPE_SIZE(MPI_DOUBLE_PRECISION,taille_reel)
14
15    if (rang==0) then
16       n=0
17       allocate(tab(m))
18    endif
19
20    allocate(win_local(n))
21    dim_win = taille_reel*n
22
23    call MPI_WIN_CREATE(win_local, dim_win, taille_reel, MPI_INFO_NULL, &
24                        MPI_COMM_WORLD, win)
```

Modèles de communication

```
25    if (rang==0) then
26       tab(:) = (/ (i, i=1,m) /)
27    else
28       win_local(:) = 0.0
29    end if
30
31    call MPI_WIN_FENCE(assert,win)
32    if (rang==0) then
33       cible = 1; nb_elements = 2; deplacement = 1
34       call MPI_PUT(tab, nb_elements, MPI_DOUBLE_PRECISION, cible, deplacement, &
35                    nb_elements, MPI_DOUBLE_PRECISION, win)
36    end if
37
38    call MPI_WIN_FENCE(assert,win)
39    if (rang==0) then
40       tab(m) = sum(tab(1:m-1))
41    else
42       win_local(n) = sum(win_local(1:n-1))
43    endif
44
45    call MPI_WIN_FENCE(assert,win)
46    if (rang==0) then
47       nb_elements = 1; deplacement = m-1
48       call MPI_GET(tab, nb_elements, MPI_DOUBLE_PRECISION, cible, deplacement, &
49                    nb_elements, MPI_DOUBLE_PRECISION, win)
50    end if
```

Modèles de communication

Avantages des RMA

- Permet de mettre en place plus efficacement certains algorithmes.
- Plus performant que les communications point à point sur certaines machines (utilisation de matériels spécialisés tels que moteur DMA, coprocesseur, mémoire spécialisée...).
- Possibilité pour l'implémentation de regrouper plusieurs opérations.

Inconvénients des RMA

- La gestion des synchronisations est délicate.
- Complexité et risques d'erreurs élevés.
- Pour les synchronisations cible passive, obligation d'allouer la mémoire avec `MPI_Alloc_mem()` qui ne respecte pas la norme Fortran (utilisation de pointeurs Cray non supportés par certains compilateurs).
- Moins performant que les communications point à point sur certaines machines.

Types de données dérivés

Types de données dérivés

Introduction

- Dans les communications, les données échangées sont typées : `MPI_INTEGER`, `MPI_REAL`, `MPI_COMPLEX`, etc .
- On peut créer des structures de données plus complexes à l'aide de sous-programmes tels que `MPI_Type_contiguous()`, `MPI_Type_vector()`, `MPI_Type_indexed()` ou `MPI_Type_create_struct()`.
- Les types dérivés permettent notamment l'échange de données non contiguës ou non homogènes en mémoire et de limiter le nombre d'appels aux sous-programmes de communications.

Types de données dérivés

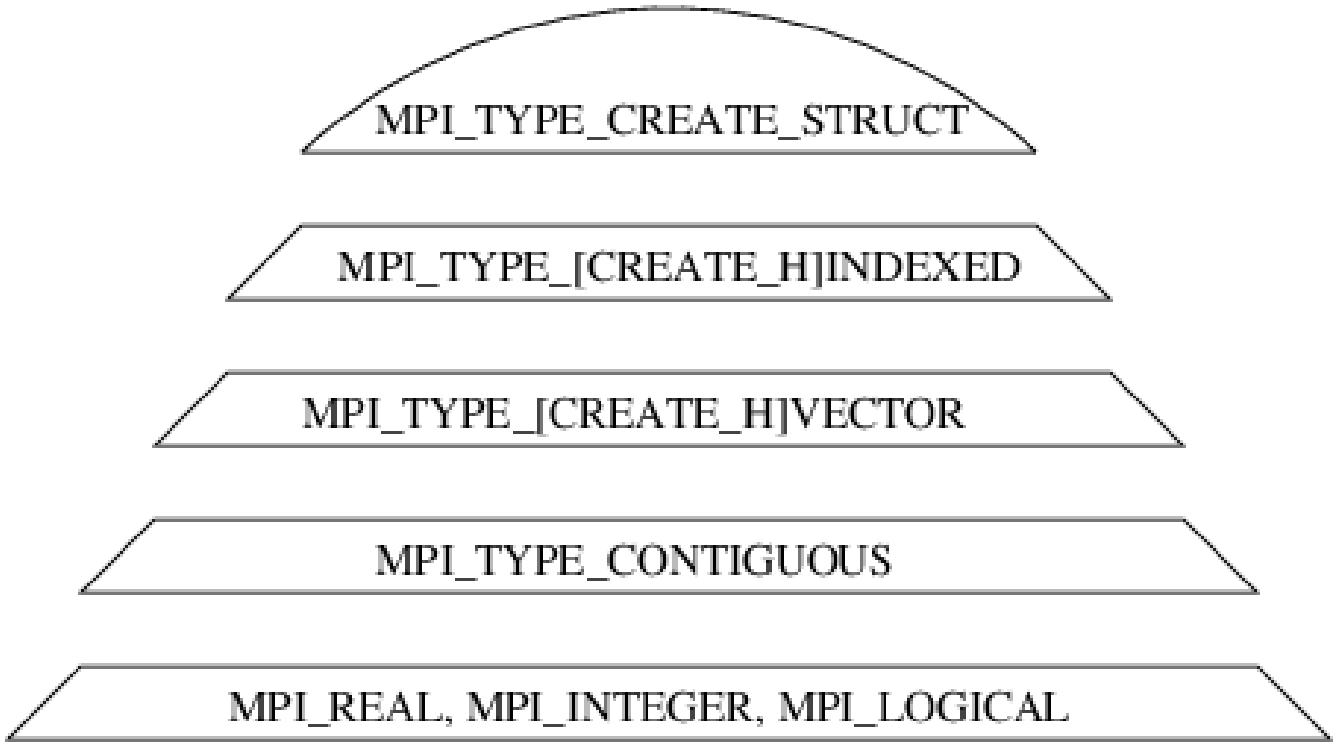


Figure 21 – Hiérarchie des constructeurs de type MPI

Types de données dérivés

Types contigus

- `MPI_Type_contiguous()` crée une structure de données à partir d'un ensemble homogène de type préexistant de données contiguës en mémoire.

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,code)
```

Figure 22 – Sous-programme MPI_Type_contiguous

```
MPI_TYPE_CONTIGUOUS(nombre,ancien_type,nouveau_type,code)

integer, intent(in)              :: nombre
TYPE(MPI_Datatype), intent(int)  :: ancien_type
TYPE(MPI_Datatype), intent(out)  :: nouveau_type
integer, optional, intent(out)   :: code
```

Types de données dérivés

Types avec un pas constant

- `MPI_Type_vector()` crée une structure de données à partir d'un ensemble homogène de type préexistant de données distantes d'un pas constant en mémoire. Le pas est donné en nombre d'éléments.

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_VECTOR(6,1,5,MPI_REAL,nouveau_type,code)
```

Figure 23 – Sous-programme MPI_Type_vector

```
MPI_TYPE_VECTOR(nombre_bloc,nbr_elt_par_bloc,pas,type_elt,nouveau_type,code)

integer, intent(in)              :: nombre_bloc,nbr_elt_par_bloc
integer, intent(in)              :: pas ! donne en elements
TYPE(MPI_Datatype), intent(in)   :: type_elt
TYPE(MPI_Datatype), intent(out)  :: nouveau_type
integer, optional, intent(out)   :: code
```

Types de données dérivés

Types avec un pas constant

- `MPI_Type_create_hvector()` crée une structure de données à partir d'un ensemble homogène de type préexistant de données distantes d'un pas constant en mémoire. Le pas est donné en nombre d'octets.
- Cette instruction est utile lorsque le type générique n'est plus un type de base (`MPI_INTEGER`, `MPI_REAL`,...) mais un type plus complexe construit à l'aide des sous-programmes MPI, parce qu'alors le pas ne peut plus être exprimé en nombre d'éléments du type générique.

```
MPI_TYPE_CREATE_HVECTOR(nombre_bloc,nbr_elt_par_bloc,pas,
                        type_elt,nouveau_type,code)

integer, intent(in)                          :: nombre_bloc,nbr_elt_par_bloc
integer(kind=MPI_ADDRESS_KIND), intent(in)   :: pas ! donne en octets
TYPE(MPI_Datatype), intent(in)               :: type_elt
TYPE(MPI_Datatype), intent(out)              :: nouveau_type
integer, optional, intent(out)               :: code
```

Types de données dérivés

Validation des types de données dérivés

- Les types dérivés doivent être validés avant d'être utilisé dans une communication. La validation s'effectue à l'aide du sous-programme `MPI_Type_commit()`.

```
MPI_TYPE_COMMIT(nouveau_type,code)

TYPE(MPI_Datatype), intent(inout) :: nouveau_type
integer, optional, intent(out)    :: code
```

- Si on souhaite réutiliser le même nom pour définir un autre type dérivé, on doit au préalable le libérer en utilisant le sous-programme `MPI_Type_free()`.

```
MPI_TYPE_FREE(nouveau_type,code)

TYPE(MPI_Datatype), intent(inout) :: nouveau_type
integer, optional, intent(out)    :: code
```

Types de données dérivés

```
 1  program colonne
 2    use mpi_f08
 3    implicit none
 4
 5    integer, parameter                       :: nb_lignes=5,nb_colonnes=6
 6    integer, parameter                       :: etiquette=100
 7    real, dimension(nb_lignes,nb_colonnes)   :: a
 8    TYPE(MPI_Status)                         :: statut
 9    integer                                  :: rang
10    TYPE(MPI_Datatype)                       :: type_colonne
11
12    call MPI_INIT()
13    call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
14
15    ! Initialisation de la matrice sur chaque processus
16    a(:,:) = real(rang)
17
18    ! Definition du type type_colonne
19    call MPI_TYPE_CONTIGUOUS(nb_lignes,MPI_REAL,type_colonne)
20
21    ! Validation du type type_colonne
22    call MPI_TYPE_COMMIT(type_colonne)
```

Types de données dérivés

```
23    ! Envoi de la premiere colonne
24    if ( rang == 0 ) then
25      call MPI_SEND(a(1,1),1,type_colonne,1,etiquette,MPI_COMM_WORLD)
26
27    ! Reception dans la derniere colonne
28    elseif ( rang == 1 ) then
29      call MPI_RECV(a(1,nb_colonnes),nb_lignes,MPI_REAL,0,etiquette,&
30                    MPI_COMM_WORLD,statut)
31    end if
32
33    ! Libere le type
34    call MPI_TYPE_FREE(type_colonne)
35
36    call MPI_FINALIZE()
37
38  end program colonne
```

Types de données dérivés

```
1  program ligne
2    use mpi_f08
3    implicit none
4
5    integer, parameter                      :: nb_lignes=5,nb_colonnes=6
6    integer, parameter                      :: etiquette=100
7    real, dimension(nb_lignes,nb_colonnes):: a
8    TYPE(MPI_Status)                        :: statut
9    integer                                 :: rang
10   TYPE(MPI_Datatype)                      :: type_ligne
11
12   call MPI_INIT()
13   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
14
15   ! Initialisation de la matrice sur chaque processus
16   a(:,:) = real(rang)
17
18   ! Definition du type type_ligne
19   call MPI_TYPE_VECTOR(nb_colonnes,1,nb_lignes,MPI_REAL,type_ligne)
20
21   ! Validation du type type_ligne
22   call MPI_TYPE_COMMIT(type_ligne)
```

Types de données dérivés

```
23    ! Envoi
24    if ( rang == 0 ) then
25      call MPI_SEND(a(2,1),nb_colonnes,MPI_REAL,1,etiquette,MPI_COMM_WORLD)
26
27    ! Reception dans l'avant-derniere ligne
28    elseif ( rang == 1 ) then
29      call MPI_RECV(a(nb_lignes-1,1),1,type_ligne,0,etiquette,&
30                    MPI_COMM_WORLD,statut)
31    end if
32
33    ! Libere le type type_ligne
34    call MPI_TYPE_FREE(type_ligne)
35
36    call MPI_FINALIZE()
37
38  end program ligne
```

Types de données dérivés

```
1  program bloc
2    use mpi_f08
3    implicit none
4
5    integer, parameter                      :: nb_lignes=5,nb_colonnes=6
6    integer, parameter                      :: etiquette=100
7    integer, parameter                      :: nb_lignes_bloc=2,nb_colonnes_bloc=3
8    real, dimension(nb_lignes,nb_colonnes):: a
9    TYPE(MPI_Status)                        :: statut
10   integer                                 :: rang
11   TYPE(MPI_Datatype)                      :: type_bloc
12
13   call MPI_INIT()
14   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
15
16   ! Initialisation de la matrice sur chaque processus
17   a(:,:) = real(rang)
18
19   ! Creation du type type_bloc
20   call MPI_TYPE_VECTOR(nb_colonnes_bloc,nb_lignes_bloc,nb_lignes,&
21                        MPI_REAL,type_bloc)
22
23   ! Validation du type type_bloc
24   call MPI_TYPE_COMMIT(type_bloc)
```

Types de données dérivés

```
25    ! Envoi d'un bloc
26    if ( rang == 0 ) then
27      call MPI_SEND(a(1,1),1,type_bloc,1,etiquette,MPI_COMM_WORLD)
28
29    ! Reception du bloc
30    elseif ( rang == 1 ) then
31      call MPI_RECV(a(nb_lignes-1,nb_colonnes-2),1,type_bloc,0,etiquette,&
32                    MPI_COMM_WORLD,statut)
33    end if
34
35    ! Liberation du type type_bloc
36    call MPI_TYPE_FREE(type_bloc)
37
38    call MPI_FINALIZE()
39
40  end program bloc
```

Types de données dérivés

Types homogènes à pas variable

- `MPI_Type_indexed()` permet de créer une structure de données composée d'une séquence de blocs contenant un nombre variable d'éléments et séparés par un pas variable en mémoire. Ce dernier est exprimé en éléments.
- `MPI_Type_create_hindexed()` a la même fonctionnalité que `MPI_Type_indexed()` sauf que le pas séparant deux blocs de données est exprimé en octets.
  Cette instruction est utile lorsque le type générique n'est pas un type de base MPI (`MPI_INTEGER`, `MPI_REAL`, ...) mais un type plus complexe construit avec les sous-programmes MPI vus précédemment. On ne peut exprimer alors le pas en nombre d'éléments du type générique d'où le recours à `MPI_Type_create_hindexed()`.
- Pour `MPI_Type_create_hindexed()`, comme pour `MPI_Type_create_hvector()`, utilisez `MPI_Type_size()` ou `MPI_Type_get_extent()` pour obtenir de façon portable la taille du pas en nombre d'octets.

Types de données dérivés

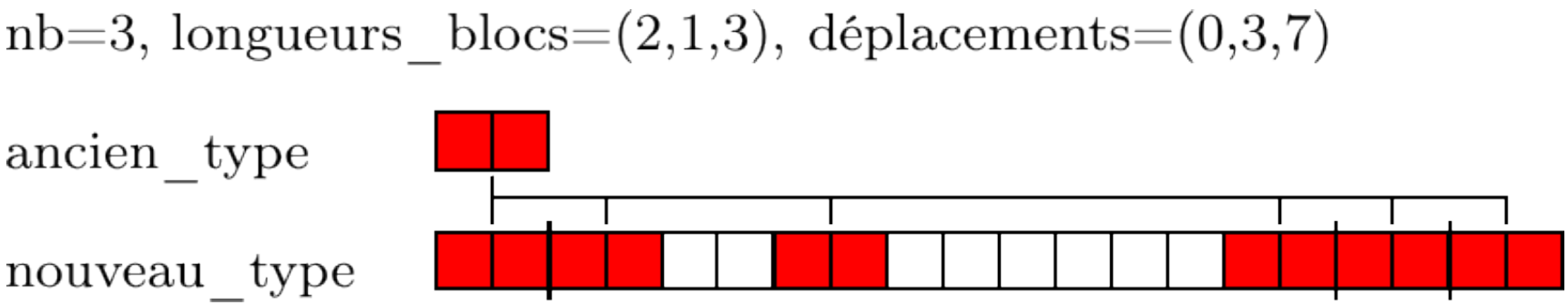


Figure 24 – Le constructeur `MPI_Type_indexed`

```
MPI_TYPE_INDEXED(nb,longueurs_blocs,deplacements,ancien_type,nouveau_type,code)

integer,intent(in)                  :: nb
integer,intent(in),dimension(nb)    :: longueurs_blocs
! Attention les deplacements sont donnes en elements
integer,intent(in),dimension(nb)    :: deplacements
TYPE(MPI_Datatype), intent(in)      :: ancien_type
TYPE(MPI_Datatype), intent(out)     :: nouveau_type
integer, optional, intent(out)      :: code
```

Types de données dérivés

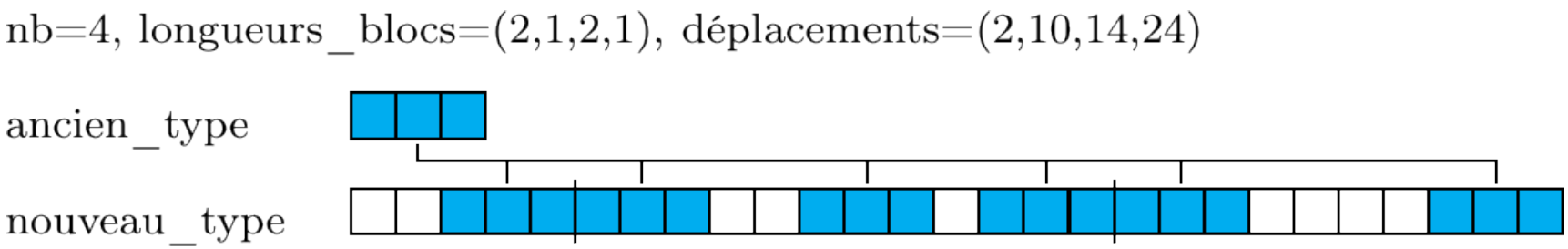


Figure 25 – Le constructeur `MPI_Type_create_hindexed`

```
MPI_TYPE_CREATE_HINDEXED(nb,longueurs_blocs,deplacements,
                         ancien_type,nouveau_type,code)

integer,intent(in)                                          :: nb
integer,intent(in),dimension(nb)                            :: longueurs_blocs
! Attention les deplacements sont donnes en octets
integer(kind=MPI_ADDRESS_KIND),intent(in),dimension(nb) :: deplacements
TYPE(MPI_Datatype), intent(in)                              :: ancien_type
TYPE(MPI_Datatype), intent(out)                             :: nouveau_type
integer, optional, intent(out)                              :: code
```

Types de données dérivés

Exemple : matrice triangulaire

Dans l'exemple suivant, chacun des deux processus :

1. initialise sa matrice (nombres croissants positifs sur le processus 0 et négatifs décroissants sur le processus 1) ;
2. construit son type de données (*datatype*) : matrice triangulaire (supérieure pour le processus 0 et inférieure pour le processus 1) ;
3. envoie sa matrice triangulaire à l'autre et reçoit une matrice triangulaire qu'il stocke à la place de celle qu'il a envoyée. Cela se fait avec l'instruction `MPI_Sendrecv_replace()` ;
4. libère ses ressources et quitte MPI.

Types de données dérivés

Processus 0

AVANT

1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63
8	16	24	32	40	48	56	64

APRÈS

1	-2	-3	-5	-8	-14	-22	-32
2	10	-4	-6	-11	-15	-23	-38
3	11	19	-7	-12	-16	-24	-39
4	12	20	28	-13	-20	-29	-40
5	13	21	29	37	-21	-30	-47
6	14	22	30	38	46	-31	-48
7	15	23	31	39	47	55	-56
8	16	24	32	40	48	56	64

Processus 1

-1	-9	-17	-25	-33	-41	-49	-57
-2	-10	-18	-26	-34	-42	-50	-58
-3	-11	-19	-27	-35	-43	-51	-59
-4	-12	-20	-28	-36	-44	-52	-60
-5	-13	-21	-29	-37	-45	-53	-61
-6	-14	-22	-30	-38	-46	-54	-62
-7	-15	-23	-31	-39	-47	-55	-63
-8	-16	-24	-32	-40	-48	-56	-64

-1	-9	-17	-25	-33	-41	-49	-57
9	-10	-18	-26	-34	-42	-50	-58
17	34	-19	-27	-35	-43	-51	-59
18	35	44	-28	-36	-44	-52	-60
25	36	45	52	-37	-45	-53	-61
26	41	49	53	58	-46	-54	-62
27	42	50	54	59	61	-55	-63
33	43	51	57	60	62	63	-64

Types de données dérivés

```
 1  program triangle
 2    use mpi_f08
 3    implicit none
 4
 5    integer,parameter                  :: n=8,etiquette=100
 6    real,dimension(n,n)                :: a
 7    TYPE(MPI_Status)                   :: statut
 8    integer                            :: i
 9    integer                            :: rang
10    TYPE(MPI_Datatype)                 :: type_triangle
11    integer,dimension(n)               :: longueurs_blocs,deplacements
12
13    call MPI_INIT()
14    call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
15
16    ! Initialisation de la matrice sur chaque processus
17    a(:,:) = reshape( (/ (sign(i,-rang),i=1,n*n) /), (/n,n/))
18
19    ! Creation du type matrice triangulaire sup pour le processus 0
20    ! et du type matrice triangulaire inferieure pour le processus1
21    if (rang == 0) then
22       longueurs_blocs(:) = (/ (i-1,i=1,n) /)
23       deplacements(:)    = (/ (n*(i-1),i=1,n) /)
24    else
25       longueurs_blocs(:) = (/ (n-i,i=1,n) /)
26       deplacements(:)    = (/ (n*(i-1)+i,i=1,n) /)
27    endif
28
29    call MPI_TYPE_INDEXED(n,longueurs_blocs,deplacements,MPI_REAL,type_triangle)
30    call MPI_TYPE_COMMIT(type_triangle)
31
32    ! Permutation des matrices triangulaires superieure et inferieure
33    call MPI_SENDRECV_REPLACE(a,1,type_triangle,mod(rang+1,2),etiquette,mod(rang+1,2), &
34                              etiquette,MPI_COMM_WORLD,statut)
35
36    ! Liberation du type triangle
37    call MPI_TYPE_FREE(type_triangle)
38    call MPI_FINALIZE()
39  end program triangle
```

Types de données dérivés

Taille des types de données

- `MPI_Type_size()` retourne le nombre d'octets nécessaire pour envoyer un type de données. Cette valeur ne tient pas compte des *trous* présents dans le type de données.

```
MPI_TYPE_SIZE(type_message,taille,code)

TYPE(MPI_Datatype), intent(in) :: type_message
integer, intent(out)           :: taille
integer, optional, intent(out) :: code
```

- L'étendue d'un type est l'espace mémoire occupé par le type (en octets). Cette valeur intervient directement pour calculer la position du prochain élément en mémoire (c'est-à-dire le pas entre des éléments successifs).

```
MPI_TYPE_GET_EXTENT(type_message,borne_inf,etendue,code)

TYPE(MPI_Datatype), intent(in)                :: type_message
integer(kind=MPI_ADDRESS_KIND), intent(out) :: borne_inf,etendue
integer, optional, intent(out)                :: code
```

Types de données dérivés

Exemple 1 : `MPI_Type_indexed(2,(/2,1/),(/1,4/),MPI_INTEGER,type,code)`

Type dérivé :

Deux éléments successifs :

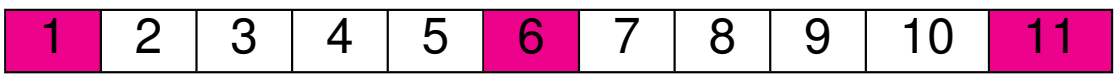
`taille = 12 (3 entiers); borne_inf = 4 (1 entier); etendue = 16 (4 entiers)`

Exemple 2 :
`MPI_Type_vector(3,1,nb_lignes,MPI_INTEGER,type_demi_ligne,code)`

Vue 2D :

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

Vue 1D :



`taille = 12 (3 entiers); borne_inf = 0; etendue = 44 (11 entiers)`

Types de données dérivés

Changer l'étendue

- L'étendue est un paramètre du type de données. Par défaut, c'est généralement l'intervalle en mémoire entre le premier et le dernier composant du type (bornes incluses et en tenant compte de l'alignement mémoire). On peut modifier l'étendue d'un type pour créer un nouveau type adapté du précédent avec `MPI_Type_create_resized()`. Cela permet de choisir le pas entre des éléments successifs.

