Plan

1. Historique
2. L'environnement
3. Communications point à point
4. Type de Données dérivés
5. Les communicateurs Intra et Inter
6. Opérations collectives
7. Topologies de processus
8. Gestion dynamique de processus

Cours basé sur

- Cours MPI de l'IDRIS
- Cours de G. Bosilca (UTK), L. Giraud (Inria),
- Livres MPI-2 + norme

Historique

Forum MPI (Message Passing Interface Forum)
la participation d'une quarantaine d'organisations

Version 1.0 : (juin 1994) Définition d'un ensemble de sous-programmes concernant la bibliothèque d'échanges de messages MPI.

Version 1.1 : juin 1995 - changements mineurs

Version 1.2 : en 1997 - 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

Approche statique qui s'oppose à PVM.

Historique

Version 2.0 : (Juillet 97), cette version apportait des compléments essentiels volontairement non intégrés dans MPI 1.0
- Gestion dynamique de processus ;
- Copies mémoire à mémoire ;
- Entrées-sorties parallèles.

Version 2.1 : juin 2008, avec seulement des clarifications dans MPI 2.0

Version 2.2 : septembre 2009, avec seulement de « petites » additions

Historique

Version 3.0 (septembre 2012)

- Meilleur support des applications actuelles et futures, notamment sur les machines massivement parallèles et many cores ;
- Principaux changements actuellement envisagés :
  - **Communications collectives non bloquantes** ;
  - Révision de l'implémentation des communications one way ;
  - Tolérance aux pannes ;
  - Bindings : Fortran (2003-**2008**), abandon du C++ ;
- Interfaçage d'outils externes (pour le débogage et les mesures de performance) ;

http://meetings.mpi-forum.org/MPI3.0mainpage.php
https://svn.mpi-forum.org/trac/mpi-forum-web/wiki

Quelques références

Message Passing Interface Forum, MPI : A Message-Passing Interface Standard, Version 2.2, High Performance Computing Center Stuttgart (HLRS), 2009
https://fs.hlrs.de/projects/par/mpi/mpi22/
http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

MPI - the complete reference. Volume 1, The MPI Core, second edition, Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra

MPI - the complete reference. Volume 2, The MPI-2 extensions, second edition, Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra

Documentations complémentaires :
http://www.mpi-forum.org/docs/
http://meetings.mpi-forum.org/MPI_3.0_main_page.php

Implémentations

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

1. MPICH2 : http://www.mcs.anl.gov/research/projects/mpich2/
2. Open MPI : http://www.open-mpi.org/

Implémentations constructeurs

1. Intel
2. SGI - MPT
3. CRAY
4. IBM (architecture power)

ENVIRONNEMENT

Un premier exemple en C

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
        int myrank, size;

        MPI_Init(NULL,NULL);
        MPI_Comm_size(MPI_COMM_WORLD,&size);
        MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

        printf("Hello world, je suis le processus %d parmi %d.\n",myrank,size);

        MPI_Finalize();
}
```


L'environnement

Pour compiler

mpicc hello.c -o hello

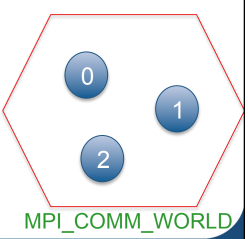
et exécution :

mpiexec –n 3 hello

Lance 3 instances du code

On obtient le résultat suivant :

Hello world, je suis le processus 1 parmi 3.
Hello world, je suis le processus 2 parmi 3.
Hello world, je suis le processus 0 parmi 3.



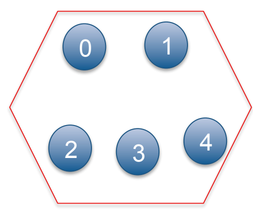
MPI_COMM_WORLD

Lancement

Single Process Multiple Data (SPMD) Model:
mpiexec [options] **<program>** [<args>]

Multiple Programme Multiple Data (MPMD) Model:
mpiexec [global_options] [local_options1] **<program1>** [<args1>] :
[local_options2] **<program2>** [<args2>]

mpiexec -n 2 hello : -n 3 hello
Hello world, je suis le processus 3 parmi 5.
Hello world, je suis le processus 4 parmi 5.
Hello world, je suis le processus 0 parmi 5.
Hello world, je suis le processus 1 parmi 5.
Hello world, je suis le processus 2 parmi 5.



MPI_COMM_WORLD

L'environnement

Les différentes fonctions :
MPI_Init : doit être appelée en tout début de programme.