```
MPI_TYPE_CREATE_RESIZED(ancien_type,nouvelle_borne_inf,nouvelle_etendue,
                        nouveau_type,code)

TYPE(MPI_Datatype), intent(in)              :: ancien_type
integer(kind=MPI_ADDRESS_KIND), intent(in) :: nouvelle_borne_inf,nouvelle_etendue
TYPE(MPI_Datatype), intent(out)             :: nouveau_type
integer, optional, intent(out)              :: code
```

Types de données dérivés

```
1  program demi_ligne
2    use mpi_f08
3    implicit none
4    integer, parameter                          :: nb_lignes=5,nb_colonnes=6, &
5                                                   taille_demi_ligne=nb_colonnes/2,etiquette=1000
6    integer, dimension(nb_lignes,nb_colonnes) :: a
7    integer                                     :: rang,i,taille_integer
8    TYPE(MPI_Datatype)                          :: type_demi_ligne1,type_demi_ligne2
9    integer(kind=MPI_ADDRESS_KIND)              :: borne_inf1,etendue1,borne_inf2,etendue2
10   TYPE(MPI_Status)                            :: statut
11
12   call MPI_INIT()
13   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
14
15   ! Initialisation de la matrice A sur chaque processus
16   a(:,:) = reshape( (/ (sign(i,-rang),i=1,nb_lignes*nb_colonnes) /), &
17                     (/ nb_lignes,nb_colonnes /) )
18
19   ! Construction du type derive type_demi_ligne1
20   call MPI_TYPE_VECTOR(taille_demi_ligne,1,nb_lignes,MPI_INTEGER,type_demi_ligne1)
21
22   ! Connaitre la taille du type de base MPI_INTEGER
23   call MPI_TYPE_SIZE(MPI_INTEGER,taille_integer)
24
25   ! Informations sur le type derive type_demi_ligne1
26   call MPI_TYPE_GET_EXTENT(type_demi_ligne1,borne_inf1,etendue1)
27   if (rang == 0) print *, "type_demi_ligne1: borne_inf=",borne_inf1,", etendue=",etendue1
28
29   ! Construction du type derive type_demi_ligne2
30   borne_inf2 = 0
31   etendue2 = taille_integer
32   call MPI_TYPE_CREATE_RESIZED(type_demi_ligne1,borne_inf2,etendue2,&
33                                type_demi_ligne2)
```

Types de données dérivés

```
34    ! Informations sur le type derive type_demi_ligne2
35    call MPI_TYPE_GET_EXTENT(type_demi_ligne2,borne_inf2,etendue2)
36    if (rang == 0) print *, "type_demi_ligne2: borne_inf=",borne_inf2,", etendue=",etendue2
37
38    ! Validation du type type_demi_ligne2
39    call MPI_TYPE_COMMIT(type_demi_ligne2)
40
41    if (rang == 0) then
42       ! Envoi de la matrice A au processus 1 avec le type type_demi_ligne2
43       call MPI_SEND(A(1,1),2,type_demi_ligne2,1,etiquette, &
44                     MPI_COMM_WORLD)
45    else
46       ! Reception pour le processus 1 dans la matrice A
47       call MPI_RECV(A(1,nb_colonnes-1),6,MPI_INTEGER,0,etiquette, &
48                     MPI_COMM_WORLD,statut)
49       print *,'Matrice A sur le processus 1'
50       do i=1,nb_lignes
51          print *,A(i,:)
52       end do
53    end if
54
55    call MPI_FINALIZE()
56 end program demi_ligne
```

```
> mpiexec -n 2 demi_ligne
 type_demi_ligne1: borne_inf=0, etendue=44
 type_demi_ligne2: borne_inf=0, etendue=4
```

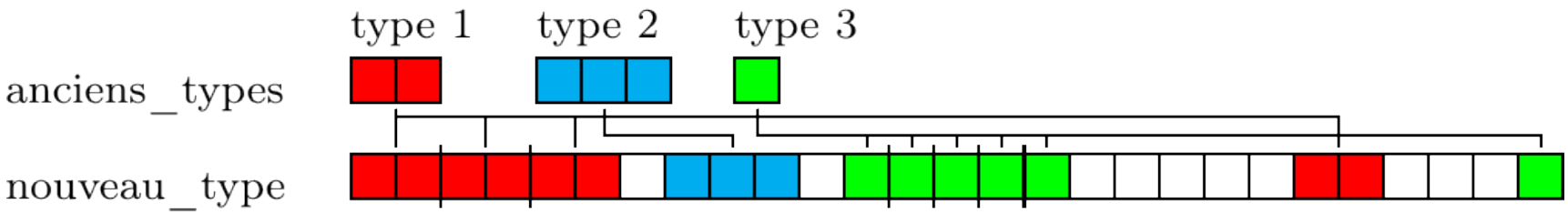
```
Matrice A sur le processus 1
 -1  -6 -11 -16  1  12
 -2  -7 -12 -17  6 -27
 -3  -8 -13 -18 11 -28
 -4  -9 -14 -19  2 -29
 -5 -10 -15 -20  7 -30
```

Types de données dérivés

Types hétérogènes

- Le sous-programme `MPI_Type_create_struct()` permet de créer une séquence de blocs de données en précisant le type, le nombre d'éléments et le pas de chaque bloc.
- Il s'agit du constructeur de types le plus complet. Il généralise `MPI_Type_indexed()` en permettant de définir un type différent pour chaque bloc.

nb=5, longueurs_blocs=(3,1,5,1,1), déplacements=(0,7,11,21,26),
anciens_types=(type1,type2,type3,type1,type3)




```
MPI_TYPE_CREATE_STRUCT(nb,longueurs_blocs,deplacements,
                       anciens_types,nouveau_type,code)

integer,intent(in)                                          :: nb
integer,intent(in), dimension(nb)                           :: longueurs_blocs
integer(kind=MPI_ADDRESS_KIND), intent(in), dimension(nb)   :: deplacements
TYPE(MPI_Datatype), intent(in), dimension(nb)               :: anciens_types
TYPE(MPI_Datatype), intent(out)                             :: nouveau_type
integer, optional, intent(out)                              :: code
```

Types de données dérivés

Calcul des déplacements

- `MPI_Type_create_struct()` est utile notamment pour créer des types MPI correspondant à des types dérivés Fortran ou à des structures C.
- L'alignement en mémoire des structures de données hétérogènes dépend de l'architecture et du compilateur.
- Attention, il faut corriger l'étendue du type MPI obtenu.
- Le sous-programme `MPI_Get_address()` permet de récupérer l'adresse d'une variable. C'est l'équivalent de l'opérateur de référencement (&) du C.
- Attention, même en C, il est préférable d'utiliser ce sous-programme MPI pour des raisons de portabilité.
- Il est conseillé d'utiliser `MPI_Aint_add()` et `MPI_Aint_diff()` pour faire des additions et soustractions sur des addresses.

```
MPI_GET_ADDRESS(variable,adresse_variable,code)

TYPE(*), dimension(..), asynchronous         :: variable
integer(kind=MPI_ADDRESS_KIND), intent(out) :: adresse_variable
integer, optional, intent(out)               :: code

integer(KIND=MPI_ADDRESS_KIND) MPI_AINT_ADD(adrr1, addr2)
integer(KIND=MPI_ADDRESS_KIND) MPI_AINT_DIFF(addr1, addr2)

integer(KIND=MPI_ADDRESS_KIND)               :: addr1, addr2
```

Types de données dérivés

```
1  program Interaction_Particules
2
3    use mpi_f08
4    implicit none
5
6    integer, parameter                              :: n=1000,etiquette=100
7    TYPE(MPI_Status)                                :: statut
8    integer                                         :: rang,i
9    TYPE(MPI_Datatype)                              :: type_particule,temp
10   TYPE(MPI_Datatype),dimension(4)                 :: types
11   integer, dimension(4)                           :: longueurs_blocs
12   integer(kind=MPI_ADDRESS_KIND), dimension(5)    :: deplacements,adresses
13   integer(kind=MPI_ADDRESS_KIND)                  :: lb,extent
14
15   type Particule
16      character(len=5)                             :: categorie
17      integer                                      :: masse
18      real, dimension(3)                           :: coords
19      logical                                      :: classe
20   end type Particule
21   type(Particule), dimension(n)                   :: p,temp_p
22
23   call MPI_INIT()
24   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
25
26   ! Construction du type de donnees
27   types = (/MPI_CHARACTER,MPI_INTEGER,MPI_REAL,MPI_LOGICAL/)
28   longueurs_blocs = (/5,1,3,1/)
```

Types de données dérivés

```
29    call MPI_GET_ADDRESS(p(1),adresses(1))
30    call MPI_GET_ADDRESS(p(1)%categorie,adresses(2))
31    call MPI_GET_ADDRESS(p(1)%masse,adresses(3))
32    call MPI_GET_ADDRESS(p(1)%coords,adresses(4))
33    call MPI_GET_ADDRESS(p(1)%classe,adresses(5))
34    ! Calcul des deplacements relatifs a l'adresse de depart
35    do i=1,4
36       deplacements(i)=MPI_AINT_DIFF(adresses(i+1),adresses(1))
37    end do
38    call MPI_TYPE_CREATE_STRUCT(4,longueurs_blocs,deplacements,types,temp)
39    call MPI_GET_ADDRESS(p(2),adresses(2))
40    lb = 0
41    extent = MPI_AINT_DIFF(adresses(2),adresses(1))
42    call MPI_TYPE_CREATE_RESIZED(temp,lb,extent,type_particule)
43    ! Validation du type structure
44    call MPI_TYPE_COMMIT(type_particule)
45
46    ! Initialisation des particules pour chaque processus
47
48    ! Envoi des particules de 0 vers 1
49    if (rang == 0) then
50       call MPI_SEND(p(1),n,type_particule,1,etiquette,MPI_COMM_WORLD)
51    else
52       call MPI_RECV(temp_p(1),n,type_particule,0,etiquette,MPI_COMM_WORLD, &
53                     statut)
54    endif
55
56    ! Liberation du type
57    call MPI_TYPE_FREE(type_particule)
58    call MPI_FINALIZE()
59 end program Interaction_Particules
```

Types de données dérivés

Conclusion

- Les types dérivés MPI sont de puissants mécanismes portables de description de données.
- Ils permettent, lorsqu'ils sont associés à des instructions comme `MPI_Sendrecv()`, de simplifier l'écriture de sous-programmes d'échanges interprocessus.
- L'association des types dérivés et des topologies (décrites dans l'un des prochains chapitres) fait de MPI l'outil idéal pour tous les problèmes de décomposition de domaine avec des maillages réguliers ou irréguliers.

Types de données dérivés

Memento

Sous-routines	longueurs_blocs	pas	types_anciens
MPI_Type_Contiguous()	constant*	constant*	constant
MPI_Type_[Create_H]Vector()	constant	constant	constant
MPI_Type_[Create_H]Indexed()	*variable*	*variable*	constant
MPI_Type_Create_Struct()	*variable*	*variable*	*variable*

(*) paramètre caché, égal à 1

Travaux pratiques MPI – Exercice 4 : Transposée d'une matrice

- Dans cet exercice, on se propose de se familiariser avec les types dérivés
- On se donne une matrice A de 5 lignes et 4 colonnes sur le processus 0
- Il s'agit pour le processus 0 d'envoyer au processus 1 cette matrice mais d'en faire automatiquement la transposition au cours de l'envoi

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

Processus 0

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

Processus 1

- Pour ce faire, on va devoir se construire deux types dérivés, un type type_ligne et un type type_transpose

Travaux pratiques MPI – Exercice 5 : Produit réparti de matrices

- Communications collectives et réductions : produit de matrices $C = A \times B$
  - On se limite au cas de matrices carrées dont l'ordre est un multiple du nombre de processus
  - Les matrices A et B sont sur le processus 0. Celui-ci distribue une tranche horizontale de la matrice A et une tranche verticale de la matrice B à chacun des processus. Chacun calcule alors un bloc diagonal de la matrice résultante C.
  - Pour calculer les blocs non diagonaux, chaque processus doit envoyer aux autres processus la tranche de A qu'il possède
  - Après quoi le processus 0 peut collecter les résultats et vérifier les résultats

Travaux pratiques MPI – Exercice 5 : Produit réparti de matrices

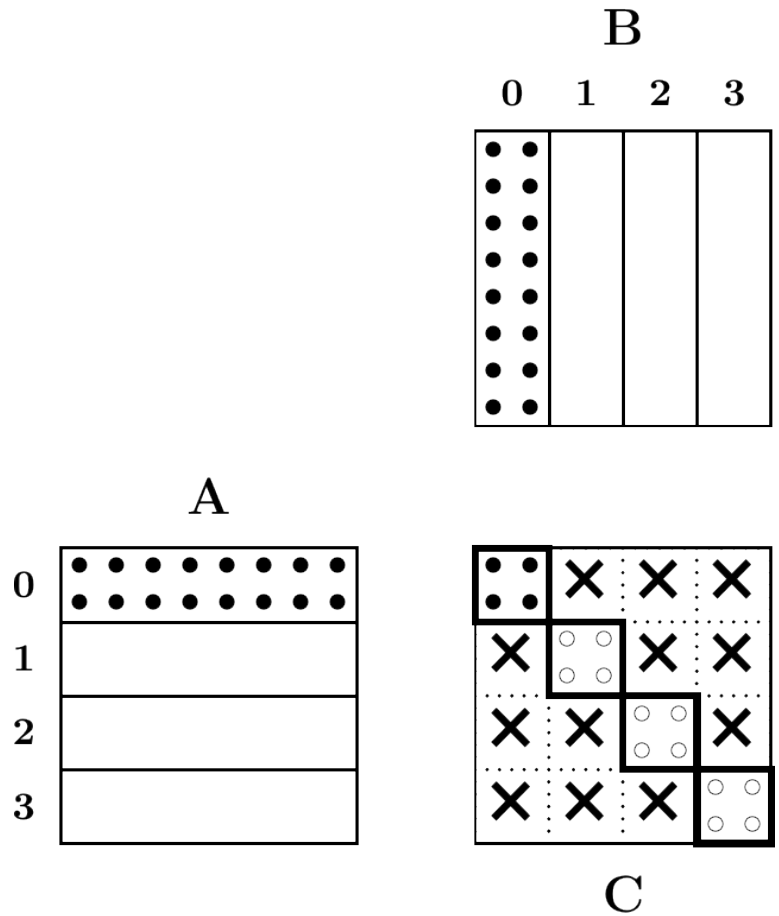


Figure 26 – Produit parallèle de matrices

Travaux pratiques MPI – Exercice 5 : Produit réparti de matrices

- Toutefois, l'algorithme qui peut sembler le plus immédiat, et qui est le plus simple à programmer, consistant à faire envoyer par chaque processus sa tranche de la matrice A à chacun des autres, n'est pas performant parce que le schéma de communication n'est pas du tout équilibré. C'est très facile à voir en faisant des mesures de performances et en représentant graphiquement les traces collectées.

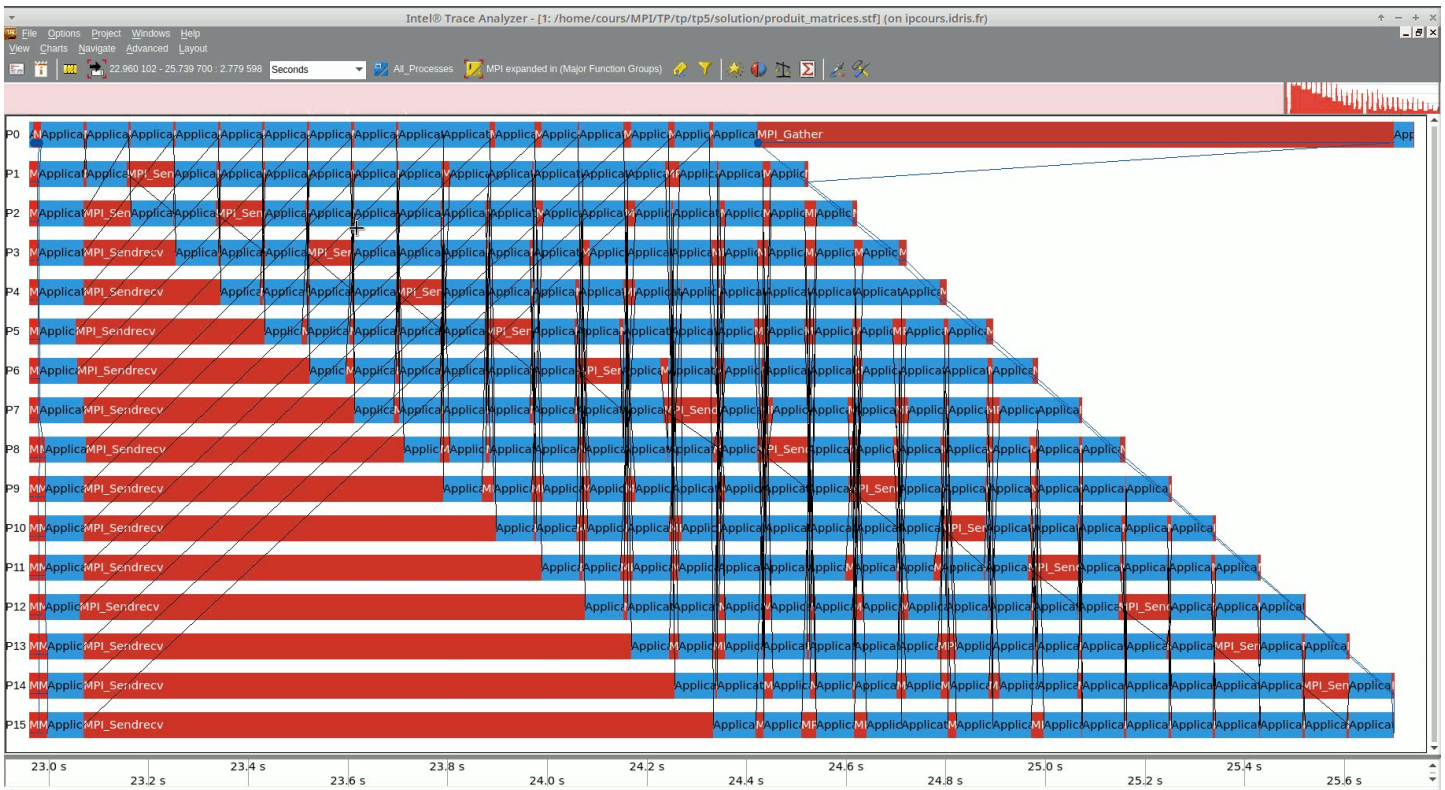


Figure 27 – Produit parallèle de matrices sur 16 processus, pour une taille de matrice de 1024 (premier algorithme)

Travaux pratiques MPI – Exercice 5 : Produit réparti de matrices

- Mais en changeant l'algorithme pour faire *glisser* le contenu des tranches de processus à processus, on peut obtenir un équilibre parfait des calculs et des communications, et gagner ainsi un facteur 2.

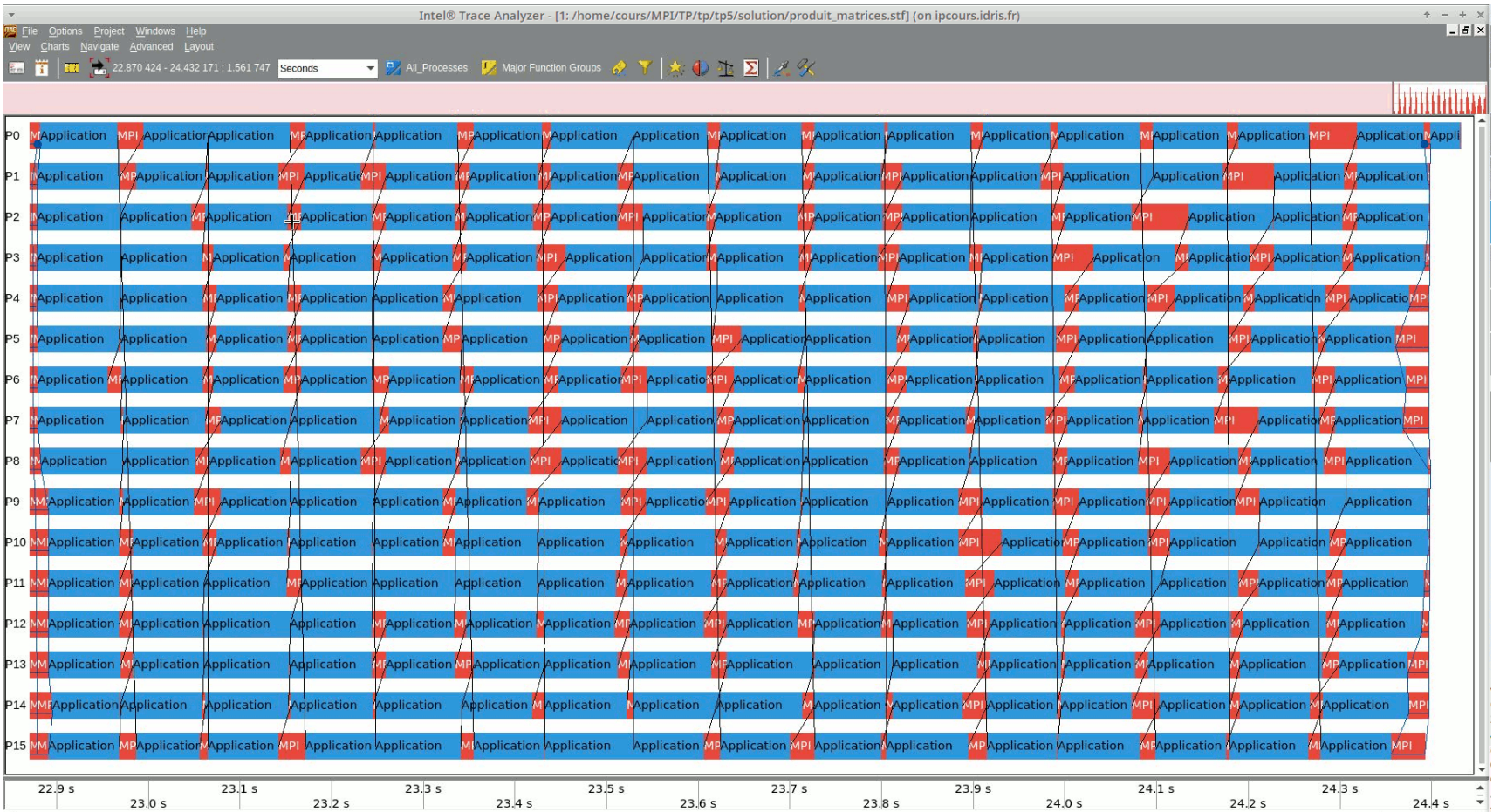


Figure 28 – Produit parallèle de matrices sur 16 processus, pour une taille de matrice de 1024 (second algorithme)

Communicateurs

Communicateurs

Introduction

Il s'agit de créer des sous-ensembles de processus sur lesquels on peut effectuer des opérations telles que des communications point à point, collectives, etc. Chaque sous-ensemble aura son propre espace de communication.

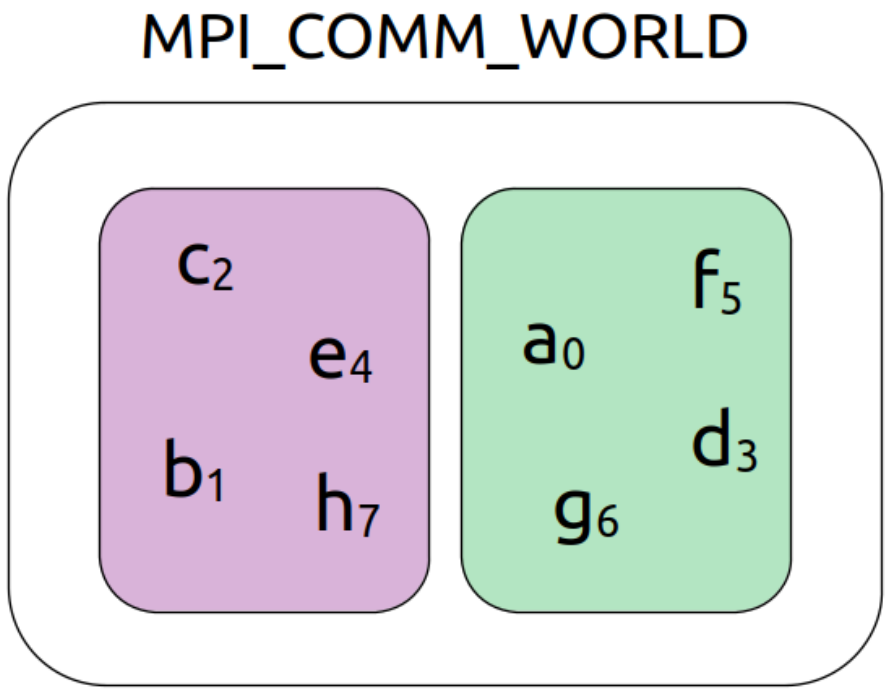


Figure 29 – Partitionnement d'un communicateur

Communicateurs

Exemple

Par exemple, on veut diffuser un message collectif aux processus de rang pair et un autre aux processus de rang impair.

- Boucler sur des *send/recv* peut être très pénalisant surtout si le nombre de processus est élevé. De plus un test serait obligatoire dans la boucle pour savoir si le rang du processus auquel le processus émetteur doit envoyer le message est pair ou impair.
- Une solution est de créer un communicateur regroupant les processus pairs et un autre regroupant les processus impairs, puis d'initier les communications collectives à l'intérieur de ces groupes.

Communicateurs

Communicateur par défaut

- On ne peut créer un communicateur qu'à partir d'un autre communicateur. Le premier sera créé à partir de `MPI_COMM_WORLD`.
- En effet, suite à l'appel à `MPI_Init()`, un communicateur est créé pour toute la durée d'exécution du programme.
- Son identificateur `MPI_COMM_WORLD` est une variable définie dans les fichiers d'en-tête.
- Il ne peut être détruit que via l'appel à `MPI_Finalize()`
- Par défaut, il fixe donc la portée des communications point à point et collectives à tous les processus de l'application

Communicateurs

Groupes et communicateurs

- Un communicateur est constitué :
  - d'un groupe, qui est un ensemble ordonné de processus ;
  - d'un contexte de communication mis en place à l'appel du sous-programme de construction du communicateur, qui permet de délimiter l'espace de communication.
- Les contextes de communication sont gérés par MPI (le programmeur n'a aucune action sur eux) : c'est un attribut « caché »
- Dans la bibliothèque MPI, divers sous-programmes existent pour construire des communicateurs : `MPI_Comm_create()`, `MPI_Comm_dup()`, `MPI_Comm_split()`
- Les constructeurs de communicateurs sont des opérateurs collectifs (qui engendrent des communications entre les processus)
- Les communicateurs que le programmeur crée peuvent être gérés dynamiquement et, de même qu'il est possible d'en créer, il est possible d'en détruire en utilisant le sous-programme `MPI_Comm_free()`

Communicateurs

Partitionnement d'un communicateur

Pour résoudre le problème de l'exemple, nous allons :

- partitionner le communicateur en processus de rang pair et d'autre part en processus de rang impair ;
- ne diffuser un message collectif qu'aux processus de rang pair et un autre qu'aux processus de rang impair.

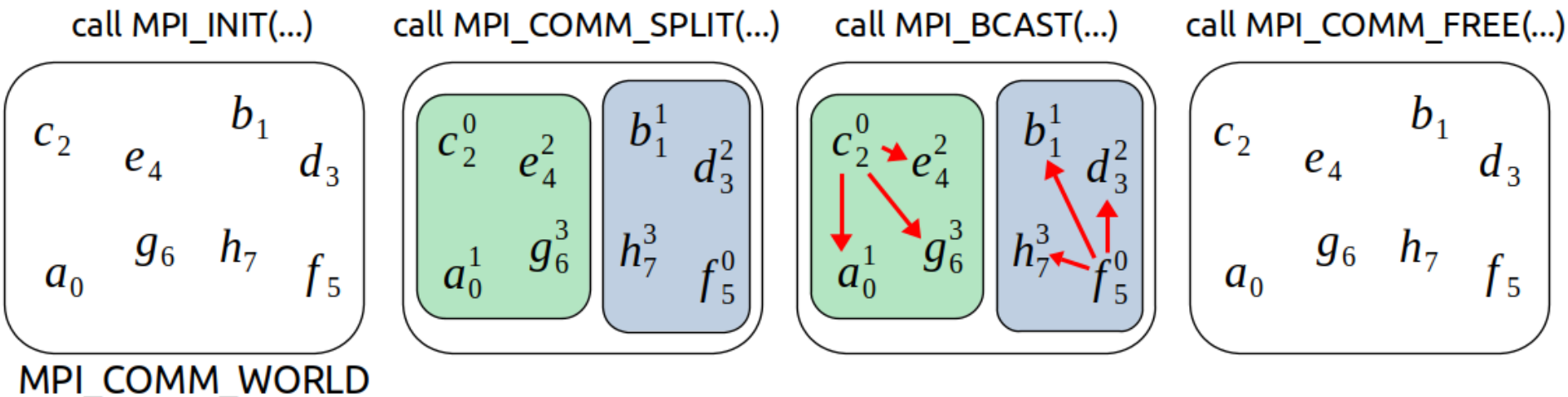


Figure 30 – Création/destruction d'un communicateur

Communicateurs

Partitionnement d'un communicateur avec `MPI_Comm_split()`

Le sous-programme `MPI_Comm_split()` permet de :

- partitionner un communicateur donné en autant de communicateurs que l'on veut
- donner le même nom à tous ces communicateurs : il aura la valeur du communicateur dans lequel se trouve le processus courant
- Méthode :
  1. définir une valeur couleur associant à chaque processus le numéro du communicateur auquel il appartiendra
  2. définir une valeur clef permettant de numéroter les processus dans chaque communicateur
  3. créer la partition où chaque communicateur s'appelle nouveau_comm

```
MPI_COMM_SPLIT(comm,couleur,clef,nouveau_comm,code)

TYPE(MPI_Comm), intent(in)      :: comm
integer, intent(in)             :: couleur, clef
TYPE(MPI_Comm), intent(out)     :: nouveau_comm
integer, optional, intent(out)  :: code
```

Un processus qui s'attribue une couleur égale à la valeur `MPI_UNDEFINED` aura pour nouveau_com le communicateur invalide `MPI_COMM_NULL`.

Communicateurs

Exemple

Voyons comment procéder pour construire le communicateur qui va subdiviser l'espace de communication entre processus de rangs pairs et impairs, via le constructeur `MPI_Comm_split()`.

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

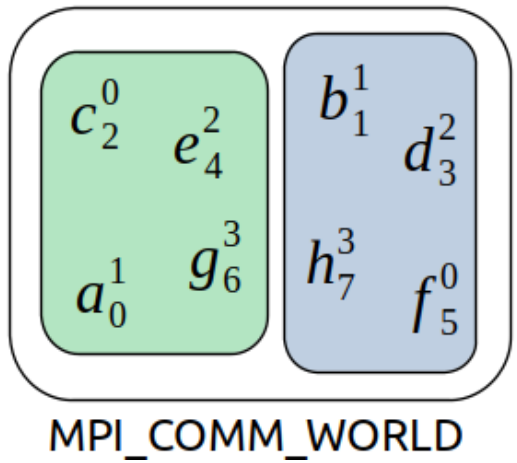
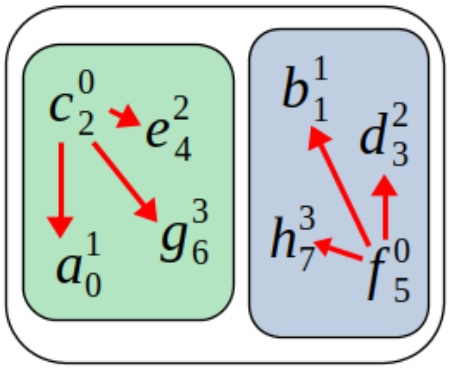


Figure 31 – Construction du communicateur `CommPairsImpairs` avec `MPI_Comm_split()`

Communicateurs

```
1  program PairsImpairs
2    use mpi_f08
3    implicit none
4
5    integer, parameter :: m=16
6    integer            :: clef
7    TYPE(MPI_Comm)     :: CommPairsImpairs
8    integer            :: rang_dans_monde
9    real, dimension(m) :: a
10
11   call MPI_INIT()
12   call MPI_COMM_RANK(MPI_COMM_WORLD,rang_dans_monde)
13
14   ! Initialisation du vecteur A
15   a(:)=0.
16   if(rang_dans_monde == 2) a(:)=2.
17   if(rang_dans_monde == 5) a(:)=5.
18
19   clef = rang_dans_monde
20   if (rang_dans_monde == 2 .OR. rang_dans_monde == 5 ) then
21      clef=-1
22   end if
23
24   ! Creation des communicateurs pair et impair en leur donnant une meme denomination
25   call MPI_COMM_SPLIT(MPI_COMM_WORLD,mod(rang_dans_monde,2),clef,CommPairsImpairs)
26
27   ! Diffusion du message par le processus 0 de chaque communicateur aux processus
28   ! de son groupe
29   call MPI_BCAST(a,m,MPI_REAL,0,CommPairsImpairs)
30
31   ! Destruction des communicateurs
32   call MPI_COMM_FREE(CommPairsImpairs)
33   call MPI_FINALIZE()
34 end program PairsImpairs
```



Communicateurs

Topologies

- Dans la plupart des applications, plus particulièrement dans les méthodes de décomposition de domaine où l'on fait correspondre le domaine de calcul à la grille de processus, il est intéressant de pouvoir disposer les processus suivant une topologie régulière
- MPI permet de définir des topologies virtuelles du type cartésien ou graphe
  - Topologies de type cartésien
    - chaque processus est défini dans une grille de processus ;
    - chaque processus a un voisin dans la grille ;
    - la grille peut être périodique ou non ;
    - les processus sont identifiés par leurs coordonnées dans la grille.
  - Topologies de type graphe
    - généralisation à des topologies plus complexes.

Figure 32 – Topologie cartésienne 2D (gauche) et topologie de type graphe (droite)

Communicateurs

Topologies cartésiennes

- Une topologie cartésienne est définie à partir d'un communicateur donné comm_ancien, en appelant le sous-programme `MPI_Cart_create()`.
- On définit :
  - un entier ndims représentant le nombre de dimensions de la grille
  - un tableau d'entiers dims de dimension ndims indiquant le nombre de processus dans chaque dimension
  - un tableau de logiques de dimension ndims indiquant la périodicité dans chaque dimension
  - un logique reorganisation indiquant si la numérotation des processus peut être changé par MPI

```
MPI_CART_CREATE(comm_ancien, ndims,dims,periods,reorganisation,comm_nouveau,code)

TYPE(MPI_Comm), intent(in)            :: comm_ancien
integer, intent(in)                   :: ndims
integer, dimension(ndims),intent(in)  :: dims
logical, dimension(ndims),intent(in)  :: periods
logical, intent(in)                   :: reorganisation
TYPE(MPI_Comm), intent(out)           :: comm_nouveau
integer, optional, intent(out)        :: code
```

Communicateurs

Exemple

Exemple sur une grille comportant 4 domaines suivant x et 2 suivant y, périodique en y.

```
use mpi_f08
TYPE(MPI_Comm)              :: comm_2D
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 = .false.

call MPI_CART_CREATE(MPI_COMM_WORLD,ndims,dims,periods,reorganisation,comm_2D)
```

Si `reorganisation = .false.` alors le rang des processus dans le nouveau communicateur (comm_2D) est le même que dans l'ancien communicateur (`MPI_COMM_WORLD`).
Si `reorganisation = .true.`, l'implémentation MPI choisit l'ordre des processus.

Communicateurs

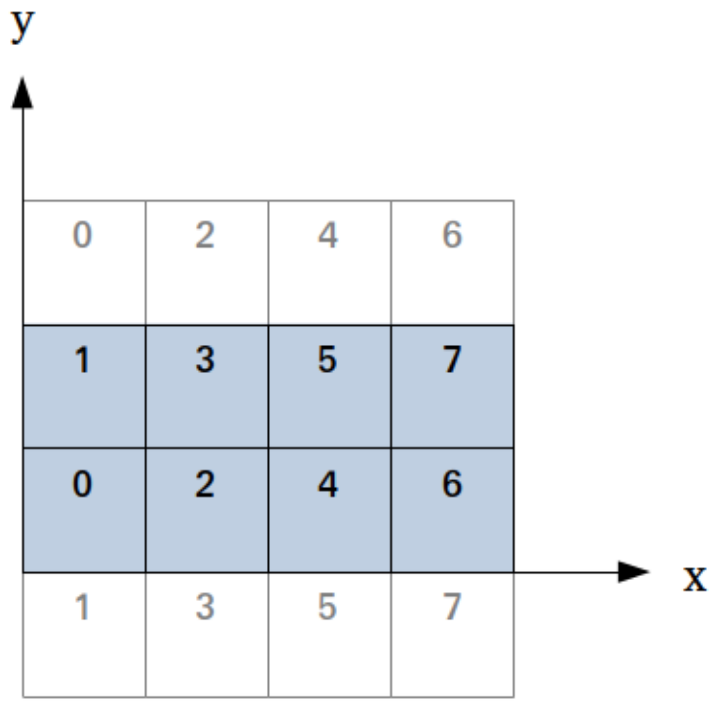


Figure 33 – Topologie cartésienne 2D périodique en y

Communicateurs

Exemple 3D

Exemple sur une grille 3D comportant 4 domaines suivant x, 2 suivant y et 2 suivant z, non périodique.

```
use mpi_f08
TYPE(MPI_Comm)              :: comm_3D
integer, parameter          :: ndims = 3
integer, dimension(ndims)   :: dims
logical, dimension(ndims)   :: periods
logical                     :: reorganisation

.............................................

dims(1) = 4
dims(2) = 2
dims(3) = 2
periods(:) = .false.
reorganisation = .false.

call MPI_CART_CREATE(MPI_COMM_WORLD,ndims,dims,periods,reorganisation,comm_3D)
```

Communicateurs

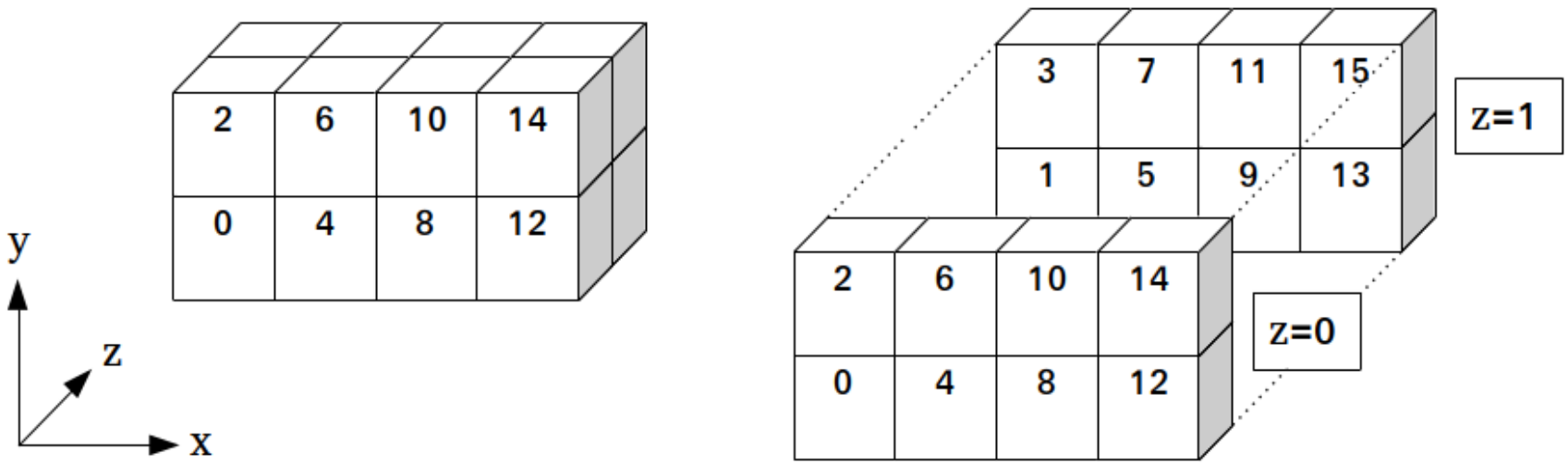


Figure 34 – Topologie cartésienne 3D non périodique

Communicateurs

Distribution des processus

Le sous-programme MPI_Dims_create() retourne le nombre de processus dans chaque dimension de la grille en fonction du nombre total de processus.

```
MPI_DIMS_CREATE(nb_procs,ndims,dims,code)

integer, intent(in)                      :: nb_procs, ndims
integer, dimension(ndims),intent(inout)  :: dims
integer, optional, intent(out)           :: code
```

Remarque : si les valeurs de dims en entrée valent toutes 0, cela signifie qu'on laisse à MPI le choix du nombre de processus dans chaque direction en fonction du nombre total de processus.

dims en entrée	MPI_Dims_create	dims en sortie
(0,0)	(8,2,dims,code)	(4,2)
(0,0,0)	(16,3,dims,code)	(4,2,2)
(0,4,0)	(16,3,dims,code)	(2,4,2)
(0,3,0)	(16,3,dims,code)	error

Rang et coordonnées d'un processus

Dans une topologie cartésienne, le rang de chaque processus est associé à ses coordonnées dans la grille.

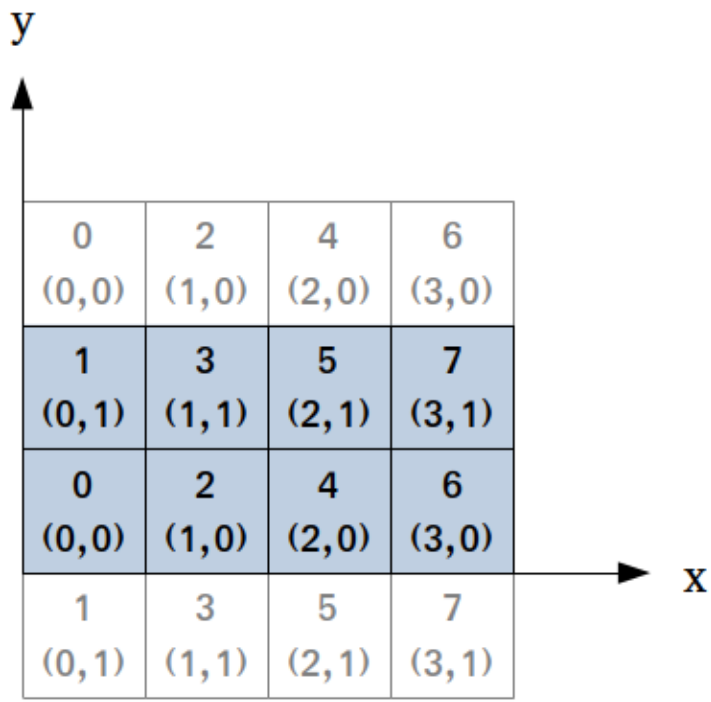


Figure 35 – Topologie cartésienne 2D périodique en y

Communicateurs

Rang d'un processus connaissant ses coordonnées

Dans une topologie cartésienne, le sous-programme `MPI_Cart_rank()` retourne le rang du processus associé aux coordonnées dans la grille.

```
MPI_CART_RANK(comm,coords,rang,code)

TYPE(MPI_Comm), intent(in)             :: comm
integer, dimension(ndims),intent(in)   :: coords
integer, intent(out)                   :: rang
integer, optional, intent(out)         :: code
```

Communicateurs

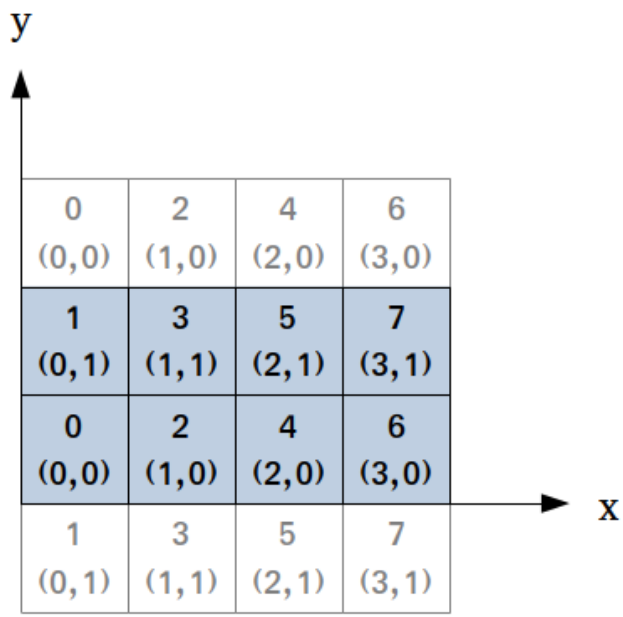


Figure 36 – Topologie cartésienne 2D périodique en y

```
coords(1)=dims(1)-1
do i=0,dims(2)-1
  coords(2) = i
  call MPI_CART_RANK(comm_2D,coords,rang(i))
end do
.........................................
i=0,en entree coords=(3,0),en sortie rang(0)=6.
i=1,en entree coords=(3,1),en sortie rang(1)=7.
```

Communicateurs

Coordonnées d'un processus connaissant son rang

Dans une topologie cartésienne, le sous-programme `MPI_Cart_coords()` retourne les coordonnées d'un processus de rang donné dans la grille.

```
MPI_CART_COORDS(comm, rang, ndims, coords, code)

TYPE(MPI_Comm), intent(in)              :: comm
integer, intent(in)                     :: rang, ndims
integer, dimension(ndims),intent(out)   :: coords
integer, optional, intent(out)          :: code
```

Communicateurs

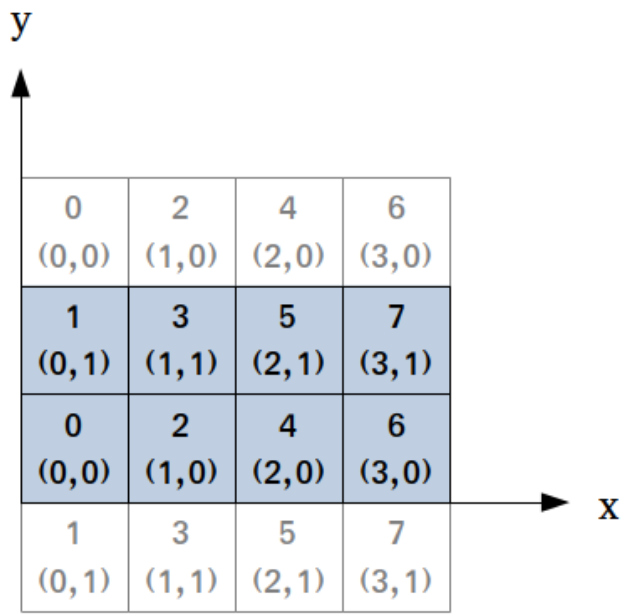


Figure 37 – Topologie cartésienne 2D périodique en y

```
if (mod(rang,2) == 0) then
  call MPI_CART_COORDS(comm_2D,rang,2,coords)
end if
.........................................
En entree, les valeurs de rang sont : 0,2,4,6.
En sortie, les valeurs de coords sont :
(0,0),(1,0),(2,0),(3,0).
```

Rang des voisins

Dans une topologie cartésienne, un processus appelant le sous-programme `MPI_Cart_Shift()` se voit retourner le rang de ses processus voisins dans une direction donnée.

```
MPI_CART_SHIFT(comm, direction, pas, rang_precedent, rang_suivant, code)

TYPE(MPI_Comm), intent(in)      :: comm
integer, intent(in)             :: direction, pas
integer, intent(out)            :: rang_precedent,rang_suivant
integer, optional, intent(out)  :: code
```

- Le paramètre direction correspond à l'axe du déplacement (xyz).
- Le paramètre pas correspond au pas du déplacement.
- Si un rang n'a pas de voisin précédent (resp. suivant) dans la direction demandée, alors la valeur du rang précédent (resp. suivant) sera `MPI_PROC_NULL`.

Communicateurs

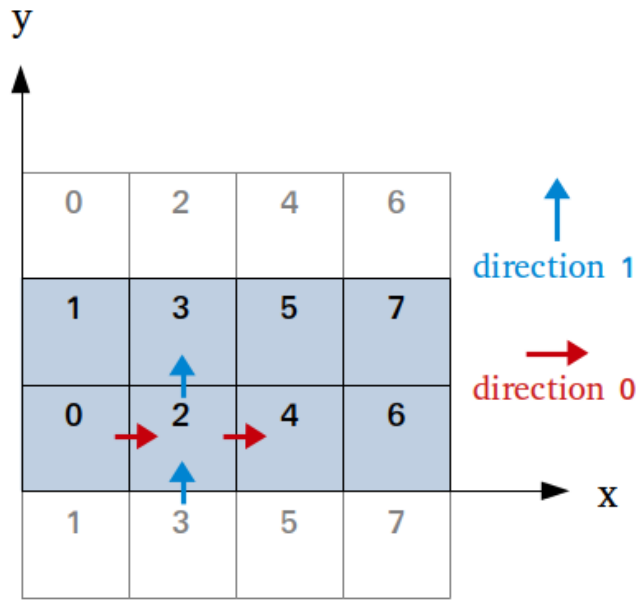


Figure 38 – Appel du sous-programme MPI_Cart_shift()

```
call MPI_CART_SHIFT(comm_2D,0,1,rang_gauche,rang_droit)
.....................................
Pour le processus 2, rang_gauche=0, rang_droit=4
```

```
call MPI_CART_SHIFT(comm_2D,1,1,rang_bas,rang_haut)
.....................................
Pour le processus 2, rang_bas=3, rang_haut=3
```

Communicateurs

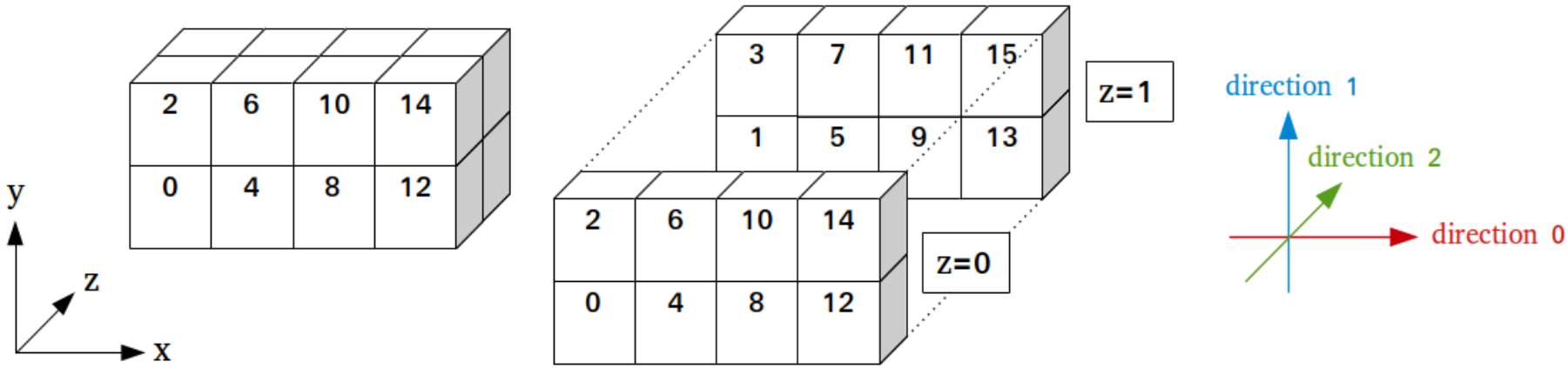


Figure 39 – Appel du sous-programme MPI_Cart_shift()

```
call MPI_CART_SHIFT(comm_3D,0,1,rang_gauche,rang_droit)
.....................................
Pour le processus 0, rang_gauche=-1, rang_droit=4
```

```
call MPI_CART_SHIFT(comm_3D,1,1,rang_bas,rang_haut)
.....................................
Pour le processus 0, rang_bas=-1, rang_haut=2
```

```
call MPI_CART_SHIFT(comm_3D,2,1,rang_avant,rang_arriere)
.....................................
Pour le processus 0, rang_avant=-1, rang_arriere=1
```

Communicateurs

Exemple

- création d'une grille cartésienne 2D périodique en y
- récupération des coordonnées de chaque processus
- récupération des rangs voisins pour chaque processus

```
1  program decomposition
2    use mpi_f08
3    implicit none
4
5    integer                    :: rang_ds_topo,nb_procs
6    TYPE(MPI_Comm)             :: comm_2D
7    integer, dimension(4)      :: voisin
8    integer, parameter         :: N=1,E=2,S=3,W=4
9    integer, parameter         :: ndims = 2
10   integer, dimension (ndims) :: dims,coords
11   logical, dimension (ndims) :: periods
12   logical                    :: reorganisation
13
14   call MPI_INIT()
15
16   call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs)
17
18   ! Connaitre le nombre de processus suivant x et y
19   dims(:) = 0
20
21   call MPI_DIMS_CREATE(nb_procs,ndims,dims)
```

Communicateurs

```
22    ! Creation grille 2D periodique en y
23    periods(1) = .false.
24    periods(2) = .true.
25    reorganisation = .false.
26
27    call MPI_CART_CREATE(MPI_COMM_WORLD,ndims,dims,periods,reorganisation,comm_2D)
28
29    ! Connaitre mes coordonnees dans la topologie
30    call MPI_COMM_RANK(comm_2D,rang_ds_topo)
31    call MPI_CART_COORDS(comm_2D,rang_ds_topo,ndims,coords)
32
33    ! Recherche de mes voisins Ouest et Est
34    call MPI_CART_SHIFT(comm_2D,0,1,voisin(W),voisin(E))
35
36    ! Recherche de mes voisins Sud et Nord
37    call MPI_CART_SHIFT(comm_2D,1,1,voisin(S),voisin(N))
38
39    call MPI_FINALIZE()
40
41  end program decomposition
```

Communicateurs

Subdiviser une topologie cartésienne

- La question est de savoir comment dégénérer une topologie cartésienne de processus 2D ou 3D en une topologie cartésienne respectivement 1D ou 2D.
- Pour MPI, dégénérer 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 cartésienne initiale.
- L'intérêt majeur est de pouvoir effectuer des opérations collectives restreintes à un sous-ensemble de processus appartenant à :
  - une même ligne (ou colonne), si la topologie initiale est 2D ;
  - un même plan, si la topologie initiale est 3D.

Communicateurs

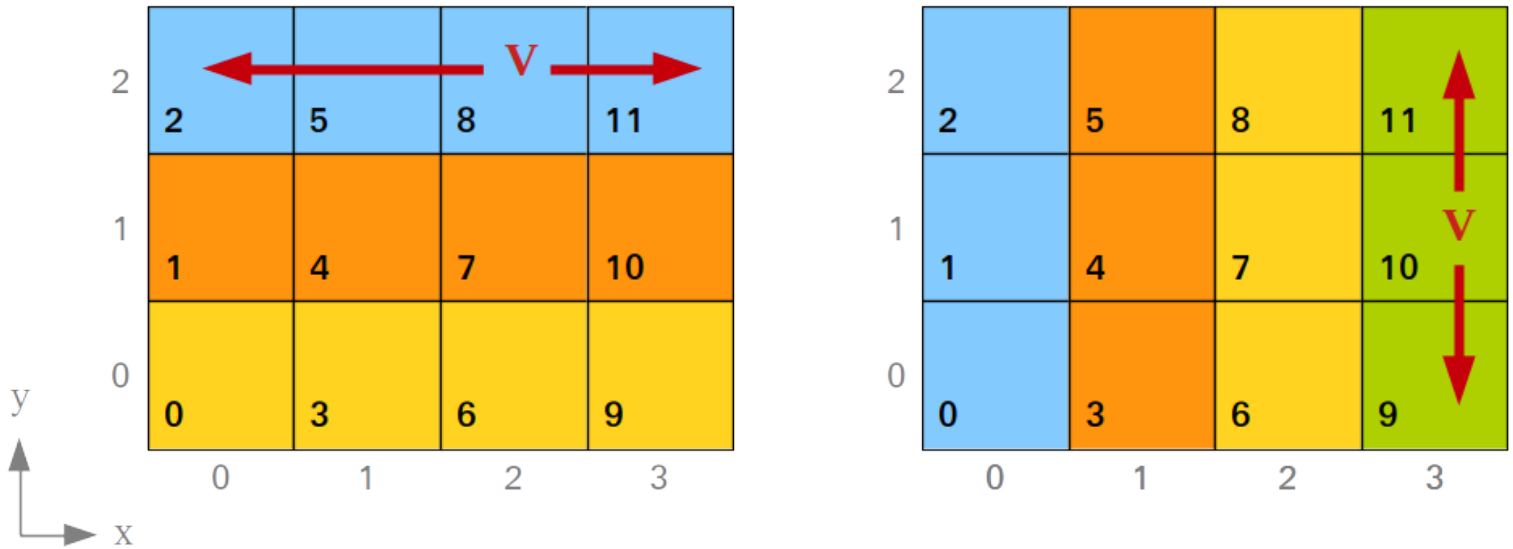


Figure 40 – Deux exemples de distribution de données dans une topologie 2D dégénérée

Communicateurs

Subdiviser une topologie cartésienne

Il existe deux façons de faire pour dégénérer une topologie :

- en utilisant le sous-programme général `MPI_Comm_split()` ;
- en utilisant le sous-programme `MPI_Cart_sub()` prévu à cet effet.