```
int MPI_Init(int *argc, char ***argv)
```

Conseil de portabilité mettre NULL comme argument

MPI_FINALIZE() : doit être appelée en fin de programme par tous les processus, il désactive l'environnement MPI. Impossible d'appeler une autre fonction MPI même un MPI_Init.

```
int MPI_Finalize(void)
```

Adv. : Tous les communications doivent être terminées sinon on a un blocage.

L'environnement

MPI_COMM_WORLD : communicateur, objet opaque qui désigne un ensemble de processus pouvant communiquer entre eux. C'est le communicateur par défaut.

On peut l'interroger pour connaître le nombre de processus impliqués

```
int MPI_Comm_size(MPI_COMM_WORLD,&size)
```

et le rang du processus dans le groupe

```
int MPI_Comm_rank(MPI_COMM_WORLD,&rank)
```

Communications point à point

Description d'un message

Deux aspects
- Enveloppe
- Contenu

L'enveloppe du message est constituée
- de la source = rang du processus émetteur
- du destination = rang du processus récepteur
- de l'étiquette (tag) du message
- du nom du communicateur

Les données échangées sont typées (entiers, réels, types dérivés personnels)

Plusieurs modes de transfert faisant appel à des protocoles différents.

Principaux type de données

Type MPI	Type C
MPI_CHAR	char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_BYTE	
MPI_PACKED	Type hétérogènes

Les noms sont différents en Fortran (page 22 draft MPI 2.2).

Exemple 2 (1)

```
#include "mpi.h »
 int main( int argc, char **argv ) {
Int myrank, tag=99 ;
MPI_Status status;
…..
MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
if (myrank == 0)    {
    strcpy(message,"Hello, there");
    MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
} else if (myrank == 1)  {
    MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
    printf("received :%s:\n", message);
}
……
```


Exemple 2 (suite)

%mpiexec -n 2 firstmsg
%firstmsg
received : Hello, there

Le processeur 0 envoie une chaine de caractères au processeur 1.

Quelques remarques

- Le type des données du message doit être le même des deux côtés ;
- Le tag aussi.

Send & Receive

Communications explicites (FIFO par paire et par communicateur)
Déplacer les données d'un processeur à un autre.

- Les données sont décrites par un type, un nombre et un emplacement mémoire
- Le processus de destination par le rang dans le communicateur
- Le matching est basé sur le tag

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

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


Communications bloquantes

Le processus est bloqué dans la fonction MPI jusqu'à ce que

- Côté réception : les données distantes aient été copiée dans le buffer de réception.
- Côté émetteur : le buffer d'envoie puisse être modifié par l'utilisateur sans impacter le transfert du message.

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

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

Communications bloquantes

Type de donnée : Basic ou complexe

MPI_TYPE_CONTIGUOUS, MPI_TYPE_VECTOR, MPI_TYPE_INDEXES

MPI_ TYPE_STRUCT, …

Buffer

- doit être du même type que les données émises
- Taille >= taille du message sinon il y a troncature du message

  code d'erreur MPI_ERR_TRUNCATE

Jokers pour les tags et sources

MPI_ANY_SOURCE et MPI_ANY_TAG

Communications bloquantes

MPI_Status est une structure C contenant 3 champs

- MPI_SOURCE
- MPI_TAG
- MPI_ERROR (non significatif avec MPI_Send)

MPI_SUCCESS permet de tester le code retour d'une fonction MPI

Pour avoir la taille exacte du message on interroge la variable status

MPI_Get_count(&status, datatype, &count)

Si on ne veut pas d'information sur le status on utilise dans MPI_Recv

MPI_STATUS_IGNORE

Exemple 2

```
#include "mpi.h »
 int main( int argc, char **argv ) {
Int myrank, tag=99 ;
MPI_Status status;
…..
MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
if (myrank == 0)    {
    strcpy(message,"Hello, there");
    MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
} else if (myrank == 1)  {
    MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
    printf("received :%s:\n", message);
}
……
```


Exemple 2 (suite)

Code récepteur (rang=1)

```
MPI_Recv(message, 20, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG,
        MPI_COMM_WORLD, &status);
printf("status : \n    MPI_SOURCE: %d\n    MPI_TAG: %d\n    MPI_ERROR: %d\n",
        status.MPI_SOURCE, status.MPI_TAG,status.MPI_ERROR);
MPI_Get_count( &status,MPI_CHAR,&count) ;
printf("Message size: %d\n ",count);
```