```
MPI_CART_SUB(CommCart,conserve_dims,CommCartD,code)

logical, intent(in), dimension(NDim) :: conserve_dims
TYPE(MPI_Comm), intent(in)           :: CommCart
TYPE(MPI_Comm), intent(out)          :: CommCartD
integer, optional, intent(out)       :: code
```

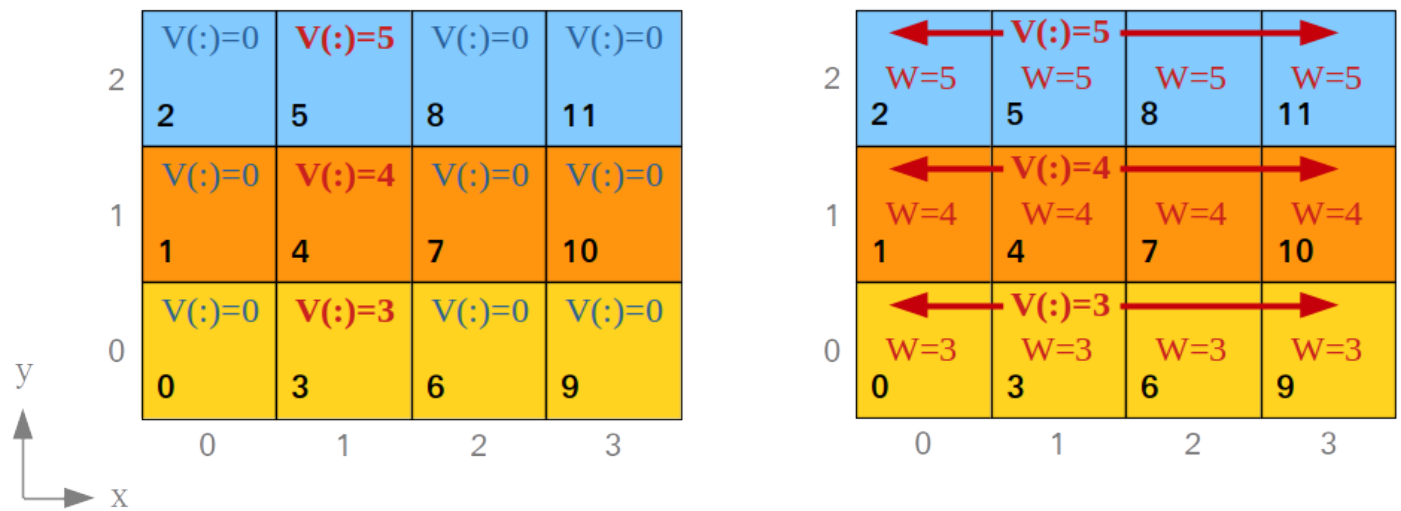


Figure 41 – Distribution d'un tableau *V* sur la grille 2D dégénérée

Communicateurs

```
1  program CommCartSub
2    use mpi_f08
3    implicit none
4
5    TYPE(MPI_Comm)              :: Comm2D,Comm1D
6    integer                     :: rang
7    integer,parameter           :: NDim2D=2
8    integer,dimension(NDim2D)   :: Dim2D,Coord2D
9    logical,dimension(NDim2D)   :: Periode,conserve_dims
10   logical                     :: Reordonne
11   integer,parameter           :: m=4
12   real, dimension(m)          :: V=0.
13   real                        :: W=0.
```

Communicateurs

```
14    call MPI_INIT()
15
16    ! Creation de la grille 2D initiale
17    Dim2D(1) = 4
18    Dim2D(2) = 3
19    Periode(:) = .false.
20    ReOrdonne = .false.
21    call MPI_CART_CREATE(MPI_COMM_WORLD,NDim2D,Dim2D,Periode,ReOrdonne,Comm2D)
22    call MPI_COMM_RANK(Comm2D,rang)
23    call MPI_CART_COORDS(Comm2D,rang,NDim2D,Coord2D)
24
25    ! Initialisation du vecteur V
26    if (Coord2D(1) == 1) V(:)=real(rang)
27
28    ! Chaque ligne de la grille doit etre une topologie cartesienne 1D
29    conserve_dims(1) = .true.
30    conserve_dims(2) = .false.
31    ! Subdivision de la grille cartesienne 2D
32    call MPI_CART_SUB(Comm2D,conserve_dims,Comm1D)
33
34    ! Les processus de la colonne 2 distribuent le vecteur V aux processus de leur ligne
35    call MPI_SCATTER(V,1,MPI_REAL,W,1,MPI_REAL,1,Comm1D)
36
37    print '("Rang : ",I2," ; Coordonnees : (",I1,",",I1,") ; W = ",F2.0)', &
38          rang,Coord2D(1),Coord2D(2),W
39
40    call MPI_FINALIZE()
41  end program CommCartSub
```

Communicateurs

```
> mpiexec -n 12 CommCartSub
Rang :  0 ; Coordonnees : (0,0) ; W = 3.
Rang :  1 ; Coordonnees : (0,1) ; W = 4.
Rang :  3 ; Coordonnees : (1,0) ; W = 3.
Rang :  8 ; Coordonnees : (2,2) ; W = 5.
Rang :  4 ; Coordonnees : (1,1) ; W = 4.
Rang :  5 ; Coordonnees : (1,2) ; W = 5.
Rang :  6 ; Coordonnees : (2,0) ; W = 3.
Rang : 10 ; Coordonnees : (3,1) ; W = 4.
Rang : 11 ; Coordonnees : (3,2) ; W = 5.
Rang :  9 ; Coordonnees : (3,0) ; W = 3.
Rang :  2 ; Coordonnees : (0,2) ; W = 5.
Rang :  7 ; Coordonnees : (2,1) ; W = 4.
```

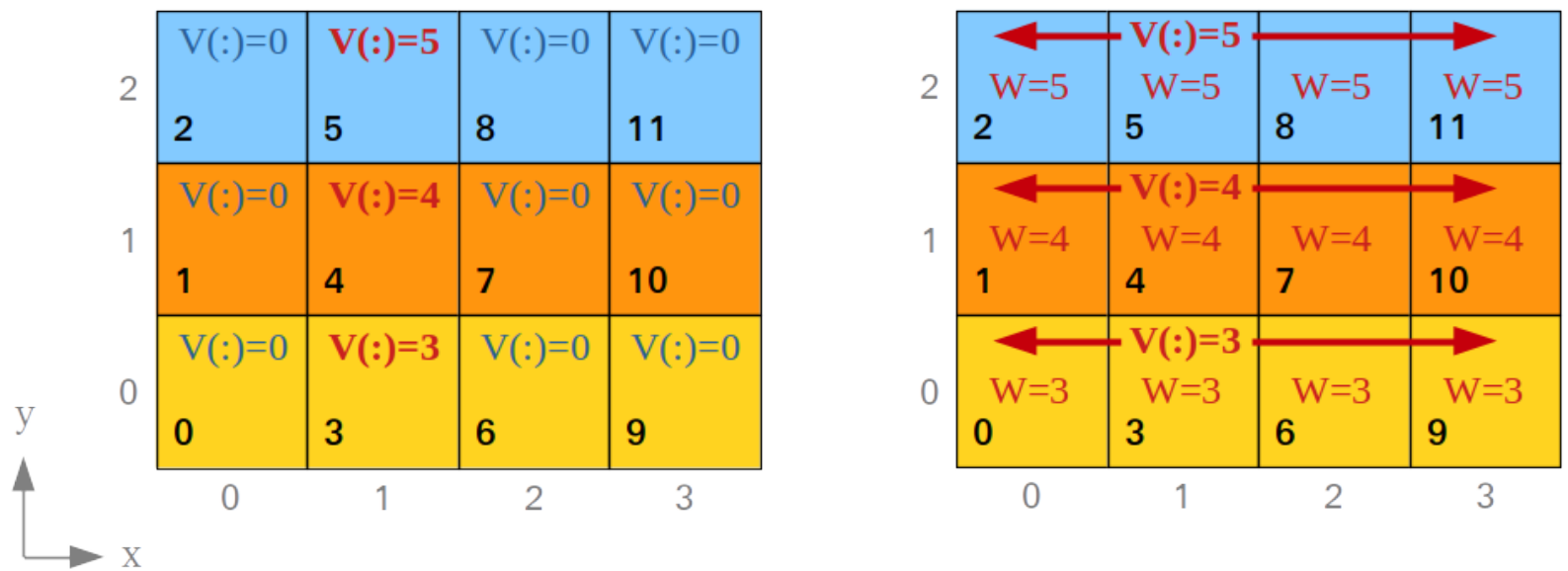


Figure 42 – Distribution d'un tableau *V* sur la grille 2D dégénérée

Travaux pratiques MPI – Exercice 6 : Communicateurs

- En partant de la topologie cartésienne définie ci-dessous, subdiviser en 2 communicateurs suivant les lignes via `MPI_Comm_split()`

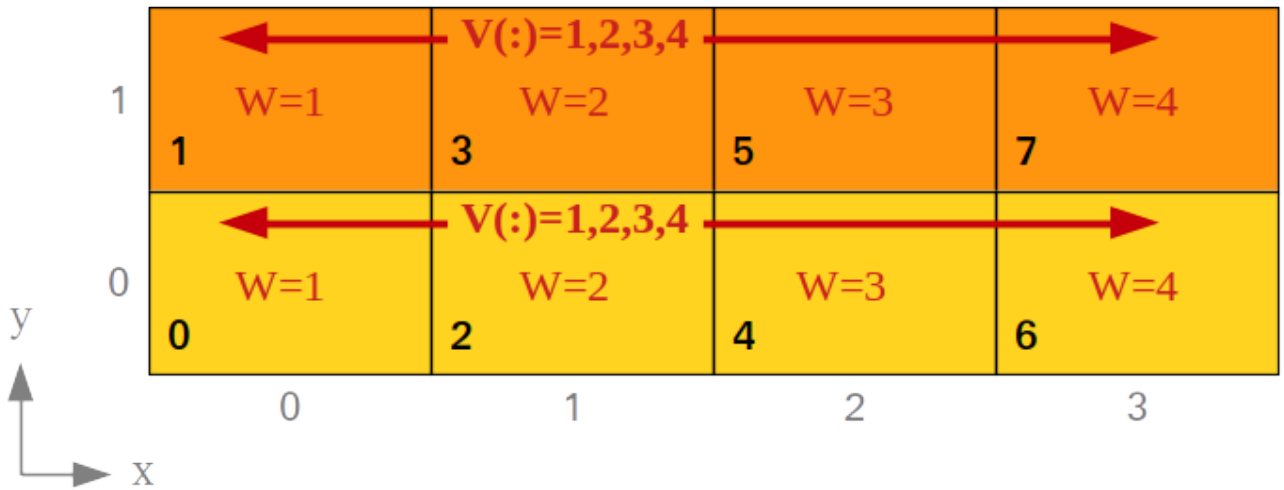


Figure 43 – Subdivision d'une topologie 2D et communication suivant la topologie 1D obtenue

- Contrainte : définir les couleurs de chaque processus sans utiliser l'opération *modulo*.

MPI-IO

Optimisation des entrées-sorties

- Très logiquement, les applications qui font des calculs volumineux manipulent également des quantités importantes de données externes, et génèrent donc un nombre conséquent d'entrées-sorties.
- Le traitement efficace de celles-ci influe donc parfois très fortement sur les performances globales des applications.
- L'optimisation des entrées-sorties de codes parallèles se fait par la combinaison :
  - de leur parallélisation, pour éviter de créer un goulet d'étranglement en raison de leur sérialisation ;
  - de techniques mises en œuvre explicitement au niveau de la programmation (lectures / écritures non-bloquantes) ;
  - d'opérations spécifiques prises en charge par le système d'exploitation (regroupement des requêtes, gestion des tampons d'entrées-sorties, etc.).
- L'utilisation d'une bibliothèque facilite l'optimisation des entrées-sorties.

L'interface MPI-IO

- La norme MPI-2 définie un ensemble de fonctions permettant de réaliser des entrées-sorties parallèles.
- L'interface est calquée sur celle utilisée pour l'échange de messages MPI. Par exemple, les opérations collectives et non-bloquantes sur les fichiers sont gérées de façon similaire à ce que propose MPI pour les messages entre processus. La définition des données accédées suivant les processus se fait par l'utilisation de types de données (de base ou bien dérivés).
- Bien sûr, de nombreux éléments (descripteurs de fichiers, attributs . . .) rappellent les interfaces d'entrées-sorties natives des langages de programmation.

MPI-IO

Exemple d'optimisation séquentielle implémentée par les bibliothèques

- Pour obtenir de bonnes performances, il est préférable de limiter le nombre de requêtes (latence) et de lire de larges blocs de données.
- Lorsqu'un seul processus accède à de nombreux petits blocs discontinus, il est possible de regrouper les requêtes pour plus de performances.
- Une bibliothèque MPI-IO peut implémenter cette optimisation de manière transparente.

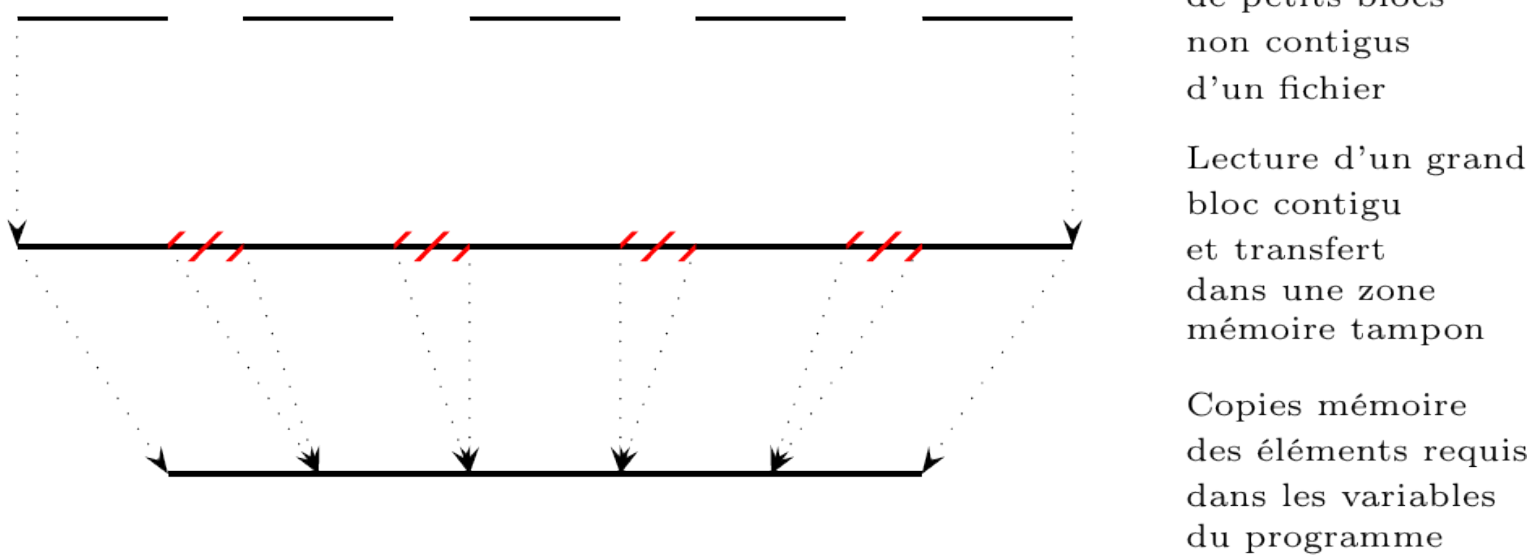


Figure 44 – Mécanisme de *passoire* (*data sieving*) dans le cas d'accès nombreux, par un seul processus, à de petits blocs discontinus

Exemple d'optimisation parallèle

Lorsqu'un ensemble de processus accède à des blocs discontinus (cas des tableaux distribués, par exemple), la bibliothèque d'I/O peut optimiser l'opération en maximisant l'accès aux données contiguës et en utilisant des communications collectives de redistribution.

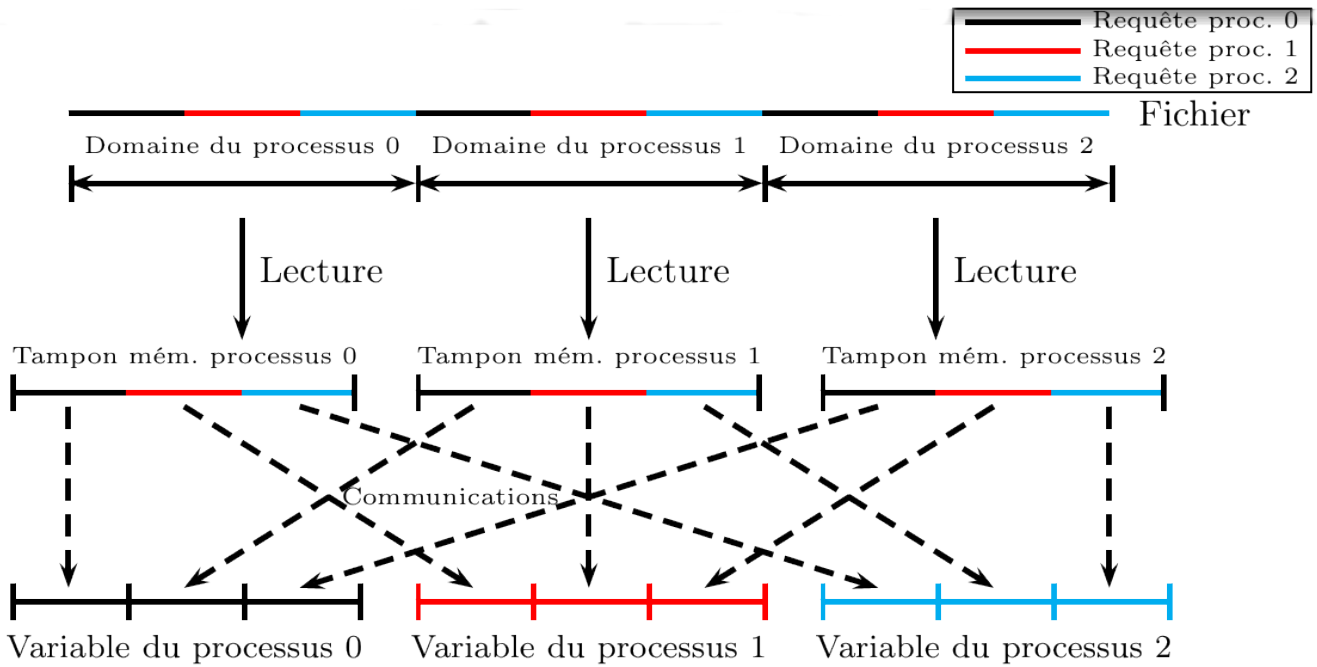


Figure 45 – Lecture en deux phases, par un ensemble de processus

Ouverture et fermeture d'un fichier

- L'ouverture et la fermeture d'un fichier sont des opérations *collectives*.
- Tous les processus du communicateur au sein duquel un fichier est ouvert participeront aux opérations collectives ultérieures d'accès aux données.
- L'ouverture d'un fichier renvoie un descripteur. C'est un objet opaque qui est ensuite utilisé comme référence dans toutes les opérations portant sur le fichier.
- A l'ouverture, les attributs décrivent les droits d'accès, le mode d'ouverture, la destruction éventuelle à la fermeture, etc. Les attributs doivent être précisés en utilisant des constantes prédéfinies et peuvent être combinés entre eux.
- Nous ne décrivons ici que les sous-programmes d'ouverture et de fermeture mais d'autres sous-programme de gestion de fichiers sont disponibles (obtention des caractéristiques, suppression, pré-allocation, etc.). Par exemple, le sous-programme `MPI_File_get_info()` permet d'obtenir des informations sur un fichier ouvert (les informations disponibles varient d'une implémentation à l'autre).

MPI-IO

```
1  program open01
2    use mpi_f08
3    implicit none
4    character(len=MPI_MAX_ERROR_STRING) :: texte_erreur
5    TYPE(MPI_File)                      :: descripteur
6    integer                             :: code, texte_longueur
7
8    call MPI_INIT()
9
10   call MPI_FILE_OPEN(MPI_COMM_WORLD,"fichier.txt", &
11                      MPI_MODE_RDWR + MPI_MODE_CREATE,MPI_INFO_NULL,descripteur,code)
12   IF (code /= MPI_SUCCESS) THEN
13     CALL MPI_ERROR_STRING(code,texte_erreur,texte_longueur)
14     PRINT *, texte_erreur(1:texte_longueur)
15     CALL MPI_ABORT(MPI_COMM_WORLD, 42)
16   END IF
17
18   call MPI_FILE_CLOSE(descripteur,code)
19   IF (code /= MPI_SUCCESS) THEN
20     PRINT *, 'Erreur fermeture fichier'
21     CALL MPI_ABORT(MPI_COMM_WORLD, 2)
22   END IF
23   call MPI_FINALIZE()
24
25 end program open01
```

```
> ls -l fichier.txt

-rw-------   1 nom       grp    0 Feb 08 12:13 fichier.txt
```

MPI-IO

Attribut	Signification
MPI_MODE_RDONLY	seulement en lecture
MPI_MODE_RDWR	en lecture et écriture
MPI_MODE_WRONLY	seulement en écriture
MPI_MODE_CREATE	création du fichier s'il n'existe pas
MPI_MODE_EXCL	erreur si le fichier existe
MPI_MODE_UNIQUE_OPEN	le fichier n'est pas ouvert ailleurs
MPI_MODE_SEQUENTIAL	accès séquentiel
MPI_MODE_APPEND	pointeurs en fin de fichier (mode ajout)
MPI_MODE_DELETE_ON_CLOSE	destruction après la fermeture

Gestion des erreurs

- Le comportement concernant l'argument code est différent pour la partie IO de MPI ;
- Il est nécessaire de tester la valeur de cet argument ;
- Il est possible de changer ce comportement avec `MPI_File_set_errhandler()` ;
- Deux gestionnaires d'erreurs sont disponibles : `MPI_ERRORS_ARE_FATAL` et `MPI_ERRORS_RETURN` ;
- `MPI_Comm_set_errhandler()` permet de changer la gestion des erreurs pour les communications.

```
MPI_FILE_SET_ERRHANDLER(descripteur,gestionnaire,code)

TYPE(MPI_File), intent(inout)     :: descripteur
TYPE(MPI_Errhandler), intent(in) :: gestionnaire
integer, optional, intent(out)    :: code
```

Pour changer le comportement par défaut, il faut utiliser `MPI_FILE_NULL` comme descripteur.

Généralités

- Les transferts de données entre fichiers et zones mémoire des processus se font via des appels explicites à des sous-programmes de lecture et d'écriture.
- On distingue trois propriétés des accès aux fichiers :
  - le positionnement, qui peut être explicite (en spécifiant un déplacement par rapport au début du fichier) ou implicite, via des pointeurs gérés par le système (ces pointeurs peuvent être de deux types : soit individuels à chaque processus, soit partagés par tous les processus) ;
  - la synchronisation, les accès pouvant être de type bloquants ou non bloquants ;
  - le regroupement, les accès pouvant être collectifs (c'est-à-dire effectués par tous les processus du communicateur au sein duquel le fichier a été ouvert) ou propres seulement à un ou plusieurs processus.
- Il est possible de mélanger les types d'accès effectués à un même fichier au sein d'une application.

MPI-IO

Positionnement	Synchronisation	individuel	collectif
adresses explicites	bloquantes	MPI_File_read_at MPI_File_write_at	MPI_File_read_at_all MPI_File_write_at_all
	non bloquantes	MPI_File_iread_at MPI_File_iwrite_at	MPI_File_iread_at_all MPI_File_iwrite_at_all
pointeurs implicites individuels	bloquantes	MPI_File_read MPI_File_write	MPI_File_read_all MPI_File_write_all
	non bloquantes	MPI_File_iread MPI_File_iwrite	MPI_File_iread_all MPI_File_iwrite_all
pointeurs implicites partagés	bloquantes	MPI_File_read_shared MPI_File_write_shared	MPI_File_read_ordered MPI_File_write_ordered
	non bloquantes	MPI_File_iread_shared MPI_File_iwrite_shared	MPI_File_read_ordered_begin MPI_File_read_ordered_end MPI_File_write_ordered_begin MPI_File_write_ordered_end

Le mécanisme de vue

- Par défaut, les fichiers sont lus comme une simple suite d'octets mais MPI-IO dispose d'un mécanisme permettant une abstraction de plus haut niveau du contenu des fichiers : il est possible de décrire des structures de données complexes et de s'en servir comme gabarit lors de l'accès aux fichiers.
- Pour l'instant, il faut seulement savoir qu'un type élémentaire de données sert d'unité de base à ces constructions et que, par défaut, le type élémentaire est l'octet.
- Ce mécanisme de vue sera décrit en détail plus tard.

Déplacements explicites

- Le déplacement est explicite lorsque la position à partir de laquelle on travaille sur le fichier est définie explicitement lors de l'opération de lecture ou d'écriture.
- La position dans un fichier s'exprime toujours comme un multiple du type élémentaire de la vue courante. Par défaut, la position s'exprime donc en octets.
- La taille de la zone de lecture/écriture est exprimée en nombre d'occurences d'un type de donnée (ex : `MPI_INTEGER`). La taille du type doit être un multiple du type élémentaire.

MPI-IO

```
1  program write_at
2    use mpi_f08
3    implicit none
4    integer, parameter                  :: nb_valeurs=10
5    integer                             :: i,rang,code,nb_octets_entier
6    TYPE(MPI_File)                      :: descripteur
7    integer(kind=MPI_OFFSET_KIND)       :: position_fichier
8    integer, dimension(nb_valeurs)      :: valeurs
9    TYPE(MPI_Status)                    :: statut
10
11   call MPI_INIT()
12   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
13   valeurs(:)= (/(i+rang*100,i=1,nb_valeurs)/)
14   print *, "Ecriture processus",rang,":",valeurs(:)
15
16   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_WRONLY + MPI_MODE_CREATE, &
17                      MPI_INFO_NULL,descripteur,code)
18   IF (code /= MPI_SUCCESS) THEN
19     PRINT *, 'Erreur ouverture fichier'
20     CALL MPI_ABORT(MPI_COMM_WORLD, 42)
21   END IF
22   call MPI_TYPE_SIZE(MPI_INTEGER,nb_octets_entier)
23   position_fichier=rang*nb_valeurs*nb_octets_entier
24
25   call MPI_FILE_SET_ERRHANDLER(descripteur,MPI_ERRORS_ARE_FATAL)
26   call MPI_FILE_WRITE_AT(descripteur,position_fichier,valeurs,nb_valeurs,MPI_INTEGER, &
27                          statut)
28
29   call MPI_FILE_CLOSE(descripteur)
30   call MPI_FINALIZE()
31 end program write_at
```

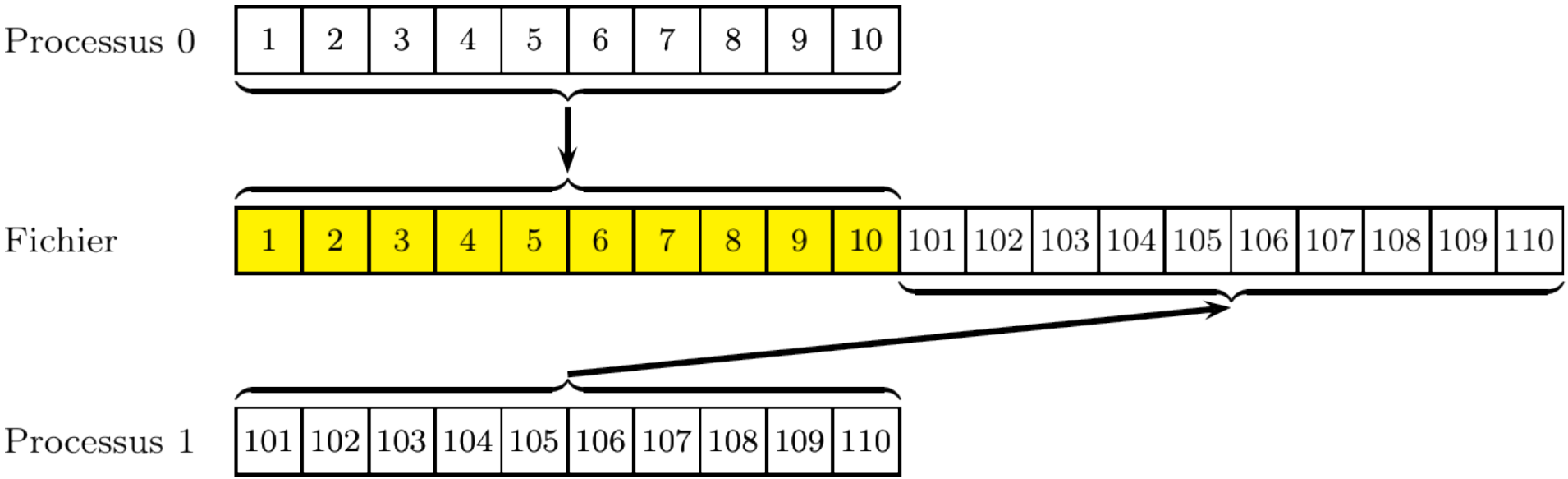


Figure 46 – Exemple d'utilisation de `MPI_File_write_at()`

```
> mpiexec -n 2 write_at

Ecriture processus 0 :    1,    2,    3,    4,    5,    6,    7,    8,    9,   10
Ecriture processus 1 :  101,  102,  103,  104,  105,  106,  107,  108,  109,  110
```

MPI-IO

```
1  program read_at
2
3    use mpi_f08
4    implicit none
5
6    integer, parameter                  :: nb_valeurs=10
7    integer                             :: rang,code,nb_octets_entier
8    TYPE(MPI_File)                      :: descripteur
9    integer(kind=MPI_OFFSET_KIND)       :: position_fichier
10   integer, dimension(nb_valeurs)      :: valeurs
11   TYPE(MPI_Status)                    :: statut
12
13   call MPI_INIT()
14   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
15   call MPI_FILE_SET_ERRHANDLER(MPI_FILE_NULL,MPI_ERRORS_ARE_FATAL);
16   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
17                      descripteur,code)
18
19   call MPI_TYPE_SIZE(MPI_INTEGER,nb_octets_entier)
20
21   position_fichier=rang*nb_valeurs*nb_octets_entier
22   call MPI_FILE_READ_AT(descripteur,position_fichier,valeurs,nb_valeurs,MPI_INTEGER, &
23                         statut)
24   print *, "Lecture processus",rang,":",valeurs(:)
25
26   call MPI_FILE_CLOSE(descripteur)
27   call MPI_FINALIZE()
28
29 end program read_at
```

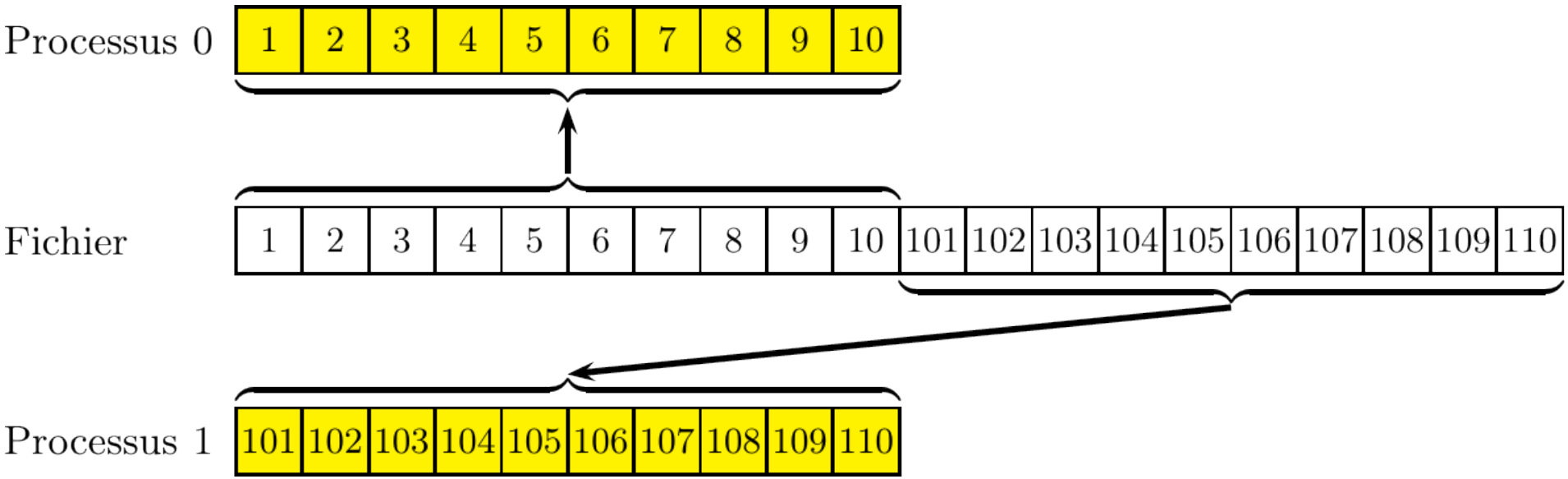


Figure 47 – Exemple d'utilisation de `MPI_File_read_at()`

```
> mpiexec -n 2 read_at

Lecture processus 0 :   1,   2,   3,   4,   5,   6,   7,   8,   9,  10
Lecture processus 1 : 101, 102, 103, 104, 105, 106, 107, 108, 109, 110
```

Déplacements implicites individuels

- Un pointeur individuel est géré par le système, et ceci par fichier et par processus.
- Pour un processus donné, deux accès successifs au même fichier permettent donc d'accéder automatiquement aux éléments consécutifs de celui-ci.
- Dans tous ces sous-programmes, les pointeurs partagés ne sont jamais accédés ou modifiés explicitement.
- Après chaque accès, le pointeur est positionné sur l'élément suivant.

MPI-IO

```
1  program read01
2
3    use mpi_f08
4    implicit none
5
6    integer, parameter                  :: nb_valeurs=10
7    integer                             :: rang,code
8    TYPE(MPI_File)                      :: descripteur
9    integer, dimension(nb_valeurs)      :: valeurs
10   TYPE(MPI_Status)                    :: statut
11
12   call MPI_INIT()
13   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
14
15   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
16                      descripteur,code)
17
18   call MPI_FILE_READ(descripteur,valeurs,6,MPI_INTEGER,statut)
19   call MPI_FILE_READ(descripteur,valeurs(7),4,MPI_INTEGER,statut)
20
21   print *, "Lecture processus",rang,":",valeurs(:)
22
23   call MPI_FILE_CLOSE(descripteur)
24   call MPI_FINALIZE()
25
26 end program read01
```

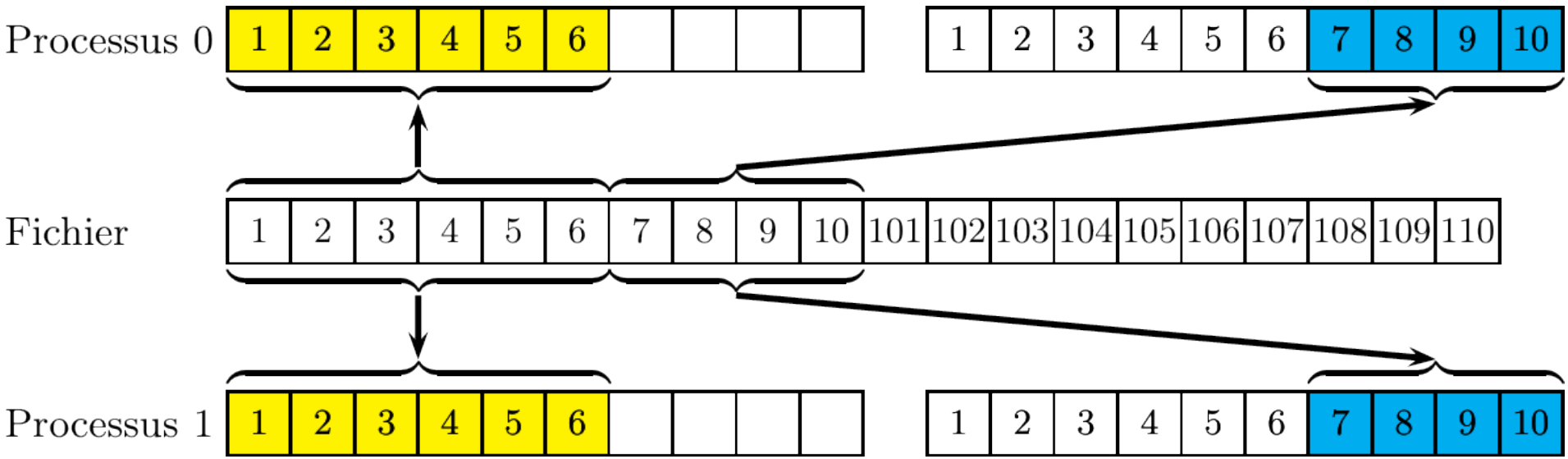


Figure 48 – Exemple 1 d'utilisation de `MPI_File_read()`

```
> mpiexec -n 2 read01

Lecture processus 1 :  1,  2,  3,  4,  5,  6,  7,  8,  9,  10
Lecture processus 0 :  1,  2,  3,  4,  5,  6,  7,  8,  9,  10
```

MPI-IO

```
1  program read02
2    use mpi_F08
3    implicit none
4
5    integer, parameter                  :: nb_valeurs=10
6    integer                             :: rang,code
7    TYPE(MPI_File)                      :: descripteur
8    integer, dimension(nb_valeurs)      :: valeurs=0
9    TYPE(MPI_Status)                    :: statut
10
11   call MPI_INIT()
12   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
13
14   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
15                      descripteur,code)
16
17   if (rang == 0) then
18      call MPI_FILE_READ(descripteur,valeurs,5,MPI_INTEGER,statut)
19   else
20      call MPI_FILE_READ(descripteur,valeurs,8,MPI_INTEGER,statut)
21      call MPI_FILE_READ(descripteur,valeurs,5,MPI_INTEGER,statut)
22   end if
23
24   print *, "Lecture processus",rang,":",valeurs(1:8)
25
26   call MPI_FILE_CLOSE(descripteur)
27   call MPI_FINALIZE()
28 end program read02
```

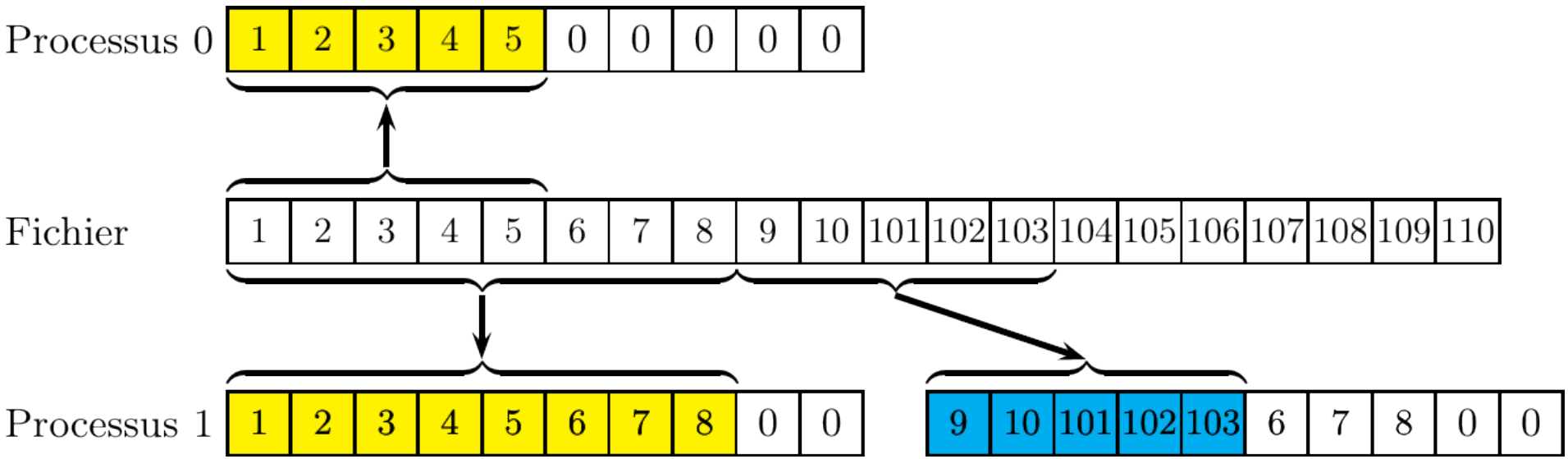


Figure 49 – Exemple 2 d'utilisation de `MPI_File_read()`

```
> mpiexec -n 2 read02

Lecture processus 0 :  1,   2,   3,   4,   5,   0,   0,   0
Lecture processus 1 :  9,  10, 101, 102, 103,   6,   7,   8
```

Déplacements implicites partagés

- Il existe un et un seul pointeur partagé par fichier, commun à tous les processus du communicateur dans lequel le fichier a été ouvert.
- Tous les processus qui font une opération d'entrée-sortie utilisant le pointeur partagé doivent employer la même vue du fichier.
- Si on utilise les variantes non collectives des sous-programmes, l'ordre de lecture n'est pas déterministe. Si le traitement doit être déterministe, il faut explicitement gérer l'ordonnancement des processus ou utiliser les variantes collectives.
- Après chaque accès, le pointeur est positionné sur l'élément suivant.
- Dans tous ces sous-programmes, les pointeurs individuels ne sont jamais accédés ou modifiés.

MPI-IO

```
1  program read_shared01
2
3    use mpi_f08
4    implicit none
5
6    integer                              :: rang,code
7    TYPE(MPI_File)                       :: descripteur
8    integer, parameter                   :: nb_valeurs=10
9    integer, dimension(nb_valeurs)       :: valeurs
10   TYPE(MPI_Status)                     :: statut
11
12   call MPI_INIT()
13   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
14
15   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
16                      descripteur,code)
17
18   call MPI_FILE_READ_SHARED(descripteur,valeurs,4,MPI_INTEGER,statut)
19   call MPI_FILE_READ_SHARED(descripteur,valeurs(5),6,MPI_INTEGER,statut)
20
21   print *, "Lecture processus",rang,":",valeurs(:)
22
23   call MPI_FILE_CLOSE(descripteur)
24   call MPI_FINALIZE()
25
26 end program read_shared01
```

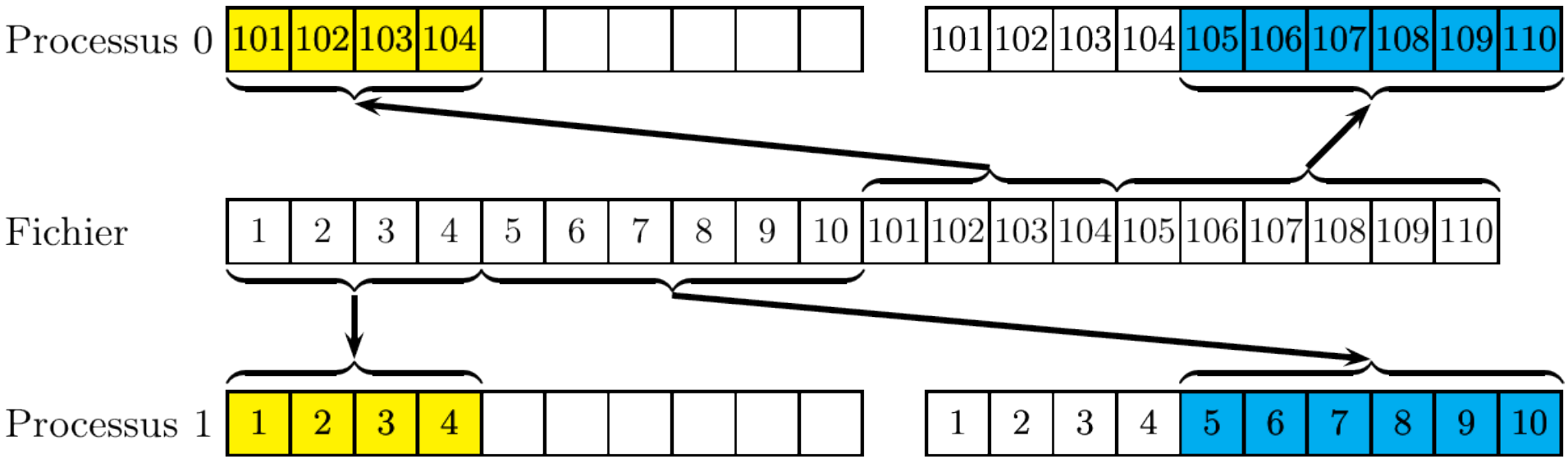


Figure 50 – Exemple 2 d'utilisation de `MPI_File_read_shared()`

```
> mpiexec -n 2 read_shared01

Lecture processus  1 :   1,   2,   3,   4,   5,   6,   7,   8,   9,  10
Lecture processus  0 : 101, 102, 103, 104, 105, 106, 107, 108, 109, 110
```

Lectures/écritures collectives

- Tous les processus du communicateur au sein duquel un fichier est ouvert participent aux opérations collectives d'accès aux données.
- Les opérations collectives sont généralement plus performantes que les opérations individuelles, parce qu'elles autorisent davantage de techniques d'optimisation mises en œuvre automatiquement ;
- Les accès sont effectués dans l'ordre des rangs des processus : le traitement est donc ici déterministe.

MPI-IO

```
1  program read_at_all
2    use mpi_f08
3    implicit none
4
5    integer, parameter                  :: nb_valeurs=10
6    integer                             :: rang,code,nb_octets_entier
7    TYPE(MPI_File)                      :: descripteur
8    integer(kind=MPI_OFFSET_KIND)       :: position_fichier
9    integer, dimension(nb_valeurs)      :: valeurs
10   TYPE(MPI_Status)                    :: statut
11
12   call MPI_INIT()
13   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
14
15   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
16                      descripteur)
17
18   call MPI_TYPE_SIZE(MPI_INTEGER,nb_octets_entier)
19   position_fichier=rang*nb_valeurs*nb_octets_entier
20   call MPI_FILE_READ_AT_ALL(descripteur,position_fichier,valeurs,nb_valeurs, &
21                             MPI_INTEGER,statut)
22   print *, "Lecture processus",rang,":",valeurs(:)
23
24   call MPI_FILE_CLOSE(descripteur)
25   call MPI_FINALIZE()
26 end program read_at_all
```

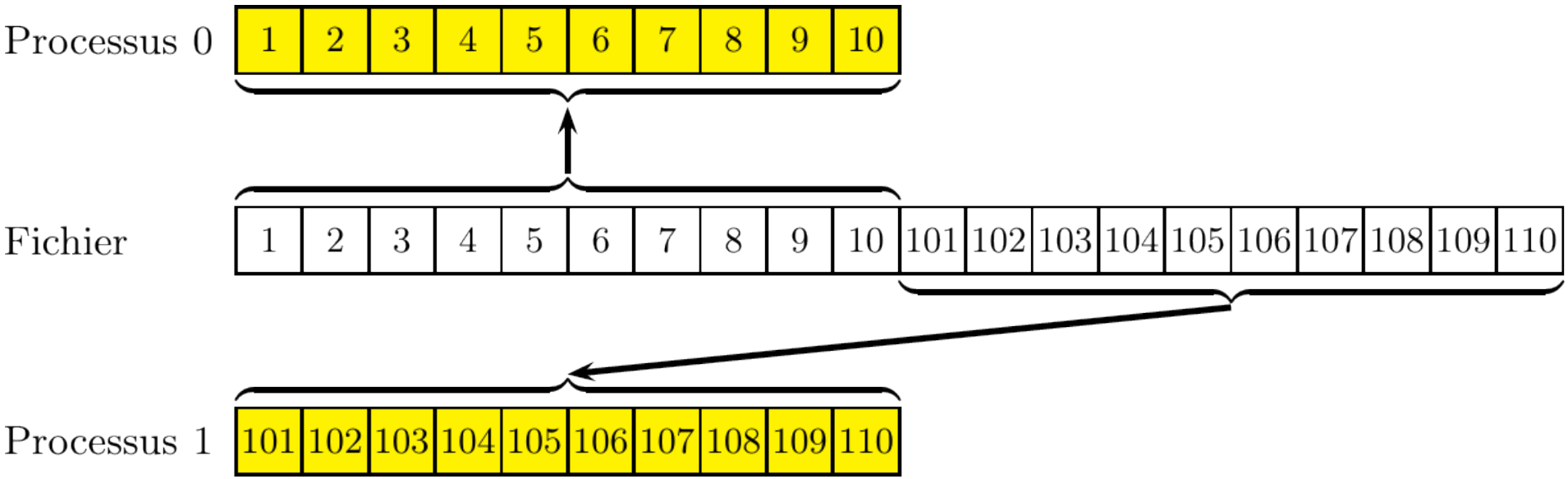


Figure 51 – Exemple d'utilisation de `MPI_File_read_at_all()`

```
> mpiexec -n 2 read_at_all

Lecture processus 0 :   1,   2,   3,   4,   5,   6,   7,   8,   9,  10
Lecture processus 1 : 101, 102, 103, 104, 105, 106, 107, 108, 109, 110
```

MPI-IO

```
1  program read_all01
2    use mpi_f08
3    implicit none
4
5    integer                              :: rang,code
6    TYPE(MPI_File)                       :: descripteur
7    integer, parameter                   :: nb_valeurs=10
8    integer, dimension(nb_valeurs)       :: valeurs
9    TYPE(MPI_Status)                     :: statut
10
11   call MPI_INIT()
12   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
13
14   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
15                      descripteur)
16
17   call MPI_FILE_READ_ALL(descripteur,valeurs,4,MPI_INTEGER,statut)
18   call MPI_FILE_READ_ALL(descripteur,valeurs(5),6,MPI_INTEGER,statut)
19
20   print *, "Lecture processus ",rang,":",valeurs(:)
21
22   call MPI_FILE_CLOSE(descripteur)
23   call MPI_FINALIZE()
24 end program read_all01
```

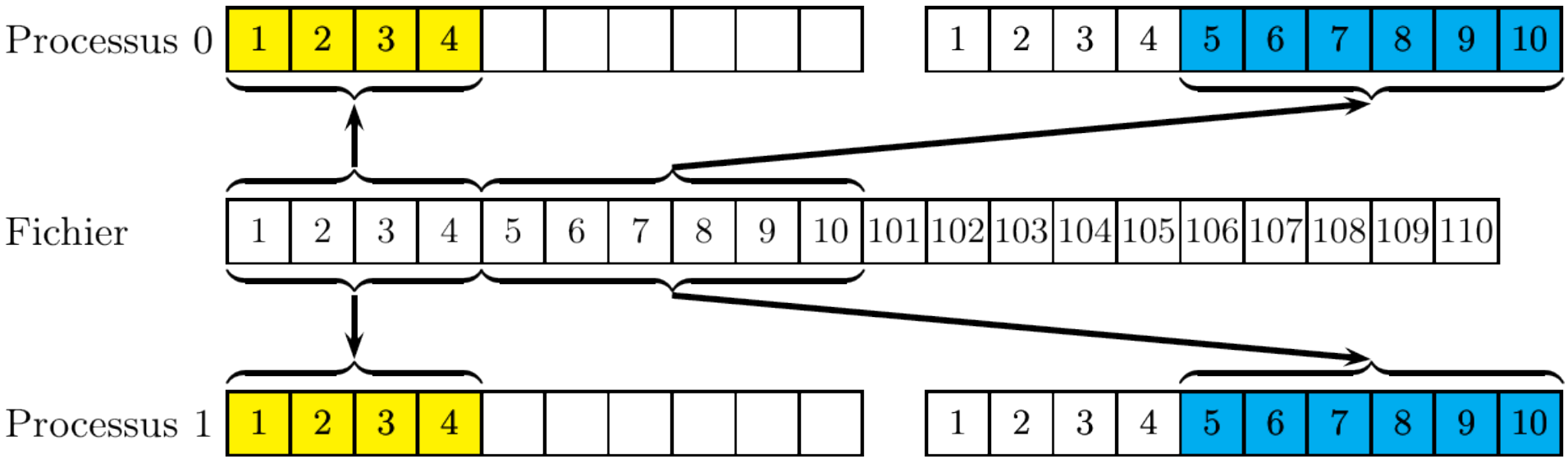


Figure 52 – Exemple 1 d'utilisation de `MPI_File_read_all()`

```
> mpiexec -n 2 read_all01

Lecture processus 0 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Lecture processus 1 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

MPI-IO

```
1  program read_all02
2    use mpi_f08
3    implicit none
4
5    integer, parameter                  :: nb_valeurs=10
6    integer                             :: rang,indice1,indice2,code
7    TYPE(MPI_File)                      :: descripteur
8    integer, dimension(nb_valeurs)      :: valeurs=0
9    TYPE(MPI_Status)                    :: statut
10
11   call MPI_INIT()
12   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
13   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
14                      descripteur)
15
16   if (rang == 0) then
17      indice1=3
18      indice2=6
19   else
20      indice1=5
21      indice2=9
22   end if
23
24   call MPI_FILE_READ_ALL(descripteur,valeurs(indice1),indice2-indice1+1, &
25                          MPI_INTEGER,statut)
26   print *, "Lecture processus",rang,":",valeurs(:)
27
28   call MPI_FILE_CLOSE(descripteur)
29   call MPI_FINALIZE()
30 end program read_all02
```