```
received :Hello, there:
status :
   MPI_SOURCE: 0        ← L'émetteur
   MPI_TAG: 99          ← le tag
   MPI_ERROR: 0
Message size: 13        ← la taille exacte du message
```

Commentaires (1)

1. Caractère asymétrique des communications
   - Réception : accepte une communication de n'importe qui.
   - Émetteur : doit préciser le destinataire

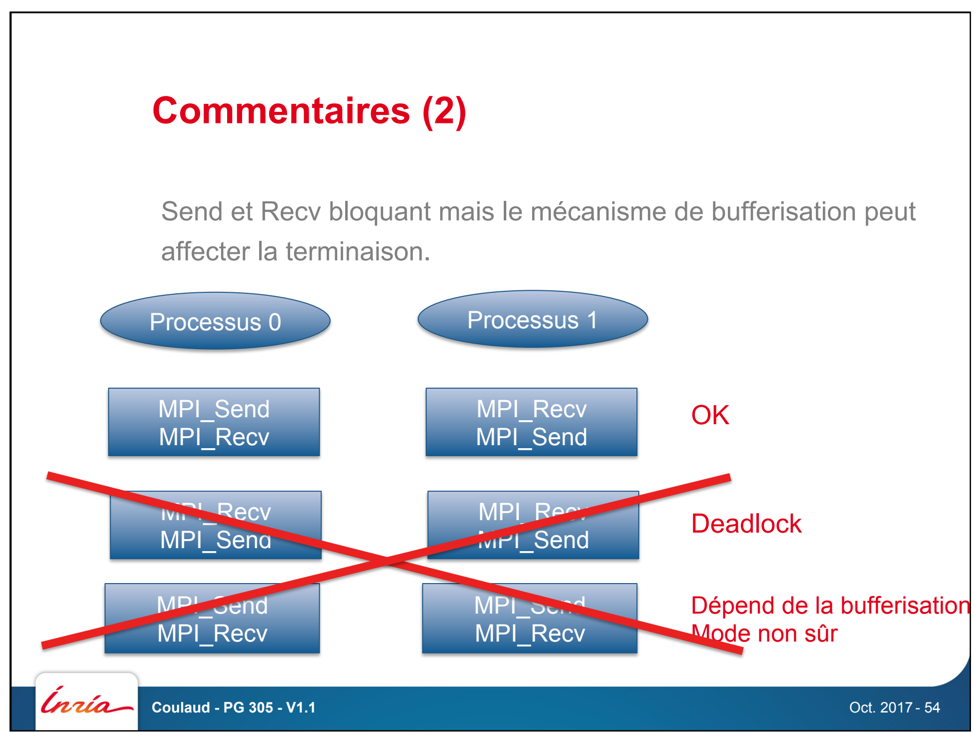
   → mécanisme de communication par Push

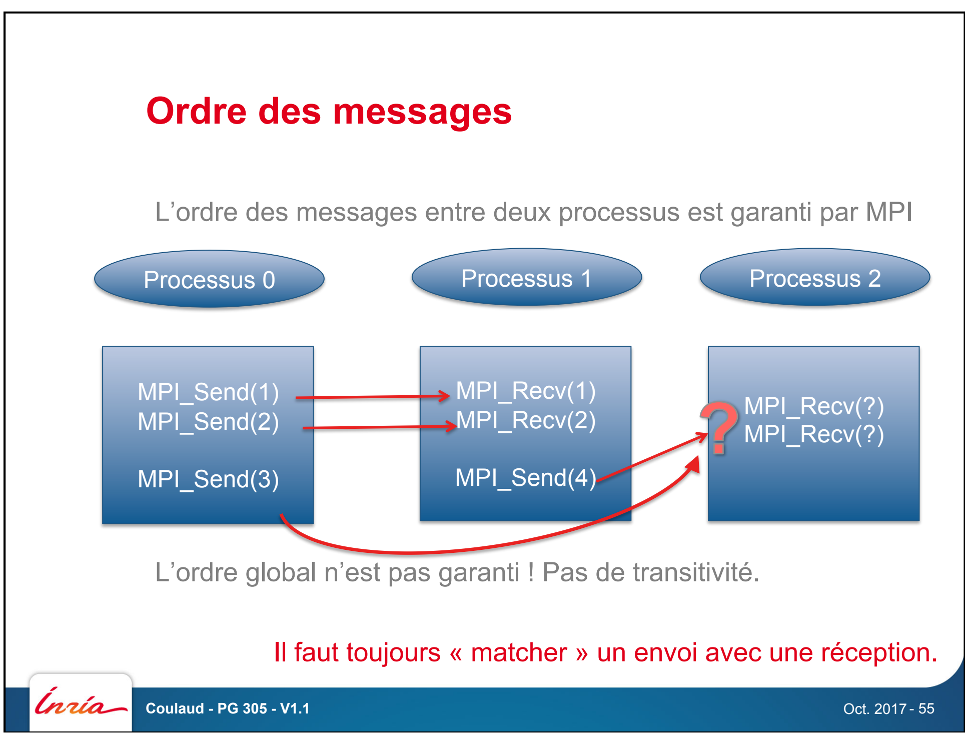
   (Pull dans l'autre sens)

2. Éviter de s'envoyer un message source = destinataire

Commentaires (2)

Send et Recv bloquant mais le mécanisme de bufferisation peut affecter la terminaison.





Quelques règles

Progression :

Pas de progressions garanties sauf à l'intérieur des appels MPI.

Exemple :