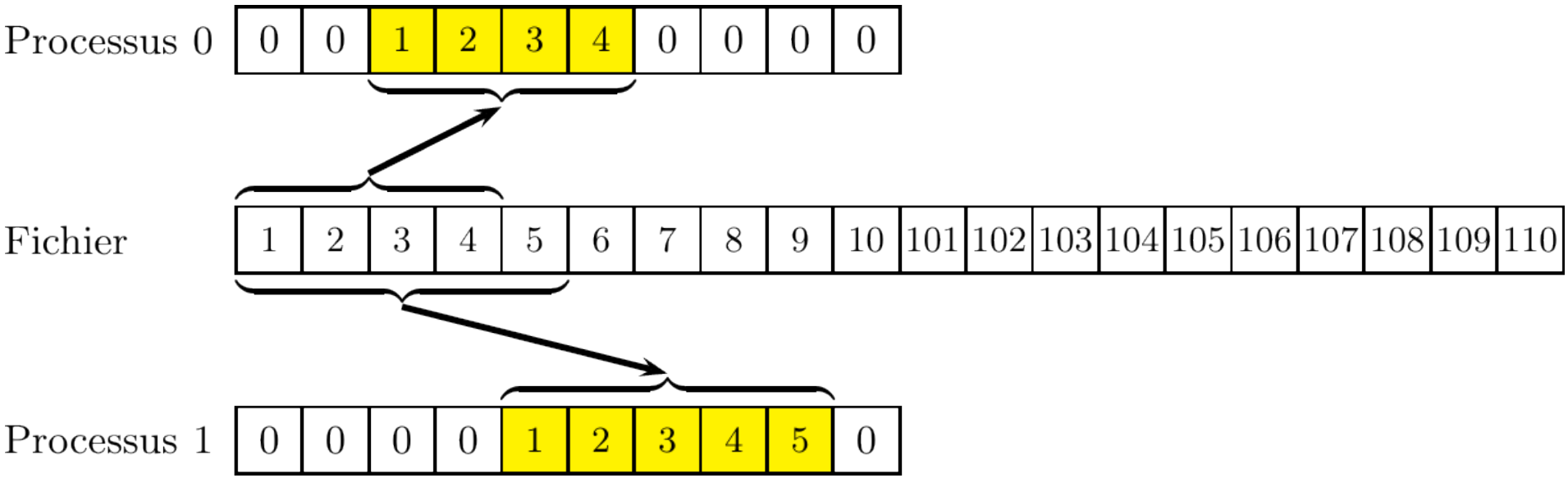


Figure 53 – Exemple 2 d'utilisation de `MPI_File_read_all()`

```
> mpiexec -n 2 read_all02

Lecture processus 1 : 0, 0, 0, 0, 1, 2, 3, 4, 5, 0
Lecture processus 0 : 0, 0, 1, 2, 3, 4, 0, 0, 0, 0
```

MPI-IO

```
1  program read_all03
2    use mpi_f08
3    implicit none
4
5    integer, parameter                  :: nb_valeurs=10
6    integer                             :: rang,code
7    TYPE(MPI_File)                      :: descripteur
8    integer, dimension(nb_valeurs)      :: valeurs=0
9    TYPE(MPI_Status)                    :: statut
10
11   call MPI_INIT()
12   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
13
14   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
15                      descripteur)
16
17   if (rang == 0) then
18      call MPI_FILE_READ_ALL(descripteur,valeurs(3),4,MPI_INTEGER,statut)
19   else
20      call MPI_FILE_READ_ALL(descripteur,valeurs(5),5,MPI_INTEGER,statut)
21   end if
22
23   print *, "Lecture processus",rang,":",valeurs(:)
24
25   call MPI_FILE_CLOSE(descripteur)
26   call MPI_FINALIZE()
27 end program read_all03
```

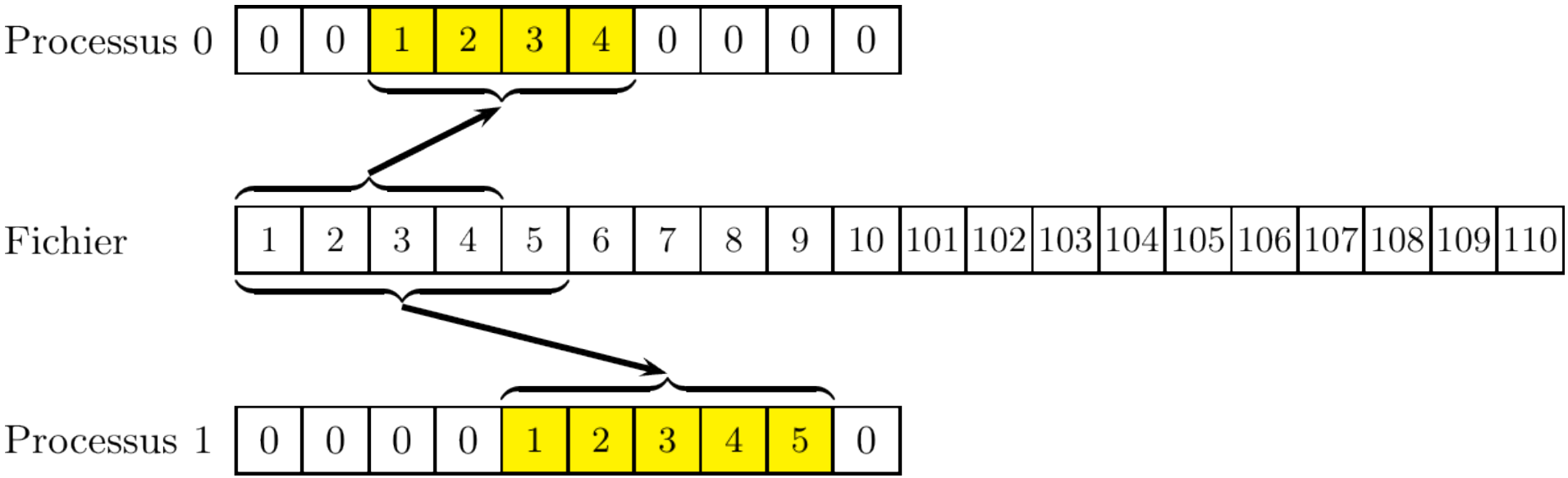


Figure 54 – Exemple 3 d'utilisation de `MPI_File_read_all()`

```
> mpiexec -n 2 read_all03

Lecture processus 1 : 0, 0, 0, 0, 1, 2, 3, 4, 5, 0
Lecture processus 0 : 0, 0, 1, 2, 3, 4, 0, 0, 0, 0
```

MPI-IO

```
1  program read_ordered
2    use mpi_f08
3    implicit none
4
5    integer                              :: rang,code
6    TYPE(MPI_File)                       :: descripteur
7    integer, parameter                   :: nb_valeurs=10
8    integer, dimension(nb_valeurs)       :: valeurs
9    TYPE(MPI_Status)                     :: statut
10
11   call MPI_INIT()
12   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
13
14   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
15                      descripteur)
16
17   call MPI_FILE_READ_ORDERED(descripteur,valeurs,4,MPI_INTEGER,statut)
18   call MPI_FILE_READ_ORDERED(descripteur,valeurs(5),6,MPI_INTEGER,statut)
19
20   print *, "Lecture processus",rang,":",valeurs(:)
21
22   call MPI_FILE_CLOSE(descripteur)
23   call MPI_FINALIZE()
24 end program read_ordered
```

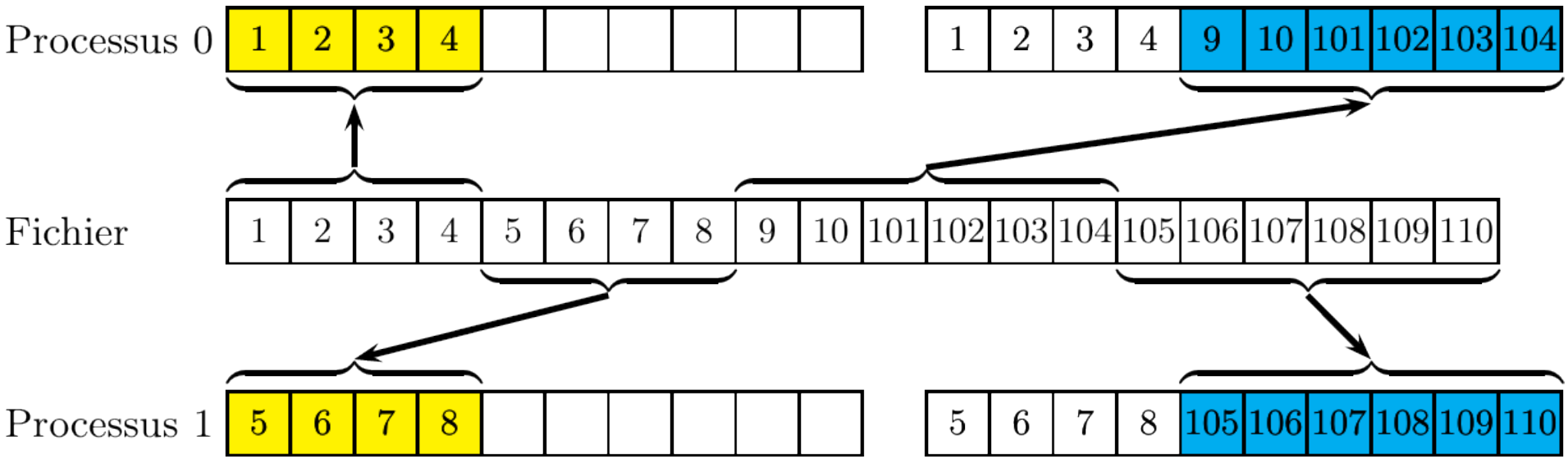


Figure 55 – Exemple d'utilisation de `MPI_File_ordered()`

```
> mpiexec -n 2 read_ordered

Lecture processus 1 : 5, 6, 7, 8, 105, 106, 107, 108, 109, 110
Lecture processus 0 : 1, 2, 3, 4,   9,  10, 101, 102, 103, 104
```

Positionnement explicite des pointeurs dans un fichier

- Les sous-programmes `MPI_File_get_position()` et `MPI_File_get_position_shared()` permettent de connaître respectivement la valeur courante des pointeurs individuels et celle du pointeur partagé.
- Il est possible de positionner explicitement les pointeurs individuels à l'aide du sous-programme `MPI_File_seek()`, et de même le pointeur partagé avec le sous-programme `MPI_File_seek_shared()`.
- Il y a trois modes possibles pour modifier la valeur d'un pointeur :
  - `MPI_SEEK_SET` permet de définir un déplacement absolu ;
  - `MPI_SEEK_CUR` permet un déplacement relativement à la position courante ;
  - `MPI_SEEK_END` positionne le pointeur à la fin du fichier, à laquelle un déplacement éventuel est ajouté.
- Avec `MPI_SEEK_CUR` et `MPI_SEEK_END`, on peut spécifier une valeur négative, ce qui permet de revenir en arrière dans le fichier.

MPI-IO

```
1  program seek
2    use mpi_f08
3    implicit none
4    integer, parameter                  :: nb_valeurs=10
5    integer                             :: rang,nb_octets_entier,code
6    TYPE(MPI_File)                      :: descripteur
7    integer(kind=MPI_OFFSET_KIND)       :: position_fichier
8    integer, dimension(nb_valeurs)      :: valeurs
9    TYPE(MPI_Status)                    :: statut
10
11   call MPI_INIT()
12   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
13   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
14                      descripteur)
15
16   call MPI_FILE_READ(descripteur,valeurs,3,MPI_INTEGER,statut)
17   call MPI_TYPE_SIZE(MPI_INTEGER,nb_octets_entier)
18   position_fichier=8*nb_octets_entier
19   call MPI_FILE_SEEK(descripteur,position_fichier,MPI_SEEK_CUR)
20   call MPI_FILE_READ(descripteur,valeurs(4),3,MPI_INTEGER,statut)
21   position_fichier=4*nb_octets_entier
22   call MPI_FILE_SEEK(descripteur,position_fichier,MPI_SEEK_SET)
23   call MPI_FILE_READ(descripteur,valeurs(7),4,MPI_INTEGER,statut)
24
25   print *, "Lecture processus",rang,":",valeurs(:)
26
27   call MPI_FILE_CLOSE(descripteur)
28   call MPI_FINALIZE()
29 end program seek
```

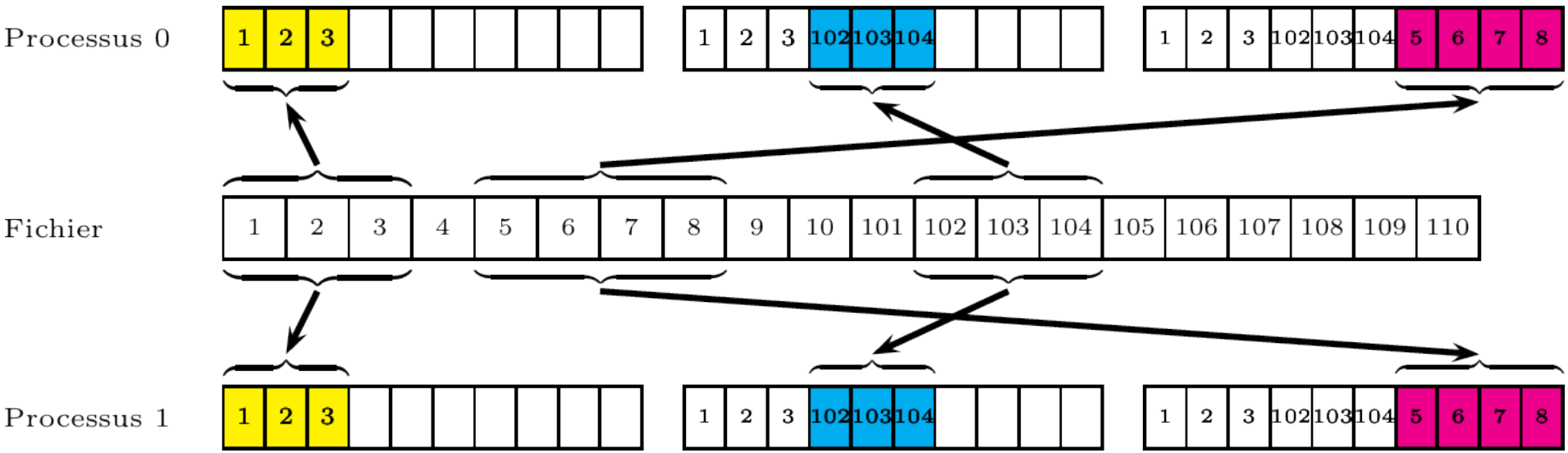


Figure 56 – Exemple d'utilisation de `MPI_File_seek()`

```
> mpiexec -n 2 seek

Lecture processus 1 : 1, 2, 3, 102, 103, 104, 5, 6, 7, 8
Lecture processus 0 : 1, 2, 3, 102, 103, 104, 5, 6, 7, 8
```

Lectures/écritures non bloquantes

- L'intérêt est de faire un recouvrement entre les calculs et les entrées-sorties.
- Les entrées-sorties non bloquantes sont implémentées suivant le modèle utilisé pour les communications non bloquantes entre processus.
- Un accès non-bloquant doit donner lieu ultérieurement à un test explicite de complétude ou à une mise en attente (via `MPI_Test()`, `MPI_Wait()`, etc.)

MPI-IO

```
1  program iread_at
2    use mpi_f08
3    implicit none
4
5    integer, parameter                  :: nb_valeurs=10
6    integer                             :: i,nb_iterations=0,rang,nb_octets_entier,code
7    TYPE(MPI_Request)                   :: requete
8    TYPE(MPI_File)                      :: descripteur
9    integer(kind=MPI_OFFSET_KIND)       :: position_fichier
10   integer, dimension(nb_valeurs)      :: valeurs
11   TYPE(MPI_Status)                    :: statut
12   logical                             :: termine
13
14   call MPI_INIT()
15   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
```

MPI-IO

```
16    call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
17                       descripteur)
18
19    call MPI_TYPE_SIZE(MPI_INTEGER,nb_octets_entier)
20
21    position_fichier=rang*nb_valeurs*nb_octets_entier
22    call MPI_FILE_IREAD_AT(descripteur,position_fichier,valeurs,nb_valeurs, &
23                           MPI_INTEGER,requete)
24
25    do while (nb_iterations < 5000)
26       nb_iterations=nb_iterations+1
27       ! Calculs recouvrant le temps demande par l'operation de lecture
28       !...
29       call MPI_TEST(requete,termine,statut)
30       if (termine) exit
31    end do
32    if (.not. termine) call MPI_WAIT(requete,statut)
33    print *,"Apres",nb_iterations,"iterations, lecture processus",rang,":",valeurs
34
35    call MPI_FILE_CLOSE(descripteur)
36    call MPI_FINALIZE()
37
38  end program iread_at
```

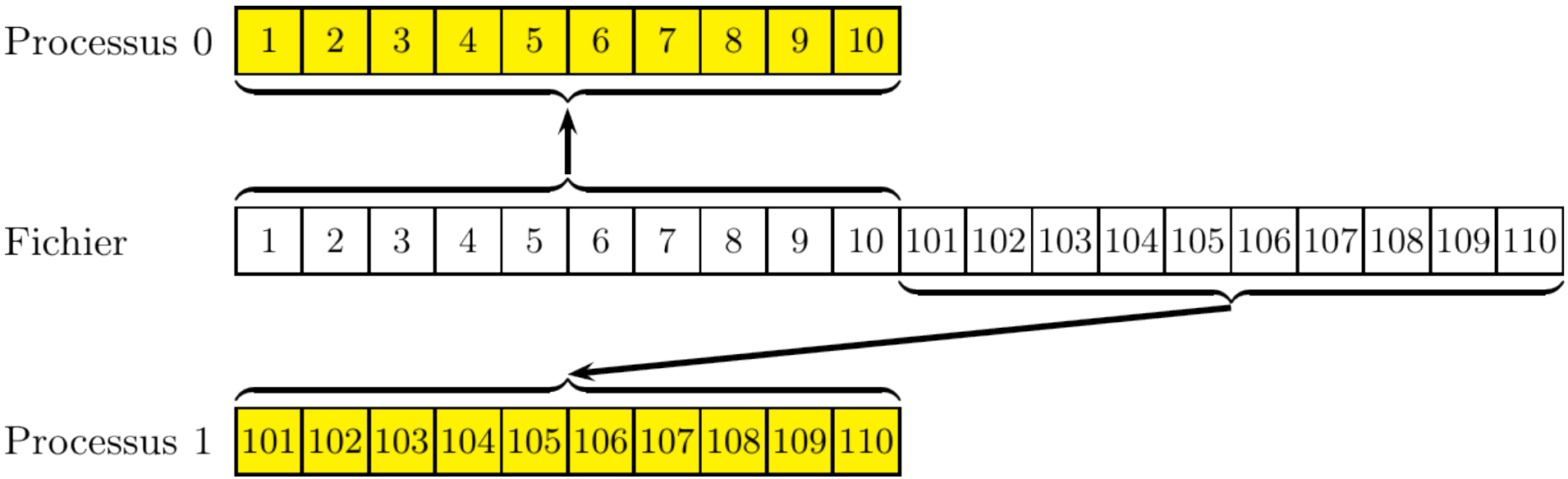


Figure 57 – Exemple d'utilisation de `MPI_File_iread_at()`

```
> mpiexec -n 2 iread_at

Apres 1 iterations, lecture processus 0 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Apres 1 iterations, lecture processus 1 : 101, 102, 103, 104, 105, 106, 107, 108, 109, 110
```

MPI-IO

```
1  program iwrite
2    use mpi_f08
3    implicit none
4
5    integer, parameter                  :: nb_valeurs=10
6    TYPE(MPI_File)                      :: descripteur
7    TYPE(MPI_Request)                   :: requete
8    integer                             :: code, nb_iterations=0
9    integer(kind=MPI_OFFSET_KIND)       :: deplacement
10   integer, dimension(nb_valeurs)      :: valeurs,temp
11   logical                             :: termine
12
13   call MPI_INIT()
14   !...
15   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_WRONLY+MPI_MODE_CREATE,&
16                      MPI_INFO_NULL, descripteur)
17   temp = valeurs
18   call MPI_FILE_SEEK(descripteur,deplacement,MPI_SEEK_SET)
19   call MPI_FILE_IWRITE(descripteur,temp,nb_valeurs,MPI_INTEGER,requete)
20   do while (nb_iterations < 5000)
21     nb_iterations=nb_iterations+1
22     !...
23     call MPI_TEST(requete,termine,MPI_STATUS_IGNORE)
24     if (termine) then
25       temp = valeurs
26       call MPI_FILE_SEEK(descripteur,deplacement,MPI_SEEK_SET)
27       call MPI_FILE_IWRITE(descripteur,temp,nb_valeurs,MPI_INTEGER,requete)
28     end if
29   end do
30   call MPI_WAIT(requete,MPI_STATUS_IGNORE)
31   call MPI_FILE_CLOSE(descripteur)
32   call MPI_FINALIZE()
33 end program iwrite
```

Lectures/écritures collectives et non bloquantes

- Il est possible d'effectuer des opérations qui soient à la fois collectives et non bloquantes.
- Il ne peut y avoir qu'une seule opération collective non bloquante en cours à la fois par processus.
- Entre les deux phases de l'opération collective non-bloquante, il est possible de faire des opérations non collectives sur le fichier, mais la zone mémoire concernée par l'opération collective ne peut être modifiée.

MPI-IO

```
1  program iread_all
2
3    use mpi_f08
4    implicit none
5
6    integer                              :: rang,code
7    TYPE(MPI_File)                       :: descripteur
8    integer, parameter                   :: nb_valeurs=10
9    integer, dimension(nb_valeurs)       :: valeurs
10   TYPE(MPI_Status)                     :: statut
11   TYPE(MPI_Request)                    :: req
12
13   call MPI_INIT()
14   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
15
16   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
17                      descripteur)
18
19   call MPI_FILE_IREAD_ALL(descripteur,valeurs,4,MPI_INTEGER,req)
20   print *, "Processus numero    :",rang
21   call MPI_WAIT(req,statut)
22
23   print *, "Lecture processus",rang,":",valeurs(1:4)
24
25   call MPI_FILE_CLOSE(descripteur)
26   call MPI_FINALIZE()
27
28 end program iread_all
```

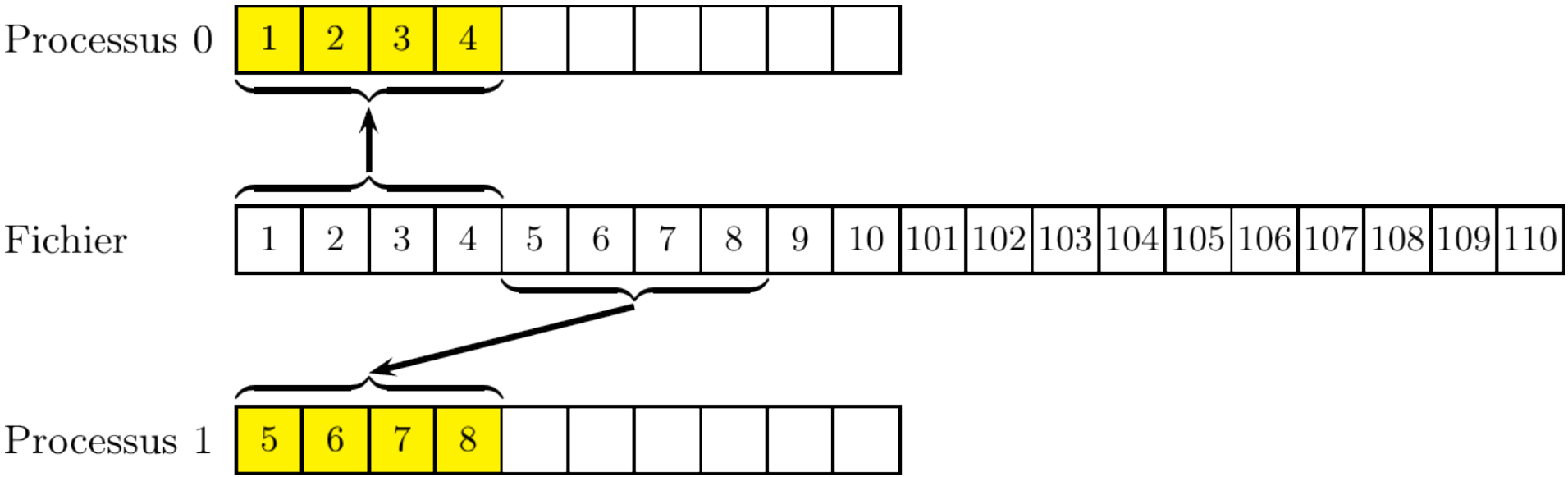


Figure 58 – Exemple d'utilisation de `MPI_File_iread_all()`

```
> mpiexec -n 2 iread_all

Processus numero    : 0
Lecture processus 0 : 1, 2, 3, 4
Processus numero    : 1
Lecture processus 1 : 1, 2, 3, 4
```

T.P. MPI – Exercice 7 : Lecture d'un fichier en mode parallèle

- On dispose du fichier binaire donnees.dat, constitué d'une suite de 484 valeurs entières
- En considérant un programme parallèle mettant en œuvre 4 processus, il s'agit de lire les 121 premières valeurs sur le processus 0, les 121 suivantes sur le processus 1, etc. et d'écrire celles-ci dans quatre fois quatre fichiers appelés fichier_XXX0.dat ··· fichier_XXX3.dat
- On emploiera pour ce faire 4 méthodes différentes, parmi celles présentées :
  - lecture via des déplacements explicites, en mode individuel ;
  - lecture via les pointeurs partagés, en mode collectif ;
  - lecture via les pointeurs individuels, en mode individuel ;
  - lecture via les pointeurs partagés, en mode individuel.
- Pour compiler utilisez la commande make, pour exécuter le code utilisez la commande make exe et pour vérifier les résultats utilisez la commande make verification qui génère des fichiers images correspondant aux quatre cas à traiter.

MPI 4.x

Fortran

Interopérabilité

Par exemple pour `MPI_RECV()` on a comme interface avec le module mpi :

```
<type> buf(*)
INTEGER :: count, datatype, source, tag, comm, ierror
INTEGER, DIMENSION(MPI_STATUS_SIZE) :: statut
```

Avec le module mpi_f08 :

```
TYPE(*), DIMENSION(..) :: buf
INTEGER                :: count, source, tag
TYPE(MPI_DATATYPE)     :: datatype
TYPE(MPI_COMM)         :: comm
TYPE(MPI_STATUS)       :: statut
INTEGER, optional      :: ierror
```

Les nouveaux types sont en fait des encapsulations d'INTEGER

```
TYPE, BIND(C) :: MPI_COMM
  INTEGER :: MPI_VAL
END TYPE MPI_COMM
```

Avec le module mpi_f08 l'expression `comm%mpi_val` est equivalente à l'argument `comm` du module mpi.

MPI 4.x

Ajout

- Grand nombre
- Communication par morceaux
- MPI Session
- Autres

Grand nombre

- Les nombres d'éléments étaient en `integer` ou `int`.
- MPI 4.0 ajoute des fonctions avec `MPI_Count` à la place.
- En *C* ces nouvelles fonctions contiennent en plus `_c` à la fin de la fonction.

```
int MPI_Send(const void * buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);
int MPI_Send_c(const void * buf, MPI_Count count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);
```

- En *Fortran* les nombres en `integer` peuvent être remplacés par `integer(kind=MPI_COUNT_KIND)`.
- Uniquement disponible avec le module `mpi_f08`.
- Pas de changement de nom grâce au polymorphisme.

```
MPI_Send(buf,count,datatype,dest,tag,comm,ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: buf
INTEGER, INTENT(IN)                :: count, dest, tag
TYPE(MPI_Datatype), INTENT(IN)     :: datatype
TYPE(MPI_Comm), INTENT(IN)         :: comm
INTEGER, OPTIONAL, INTENT(OUT)     :: ierror

MPI_Send(buf,count,datatype,dest,tag,comm,ierror)
TYPE(*), DIMENSION(..), INTENT(IN)        :: buf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN)            :: datatype
INTEGER, INTENT(IN)                       :: dest, tag
TYPE(MPI_Comm), INTENT(IN)                :: comm
INTEGER, OPTIONAL, INTENT(OUT)            :: ierror
```

Communication par morceaux

- Contribution multiple à une communication.
- Utile pour l'hybride.
- Initialisation avec `MPI_Psend_init()` ou `MPI_Precv_init()` en fournissant le nombre d'éléments par partition et le nombre de partitions.
- `MPI_Start()` pour démarrer la communication.
- `MPI_Pready()` pour signaler qu'une partition est prête.
- Il n'est pas possible de faire un `MPI_Recv()` d'un `MPI_Psend_init()`
- `MPI_Wait()` pour terminer la communication
- `MPI_Parrived()` permet de savoir si une partition a été reçue

Sessions

- Permettre de faire plusieurs `MPI_Init()`/`MPI_Finalize()`.
- `MPI_Session_init()` pour lancer une session.
- `MPI_Session_finalize()` pour terminer la session.
- Plus de notion de `MPI_COMM_WORLD`.
- Notion de *Process Sets* : `mpi://WORLD` et `mpi://SELF`.
- `MPI_Group_from_session_pset()` pour faire un groupe à partir d'un *pset*.
- `MPI_Comm_create_from_group()` pour faire un communicateur à partir d'un groupe.
- `MPI_Session_get_num_psets()` permet d'obtenir le nombre de *pset* disponible.
- `MPI_Session_get_nth_pset()` permet d'avoir le nom d'un *pset* disponible.

Autres

- Ajout de `MPI_Isendrecv` et `MPI_Isendrecv_replace`.
- Ajout des communications collectives persistentes.
- Ajout de l'option `mpi_initial_errhandler` pour *mpiexec* afin de spécifier le gestionnaire d'erreur par défaut.

MPI-IO Vues

MPI-IO Vues

Définition des vues

- C'est un mécanisme permettant de décrire un schéma d'accès aux fichiers.
- Une vue est définie par trois variables : un déplacement initial, un type élémentaire de données et un motif.
- L'accès aux fichiers s'effectue par répétition du motif, une fois le positionnement initial effectué.

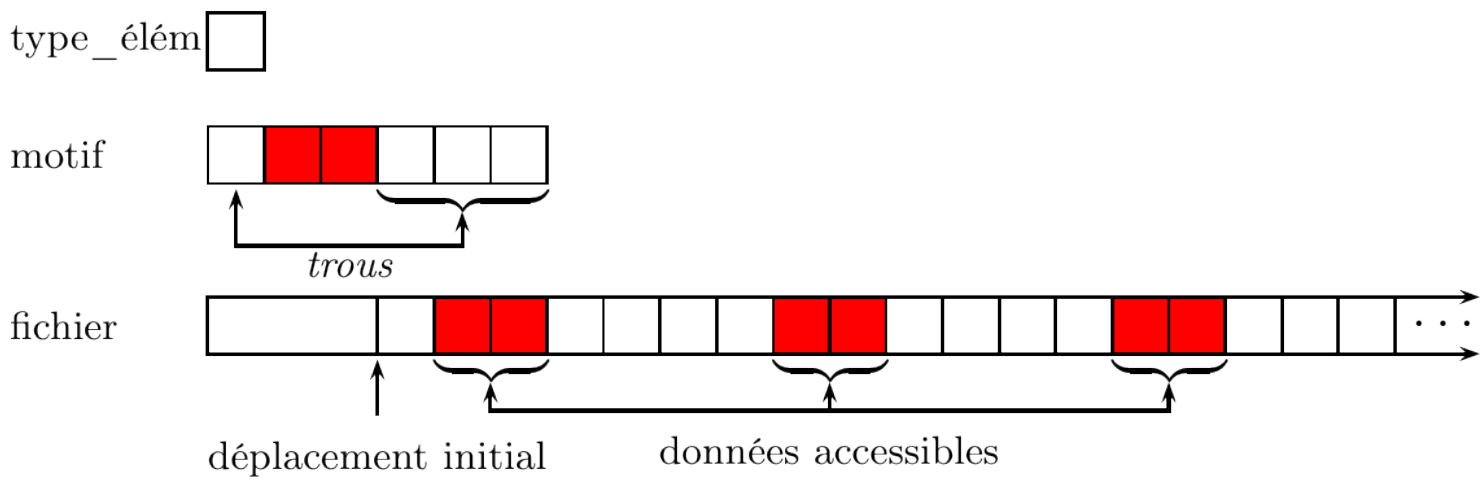


Figure 59 – Type élémentaire de donnée et motif

MPI-IO Vues

Définition des vues

- Les vues sont construites à l'aide de types dérivés MPI.
- Il est possible de définir des trous dans une vue, de façon à ne pas tenir compte de certaines parties des données.
- La vue par défaut consiste en une simple suite d'octets (déplacement initial nul, type_élém et motif égaux à `MPI_BYTE`).

Vues multiples

- Un processus donné peut définir et utiliser successivement plusieurs vues d'un même fichier.
- Les processus peuvent avoir des vues différentes du fichier, de façon à accéder à des parties complémentaires de celui-ci.

MPI-IO Vues

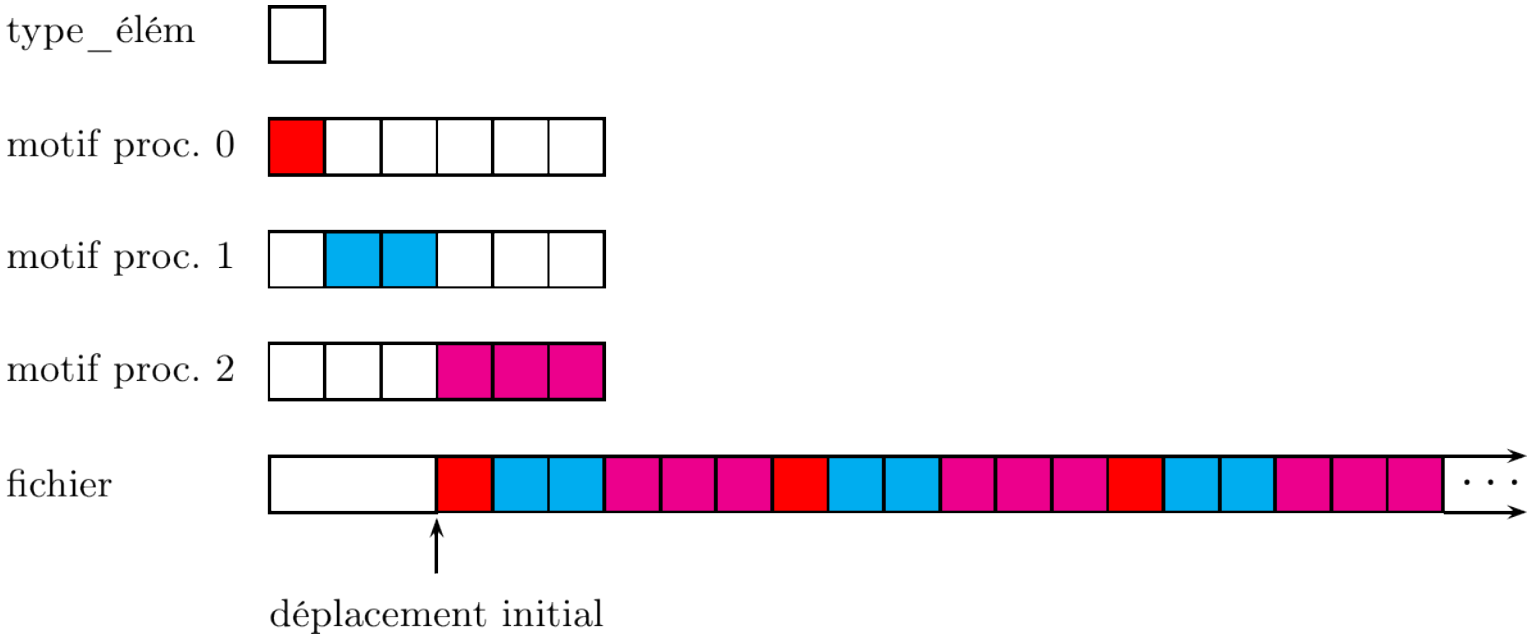


Figure 60 – Exemple de définition de motifs différents selon les processus

Remarques :

- Un pointeur partagé n'est utilisable avec une vue que si tous les processus ont la même vue.
- Si le fichier est ouvert en écriture, les zones décrites par les différentes vues ne peuvent se recouvrir, même partiellement.

MPI-IO Vues

Changement de la vue sur un fichier : `MPI_File_set_view()`

```
MPI_FILE_SET_VIEW(descripteur,
                  deplacement_initial,type_elem,motif,
                  mode,info,code)

TYPE(MPI_File), intent(in)                 :: descripteur
integer(kind=MPI_OFFSET_KIND), intent(in)  :: deplacement_initial
TYPE(MPI_Datatype), intent(in)             :: type_elem, motif
character(len=*), intent(in)               :: mode
TYPE(MPI_Info), intent(in)                 :: info
integer, optional, intent(out)             :: code
```

- C'est une opération collective à l'ensemble des processus impliqués dans l'accès au fichier. Chaque processus peut définir un déplacement initial et un motif différent. L'étendue du type élémentaire doit être identique.
- Les pointeurs individuels et le pointeur partagé sont réinitialisés au début de la vue.

Notes :

- Les types dérivés utilisés dans la vue doivent avoir été validés au préalable à l'aide du sous-programme `MPI_Type_commit()`.
- Il y a trois représentations possibles des données (mode) : "native", "internal" ou "external32".

Construction de sous-tableaux

Un type dérivé utile pour créer un motif est le type “subarray”, qu’on introduit ici. Ce type permet de créer un sous-tableau à partir d’un tableau et se définit via le sous-programme `MPI_Type_create_subarray()`.

Le profil d’un tableau est un vecteur dont chaque élément est le nombre d’éléments dans chaque dimension. Soit par exemple le tableau `T(10,0:5,-10:10)` (ou `T[10][6][21]`), son profil est le vecteur (10,6,21).

MPI-IO Vues / Types de données dérivés

```
MPI_TYPE_CREATE_SUBARRAY(nb_dims,profil_tab,profil_sous_tab,coord_debut,
                         ordre,ancien_type,nouveau_type,code)

integer, intent(in)                        :: nb_dims
integer, dimension(nb_dims), intent(in) :: profil_tab,profil_sous_tab,coord_debut
integer, intent(in)                        :: ordre,
TYPE(MPI_Datatype), intent(in)             :: ancien_type
TYPE(MPI_Datatype), intent(out)            :: nouveau_type
integer, optional, intent(out)             :: code
```

Description des arguments

- `nb_dims` : nombre de dimension du tableau
- `profil_tab` : profil du tableau à partir duquel on va extraire un sous-tableau
- `profil_sous_tab` : profil du sous-tableau
- `coord_debut` : coordonnées de départ du sous-tableau dans le tableau, les indices du tableau commençant à 0. Par exemple, si on veut que les coordonnées de départ du sous-tableau soient `tab(2,3)`, il faut que `coord_debut(:)=(/ 1,2 /)`
- `ordre` : ordre de stockage des éléments
  - `MPI_ORDER_FORTRAN` spécifie le mode de stockage en Fortran, c.-à-d. suivant les colonnes
  - `MPI_ORDER_C` spécifie le mode de stockage en C, c.-à-d. suivant les lignes

MPI-IO Vues / Types de données dérivés

Échanges entre 2 processus avec subarray

AVANT

1	5	9
2	6	10
3	7	11
4	8	12

Processus 0

-1	-5	-9
-2	-6	-10
-3	-7	-11
-4	-8	-12

Processus 1

APRÈS

1	5	9
-7	-11	10
-8	-12	11
4	8	12

Processus 0

-1	-5	-9
-2	-6	-10
-3	2	6
-4	3	7

Processus 1

MPI-IO Vues / Types de données dérivés

Échanges entre 2 processus avec subarray : code

```
1  program subarray
2
3    use mpi_f08
4    implicit none
5
6    integer,parameter                          :: nb_lignes=4,nb_colonnes=3,&
7                                                  etiquette=1000,nb_dims=2
8    integer                                    :: code,rang,i
9    TYPE(MPI_Datatype)                         :: type_sous_tab
10   integer,dimension(nb_lignes,nb_colonnes)   :: tab
11   integer,dimension(nb_dims)                 :: profil_tab,profil_sous_tab,coord_debut
12   TYPE(MPI_Status)                           :: statut
13
14   call MPI_INIT()
15   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
16
17   ! Initialisation du tableau tab sur chaque processus
18   tab(:,:) = reshape( (/ (sign(i,-rang),i=1,nb_lignes*nb_colonnes) /) , &
19                       (/ nb_lignes,nb_colonnes /) )
```

MPI-IO Vues / Types de données dérivés

Échanges entre 2 processus avec subarray : code (suite)

```
20    ! Profil du tableau tab a partir duquel on va extraire un sous-tableau
21    profil_tab(:) = shape(tab)
22    ! La fonction F95 shape donne le profil du tableau passe en argument.
23    ! ATTENTION, si le tableau concerne n'a pas ete alloue sur tous les processus,
24    ! il faut mettre explicitement le profil du tableau pour qu'il soit connu
25    ! sur tous les processus, soit profil_tab(:) = (/ nb_lignes,nb_colonnes /)
26
27    ! Profil du sous-tableau
28    profil_sous_tab(:) = (/ 2,2 /)
29
30    ! Coordonnees de depart du sous-tableau
31    ! Pour le processus 0 on part de l'element tab(2,1)
32    ! Pour le processus 1 on part de l'element tab(3,2)
33    coord_debut(:) = (/ rang+1,rang /)
34
35    ! Creation du type derive type_sous_tab
36    call MPI_TYPE_CREATE_SUBARRAY(nb_dims,profil_tab,profil_sous_tab,coord_debut,&
37                                  MPI_ORDER_FORTRAN,MPI_INTEGER,type_sous_tab)
38    call MPI_TYPE_COMMIT(type_sous_tab)
39
40    ! Permutation du sous-tableau
41    call MPI_SENDRECV_REPLACE(tab,1,type_sous_tab,mod(rang+1,2),etiquette,&
42                              mod(rang+1,2),etiquette,MPI_COMM_WORLD,statut)
43    call MPI_TYPE_FREE(type_sous_tab)
44    call MPI_FINALIZE()
45  end program subarray
```

Exemple 1 : lecture d'un fichier par blocs de deux éléments

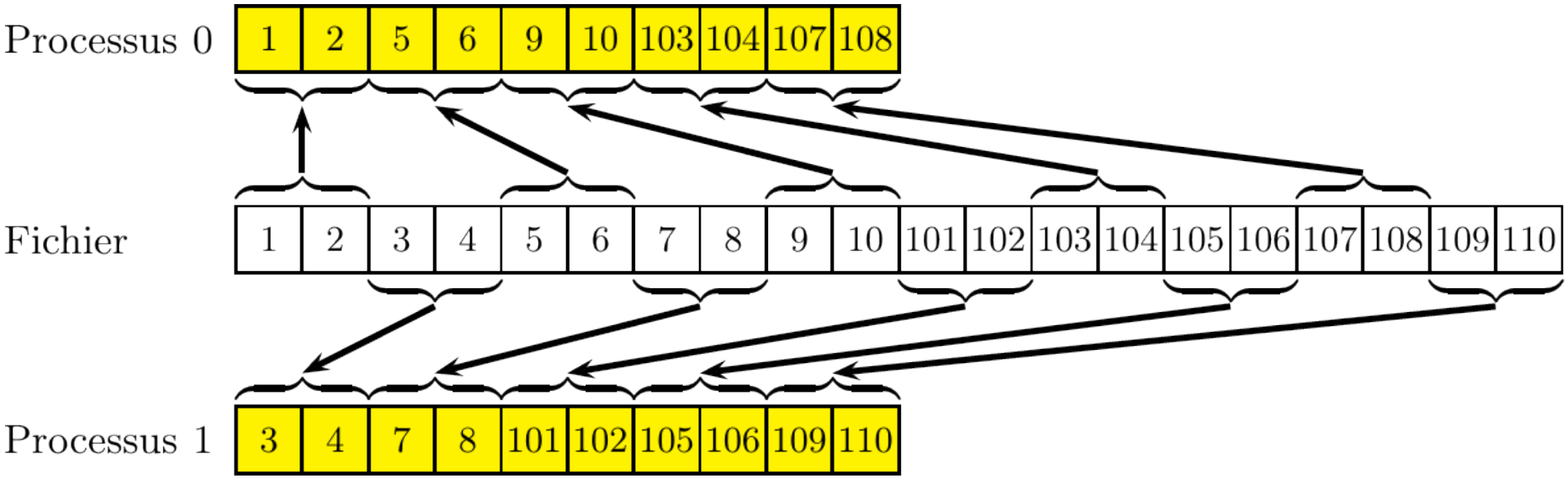


Figure 61 – Exemple 1 : lecture d'un fichier par blocs de deux éléments

```
> mpiexec -n 2 read_view01

Lecture processus 1 : 3, 4, 7, 8, 101, 102, 105, 106, 109, 110
Lecture processus 0 : 1, 2, 5, 6,   9,  10, 103, 104, 107, 108
```

MPI-IO Vues

Exemple 1 (suite)

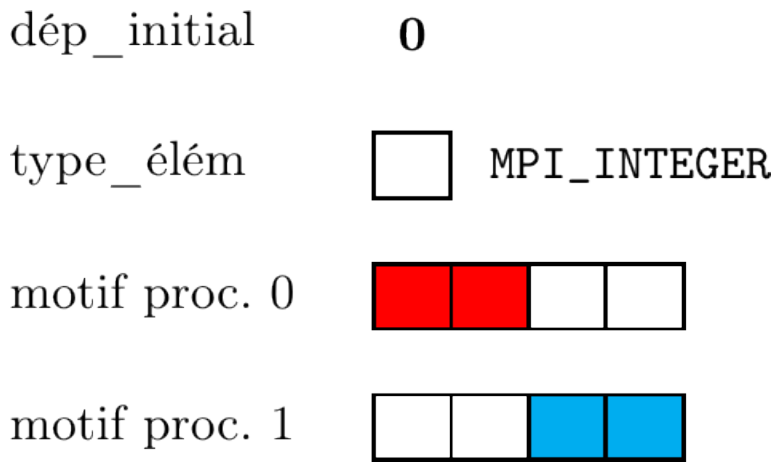


Figure 62 – Exemple 1 (suite) : création des vues sur le fichier

```
1   if (rang == 0) coord=1
2   if (rang == 1) coord=3
3
4   call MPI_TYPE_CREATE_SUBARRAY(1,(/4/),(/2/),(/coord - 1/), &
5                                 MPI_ORDER_FORTRAN,MPI_INTEGER,motif)
6   call MPI_TYPE_COMMIT(motif)
7
8   deplacement_initial=0
9
10  call MPI_FILE_SET_VIEW(descripteur,deplacement_initial,MPI_INTEGER,motif, &
11                         "native",MPI_INFO_NULL)
```

MPI-IO Vues

Exemple 1 : code complet

```
1  program read_view01
2    use mpi_f08
3    implicit none
4    integer, parameter                  :: nb_valeurs=10
5    integer                             :: rang,coord,code
6    TYPE(MPI_Datatype)                  :: motif
7    TYPE(MPI_File)                      :: descripteur
8    integer(kind=MPI_OFFSET_KIND)       :: deplacement_initial
9    integer, dimension(nb_valeurs)      :: valeurs
10   TYPE(MPI_Status)                    :: statut
11   call MPI_INIT()
12   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
13
14   if (rang == 0) coord=1
15   if (rang == 1) coord=3
16
17   call MPI_TYPE_CREATE_SUBARRAY(1,(/4/),(/2/),(/coord - 1/), &
18                                 MPI_ORDER_FORTRAN,MPI_INTEGER,motif)
19   call MPI_TYPE_COMMIT(motif)
20
21   call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
22                      descripteur)
23
24   deplacement_initial=0
25   call MPI_FILE_SET_VIEW(descripteur,deplacement_initial,MPI_INTEGER,motif, &
26                          "native",MPI_INFO_NULL)
27   call MPI_FILE_READ(descripteur,valeurs,nb_valeurs,MPI_INTEGER,statut)
28
29   print *, "Lecture processus",rang,":",valeurs(:)
30
31   call MPI_FILE_CLOSE(descripteur)
32   call MPI_FINALIZE()
33 end program read_view01
```

MPI-IO Vues

Exemple 2 : utilisation successive de plusieurs vues

Figure 63 – Exemple 2 : utilisation successive de plusieurs vues

```
1  program read_view02
2
3    use mpi_f08
4    implicit none
5
6    integer, parameter                  :: nb_valeurs=10
7    integer                             :: rang,code, nb_octets_entier
8    TYPE(MPI_File)                      :: descripteur
9    TYPE(MPI_Datatype)                  :: motif_1,motif_2
10   integer(kind=MPI_OFFSET_KIND)       :: deplacement_initial
11   integer, dimension(nb_valeurs)      :: valeurs
12   TYPE(MPI_Status)                    :: statut
13
14   call MPI_INIT()
15   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
```

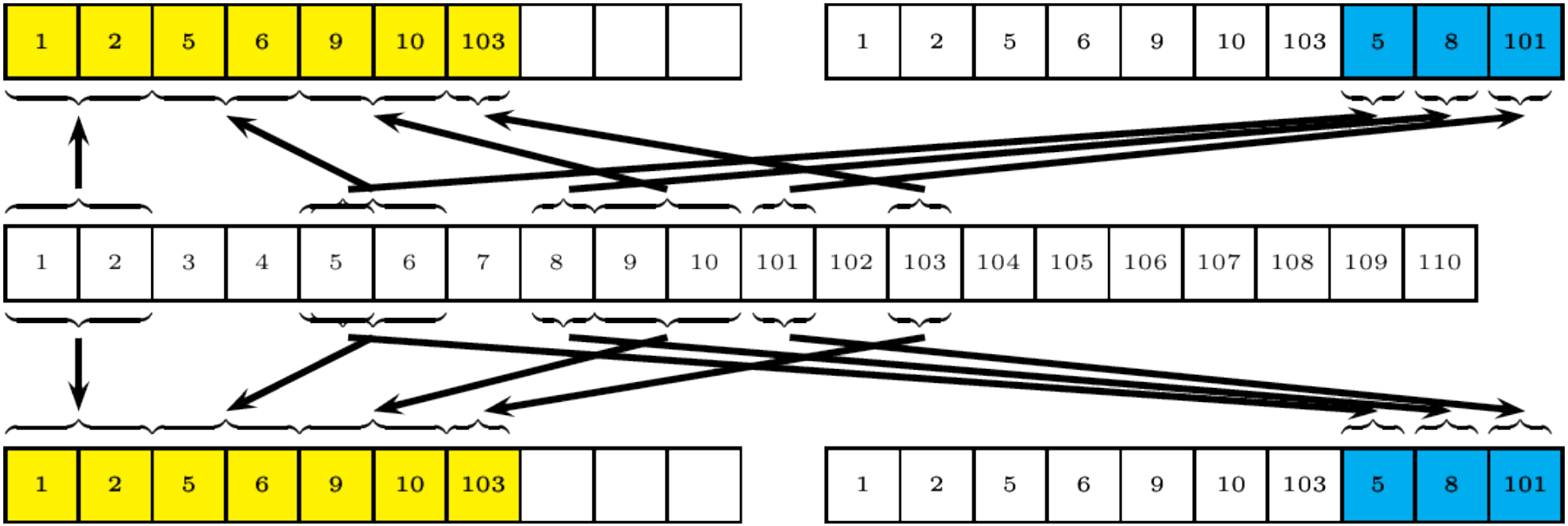
MPI-IO Vues

Exemple 2 (suite du code)

```
16    call MPI_TYPE_CREATE_SUBARRAY(1,(/4/),(/2/),(/0/), &
17                                  MPI_ORDER_FORTRAN,MPI_INTEGER,motif_1)
18    call MPI_TYPE_COMMIT(motif_1)
19
20    call MPI_TYPE_CREATE_SUBARRAY(1,(/3/),(/1/),(/2/), &
21                                  MPI_ORDER_FORTRAN,MPI_INTEGER,motif_2)
22    call MPI_TYPE_COMMIT(motif_2)
23
24    call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
25                       descripteur)
26
27    ! Lecture en utilisant la premiere vue
28    deplacement_initial=0
29    call MPI_FILE_SET_VIEW(descripteur,deplacement_initial,MPI_INTEGER,motif_1, &
30                           "native",MPI_INFO_NULL)
31    call MPI_FILE_READ(descripteur,valeurs,4,MPI_INTEGER,statut)
32    call MPI_FILE_READ(descripteur,valeurs(5),3,MPI_INTEGER,statut)
33
34    ! Lecture en utilisant la seconde vue
35    call MPI_TYPE_SIZE(MPI_INTEGER,nb_octets_entier)
36    deplacement_initial=2*nb_octets_entier
37    call MPI_FILE_SET_VIEW(descripteur,deplacement_initial,MPI_INTEGER,motif_2, &
38                           "native",MPI_INFO_NULL)
39    call MPI_FILE_READ(descripteur,valeurs(8),3,MPI_INTEGER,statut)
40
41    print *, "Lecture processus",rang,":",valeurs(:)
42
43    call MPI_FILE_CLOSE(descripteur)
44    call MPI_FINALIZE()
45  end program read_view02
```

MPI-IO Vues

Exemple 2 : illustration




```
> mpiexec -n 2 read_view02

Lecture processus 1 : 1, 2, 5, 6, 9, 10, 103, 5, 8, 101
Lecture processus 0 : 1, 2, 5, 6, 9, 10, 103, 5, 8, 101
```

MPI-IO Vues

Exemple 3 : gestion des trous dans les types de données

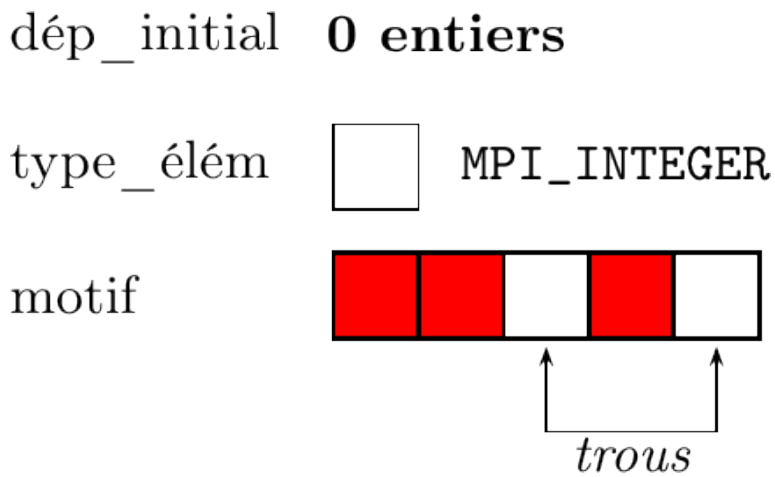


Figure 64 – Exemple 3 : gestion des trous dans les types de données

```
1  program read_view03_indexed
2    use mpi_f08
3    implicit none
4    integer, parameter                      :: nb_valeurs=9
5    integer                                 :: rang,nb_octets_entier,code
6    TYPE(MPI_File)                          :: descripteur
7    TYPE(MPI_Datatype)                      :: motif_temp,motif
8    integer(kind=MPI_OFFSET_KIND)           :: deplacement_initial
9    integer(kind=MPI_ADDRESS_KIND)          :: borne_inf,etendue
10   integer, dimension(2)                   :: longueurs,deplacements
11   integer, dimension(nb_valeurs)          :: valeurs
12   TYPE(MPI_Status)                        :: statut
13   call MPI_INIT()
14   call MPI_COMM_RANK(MPI_COMM_WORLD,rang)
```

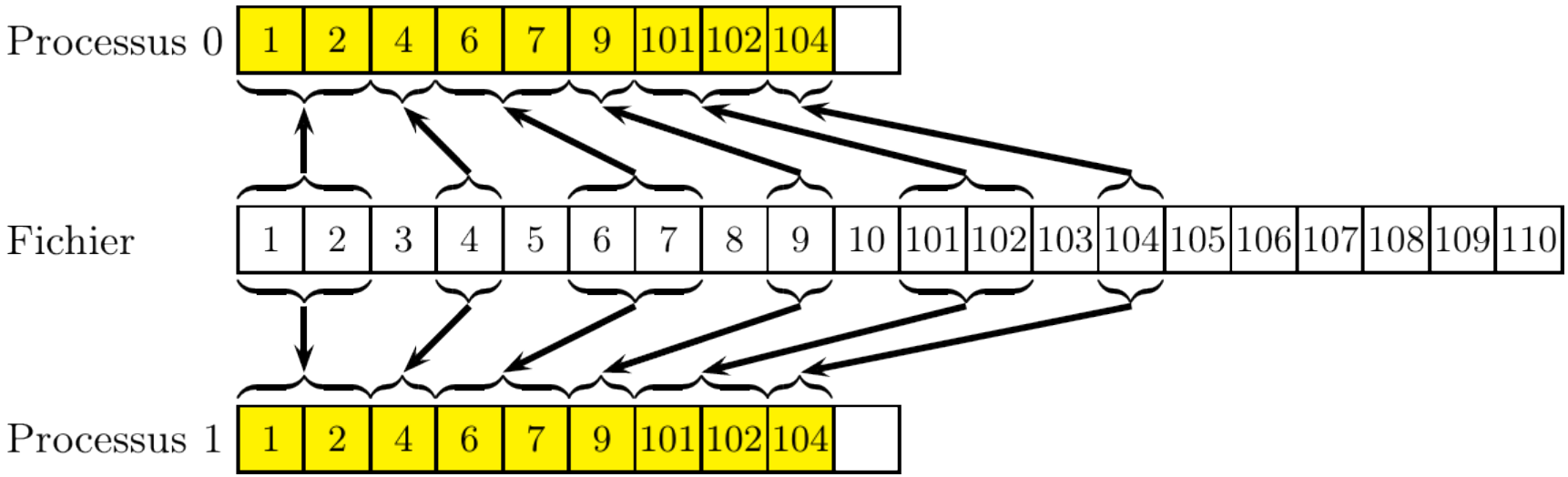
MPI-IO Vues

Exemple 3 (suite du code)

```
15    ! motif_temp: type MPI d'etendu 4*MPI_INTEGER
16    deplacements(1)=0
17    longueurs(1)=2
18    deplacements(2)=3
19    longueurs(2)=1
20    call MPI_TYPE_INDEXED(2,longueurs,deplacements,MPI_INTEGER,motif_temp)
21
22    ! motif: type MPI d'etendu 5*MPI_INTEGER
23    call MPI_TYPE_SIZE(MPI_INTEGER,nb_octets_entier)
24    call MPI_TYPE_GET_EXTENT(motif_temp,borne_inf,etendue)
25    etendue = etendue + nb_octets_entier
26    call MPI_TYPE_CREATE_RESIZED(motif_temp,borne_inf,borne_inf+etendue,motif)
27    call MPI_TYPE_COMMIT(motif)
28
29    call MPI_FILE_OPEN(MPI_COMM_WORLD,"donnees.dat",MPI_MODE_RDONLY,MPI_INFO_NULL, &
30                       descripteur)
31
32    deplacement_initial=0
33    call MPI_FILE_SET_VIEW(descripteur,deplacement_initial,MPI_INTEGER,motif, &
34                           "native",MPI_INFO_NULL)
35
36    call MPI_FILE_READ(descripteur,valeurs,9,MPI_INTEGER,statut)
37
38    print *,"Lecture processus",rang,":",valeurs(:)
39
40    call MPI_FILE_CLOSE(descripteur)
41    call MPI_FINALIZE()
42
43  end program read_view03_indexed
```

MPI-IO Vues

Exemple 3 : illustration




```
> mpiexec -n 2 read_view03

Lecture, processus 0 : 1, 2, 4, 6, 7, 9, 101, 102, 104
Lecture, processus 1 : 1, 2, 4, 6, 7, 9, 101, 102, 104
```

MPI-IO Vues

Exemple 3 : implémentation alternative utilisant un type structure

```
1  program read_view03_struct
2    [...]
3    integer(kind=MPI_ADDRESS_KIND), dimension(2) :: deplacements
4    [...]
5
6    call MPI_TYPE_CREATE_SUBARRAY(1,(/3/),(/2/),(/0/),MPI_ORDER_FORTRAN, &
7         MPI_INTEGER,temp_motif1)
8
9    call MPI_TYPE_CREATE_SUBARRAY(1,(/2/),(/1/),(/0/),MPI_ORDER_FORTRAN, &
10        MPI_INTEGER,temp_motif2)
11
12   call MPI_TYPE_SIZE(MPI_INTEGER,nb_octets_entier)
13
14   deplacements(1) = 0
15   deplacements(2) = 3*nb_octets_entier
16
17   call MPI_TYPE_CREATE_STRUCT(2,(/1,1/),deplacements,&
18        (/temp_motif1,temp_motif2/),motif)
19   call MPI_TYPE_COMMIT(motif)
20
21   [...]
22
23 end program read_view03_struct
```

MPI-IO Vues

Conclusion

MPI-IO offre une interface de haut niveau et un ensemble de fonctionnalités très riche. Il est possible de réaliser des opérations complexes et de tirer parti des optimisations implémentées dans la bibliothèque. MPI-IO offre aussi une bonne portabilité.

Conseils

- L'utilisation des sous-programmes à positionnement explicite dans les fichiers doit être réservée à des cas particuliers, l'utilisation implicite de pointeurs individuels avec des vues offrant une interface de plus haut niveau.
- Lorsque les opérations font intervenir l'ensemble des processus (ou un sous-ensemble identifiable par un sous-communicateur MPI), il faut généralement privilégier la forme collective des opérations.
- Exactement comme pour le traitement des messages lorsque ceux-ci représentent une part importante de l'application, le non-bloquant est une voie privilégiée d'optimisation à mettre en œuvre par les programmeurs, mais ceci ne doit être implémenté qu'après qu'on se soit assuré du comportement correct de l'application en mode bloquant.

Conclusion

Conclusion

- Utiliser les communications point-à-point bloquantes, ceci avant de passer aux communications non-bloquantes. Il faudra alors essayer de faire du recouvrement calcul/communications.
- Utiliser les fonctions d'entrées-sorties bloquantes, ceci avant de passer aux entrées-sorties non-bloquantes. De même, il faudra alors faire du recouvrement calcul/entrées-sorties.
- Écrire les communications comme si les envois étaient synchrones (`MPI_Ssend()`).
- Éviter les barrières de synchronisation (`MPI_Barrier()`), surtout sur les fonctions collectives qui sont bloquantes.
- La programmation mixte MPI/OpenMP peut apporter des gains d'extensibilité, mais pour que cette approche fonctionne bien, il est évidemment nécessaire d'avoir de bonnes performances OpenMP à l'intérieur de chaque processus MPI. Un cours est dispensé à l'IDRIS (`https://cours.idris.fr`).

Travaux pratiques MPI – Exercice 8 : Équation de Poisson

On considère l'équation de Poisson suivante :

$$\begin{cases} \dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2} & = f(x,y) \quad \text{dans } [0,1]\text{x}[0,1] \\ u(x,y) & = 0. \quad \text{sur les frontières} \\ f(x,y) & = 2.\left(x^2 - x + y^2 - y\right) \end{cases}$$

On va résoudre cette équation avec une méthode de décomposition de domaine :

- L'équation est discretisée sur le domaine via la méthode des différences finies.
- Le système obtenu est résolu avec un solveur suivant la méthode de Jacobi.
- Le domaine global est découpé en sous domaines.

La solution exacte est connue et est $u_{exacte}(x,y) = xy\,(x-1)\,(y-1)$

Travaux pratiques MPI – Exercice 8 : Équation de Poisson

Pour discrétiser l'équation, on définit une grille constituée d'un ensemble de points (x_i, y_j)

$$
\begin{aligned}
x_i &= i\ h_x \quad \text{pour } i = 0, \ldots, ntx + 1 \\
y_j &= j\ h_y \quad \text{pour } j = 0, \ldots, nty + 1
\end{aligned}
$$

$$
\begin{aligned}
h_x &= \frac{1}{(ntx + 1)} \\
h_y &= \frac{1}{(nty + 1)}
\end{aligned}
$$

h_x : pas suivant x
h_y : pas suivant y
ntx : nombre de points intérieurs suivant x
nty : nombre de points intérieurs suivant y

Il y a au total *ntx+2* points suivant x et *nty+2* points suivant y

Travaux pratiques MPI – Exercice 8 : Équation de Poisson

- Soit u_{ij} l'estimation de la solution à la position $x_i = ih_x$ et $x_j = jh_y$.
- La méthode de Jacobi consiste à calculer :

$$u_{ij}^{n+1} = c_0(c_1(u_{i+1j}^n + u_{i-1j}^n) + c_2(u_{ij+1}^n + u_{ij-1}^n) - f_{ij})$$

avec :

$$c_0 = \frac{1}{2}\frac{h_x^2 h_y^2}{h_x^2 + h_y^2}$$

$$c_1 = \frac{1}{h_x^2}$$

$$c_2 = \frac{1}{h_y^2}$$

Travaux pratiques MPI – Exercice 8 : Équation de Poisson

- En parallèle, les valeurs aux interfaces des sous-domaines doivent être échangées entre les voisins.
- On utilise des cellules fantômes, ces cellules servent de buffer de réception pour les échanges entre voisins.

Travaux pratiques MPI – Exercice 8 : Équation de Poisson

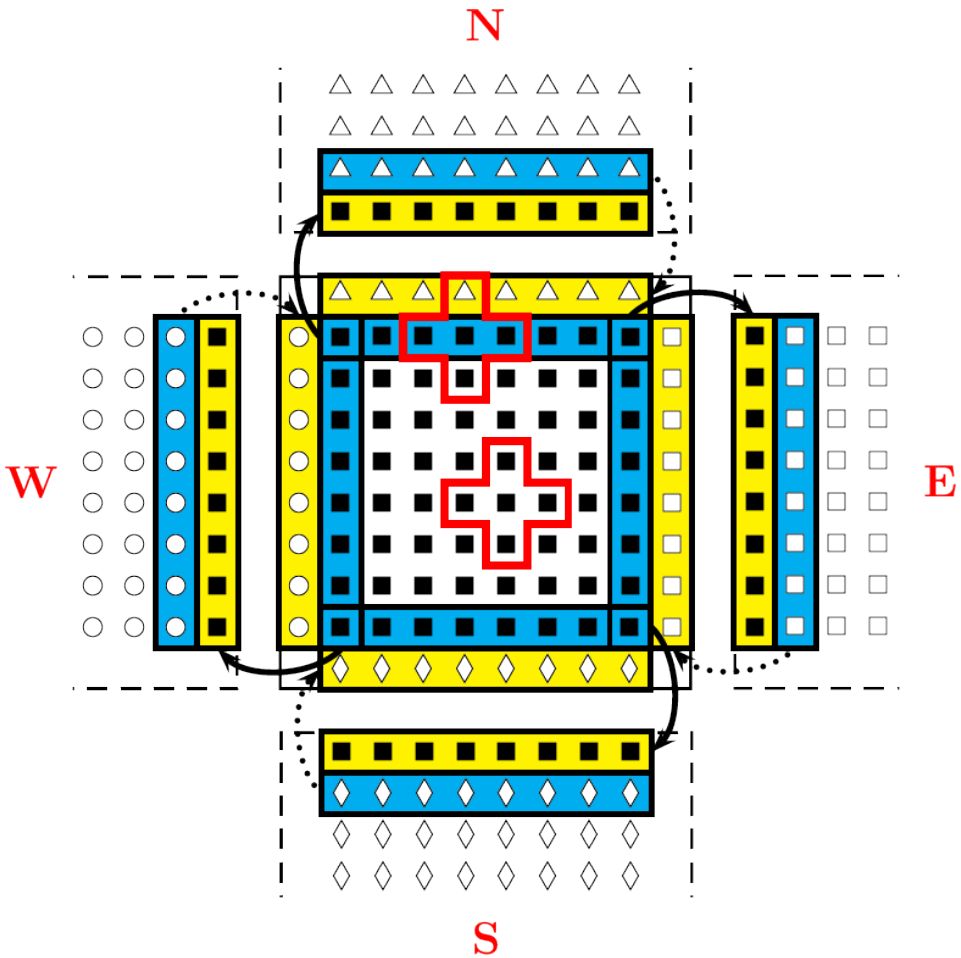


Figure 65 – Échange de points aux interfaces

Travaux pratiques MPI – Exercice 8 : Équation de Poisson

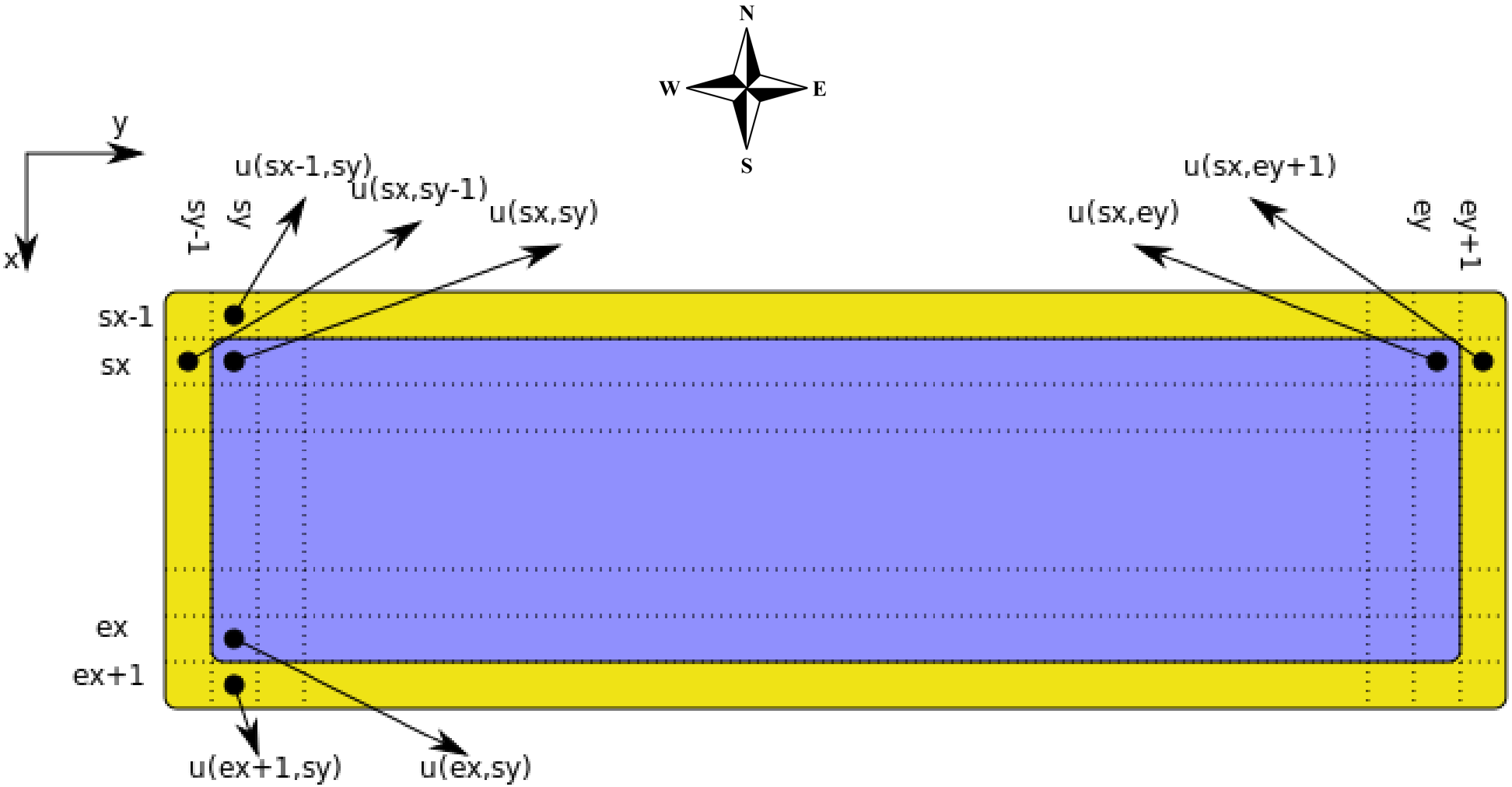


Figure 66 – Numérotation des points dans les différents sous-domaines

Travaux pratiques MPI – Exercice 8 : Équation de Poisson

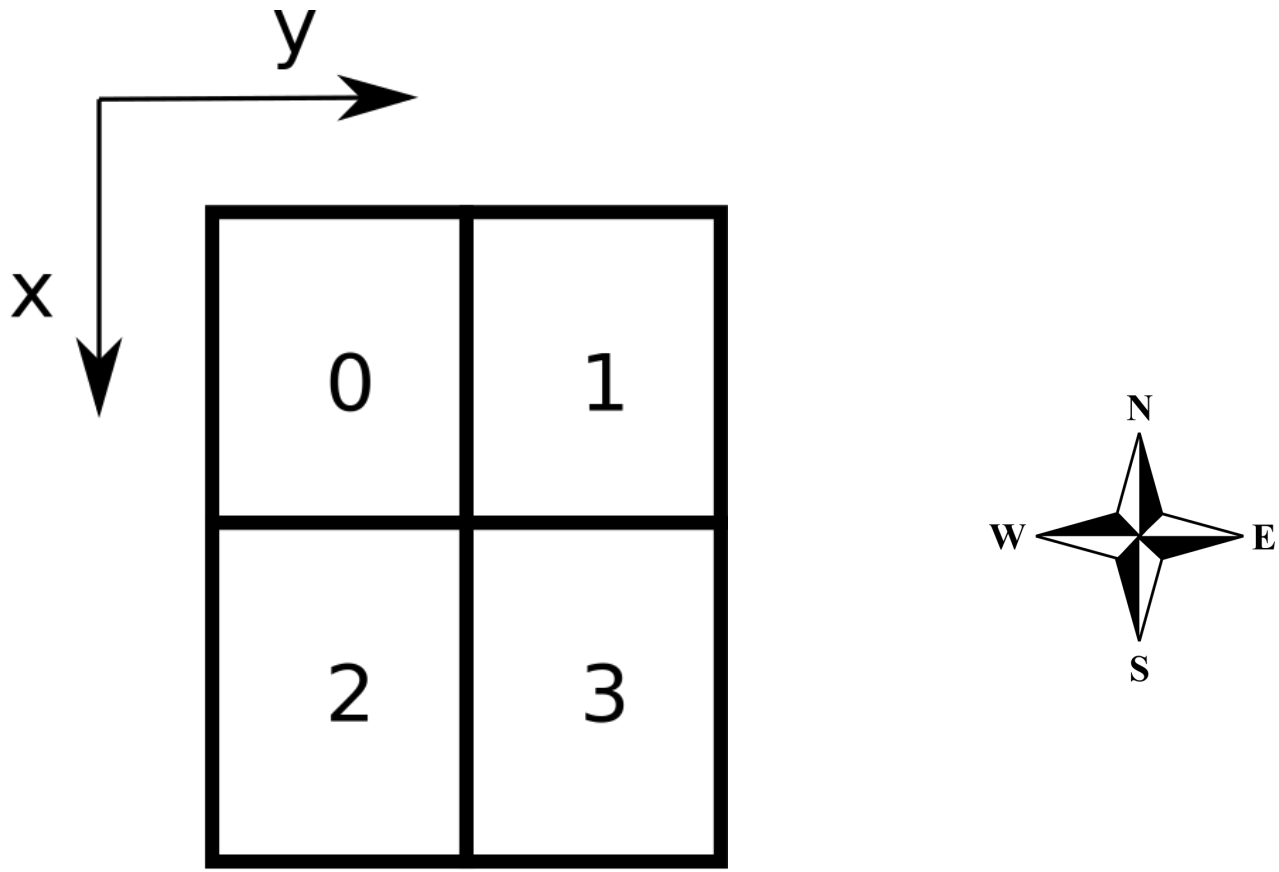


Figure 67 – Rang correspondant aux différents sous-domaines

Travaux pratiques MPI – Exercice 8 : Équation de Poisson

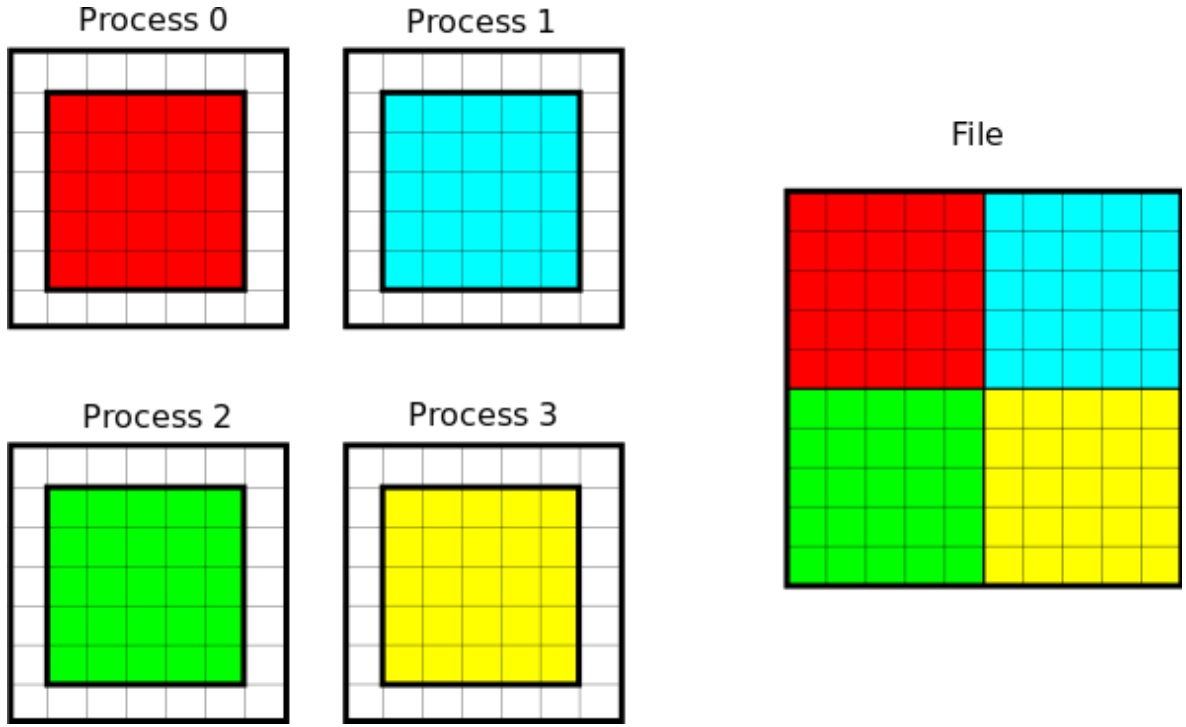


Figure 68 – Ecriture de la matrice u globale dans un fichier

Il s'agit de définir :

- Une vue, pour ne voir dans le fichier que la partie de la matrice u globale que l'on possède ;
- Un type afin d'écrire la matrice u locale (sans les cellules fantômes) ;
- Appliquer la vue au fichier ;
- Faire l'écriture en une fois.

Travaux pratiques MPI – Exercice 8 : Équation de Poisson

- initialiser l'environnement MPI ;
- créer la topologie cartésienne 2D ;
- déterminer les indices de tableau pour chaque sous-domaine ;
- déterminer les 4 processus voisins d'un processus traitant un sous-domaine donné ;
- créer deux types dérivés *type_ligne* et *type_colonne* ;
- échanger les valeurs aux interfaces avec les autres sous-domaines ;
- calculer l'erreur globale. Lorsque l'erreur globale sera inférieure à une valeur donnée (précision machine par exemple), alors on considérera qu'on a atteint la solution ;
- reformer la matrice u globale (identique à celle obtenue avec la version monoprocesseur) dans un fichier `donnees.dat`.

Travaux pratiques MPI – Exercice 8 : Équation de Poisson

- Un squelette de la version parallèle est proposé : il s'agit d'un programme principal (`poisson.f90`) et de plusieurs sous-programmes. Les modifications sont à effectuer dans le fichier `parallel.f90`.
- Pour compiler utilisez la commande make, pour exécuter le code utilisez la commande make exe. Pour vérifier les résultats, utilisez la commande make verification qui exécute un programme de relecture du fichier `donnees.dat` puis le compare avec la version monoprocesseur.