```
if (myrank == 0){
    MPI_Bsend(buff1, 1000, MPI_INT, 1, tag1, MPI_COMM_WORLD);
    MPI_Send(buff2, 1000, MPI_INT, 1, tag2, MPI_COMM_WORLD);}
else if (myrank == 1) {
    MPI_Recv(buff2, 1000, MPI_INT, 0, tag2, MPI_COMM_WORLD, &status);
    MPI_Recv(buff1, 1000, MPI_INT, 0, tag1, MPI_COMM_WORLD, &status)}
```

Équité : pas de garantie dans la gestion des communications. Toutefois une approche « best effort » est implémentée dans MPI.

Quelques définitions

Local : ne nécessite aucune communication.
Par ex. pour générer un objet MPI local.

Non-local : nécessite l'exécution d'une procédure MPI sur un autre processus.
Par ex. un envoi de message (nécessite une réception).

Quelques définitions (2)

La plupart des opérations MPI peuvent être utilisées en mode bloquant ou non bloquant.

Bloquant : à la fin d'un tel appel, le programme appelant peut réutiliser les ressources utilisées dans l'appel: par ex. un buffer.
Un envoi bloquant peut être :

- synchrone. La communication est établie et le message est effectivement arrivé à destination. (poignée de main)
- asynchrone. Si un buffer est utilisé pour stocker les données avant l'envoi.

Une réception bloquante retourne uniquement si les données sont reçues et prêtes à l'emploi.

Quelques définitions (3)

Non-bloquant : la primitive rend **immédiatement** la main. Elle n’attend pas la terminaison de la communication.

- La bibliothèque MPI réalise l’opération quand c’est possible.
- La modification du buffer est risquée sans plus d’information sur la progression de la communication.
- Très utile, car l'utilisateur peut alors exécuter d’autres codes pendant la progression de la communication. Recouvrement de la communication par des calculs → gain de performance.

Les modes de communication

4 modes de communication pour l’émission :

Standard : communication bloquante ou non bloquante. Retour au programme après terminaison. Bufférisé ou non au choix de l’implémentation.

Buffered. Il est à la charge de l’utilisateur d’effectuer une recopie temporaire du message. L’envoi se termine lorsque la recopie est achevée. L’envoi est découplé de la réception.

Synchronous : synchronise les processus d’envoi et de réception. L’envoi du message est terminé si la réception est postée et la lecture terminée.

Les modes de communication

Ready : L'envoi du message ne peut commencer que si la réception a été postée auparavant.

Mode intéressant pour les applications client-server.

Standard	Buffered	Synchronous	Ready
MPI_Send	MPI_Bsend	MPI_Ssend	MPI_Rsend

Il existe des variantes syntaxiques,

MPI_SENDRECV et MPI_SENDRECV_REPLACE

qui enchaînent un envoi vers un processeur A et une réception d'un processeur B. (pages 73-75 du draft 2.2)

Attention à la sémantique de l'envoi bloquant

Les protocoles de communications

Deux protocoles

- Eager pour des petits messages
- Rendez-vous

Eager

- Envoie direct

P0 P1

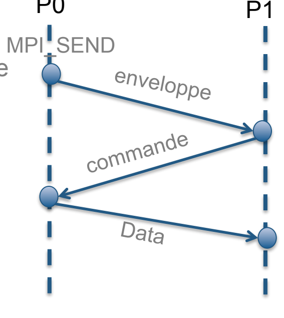
MPI_SEND

enveloppe

Data

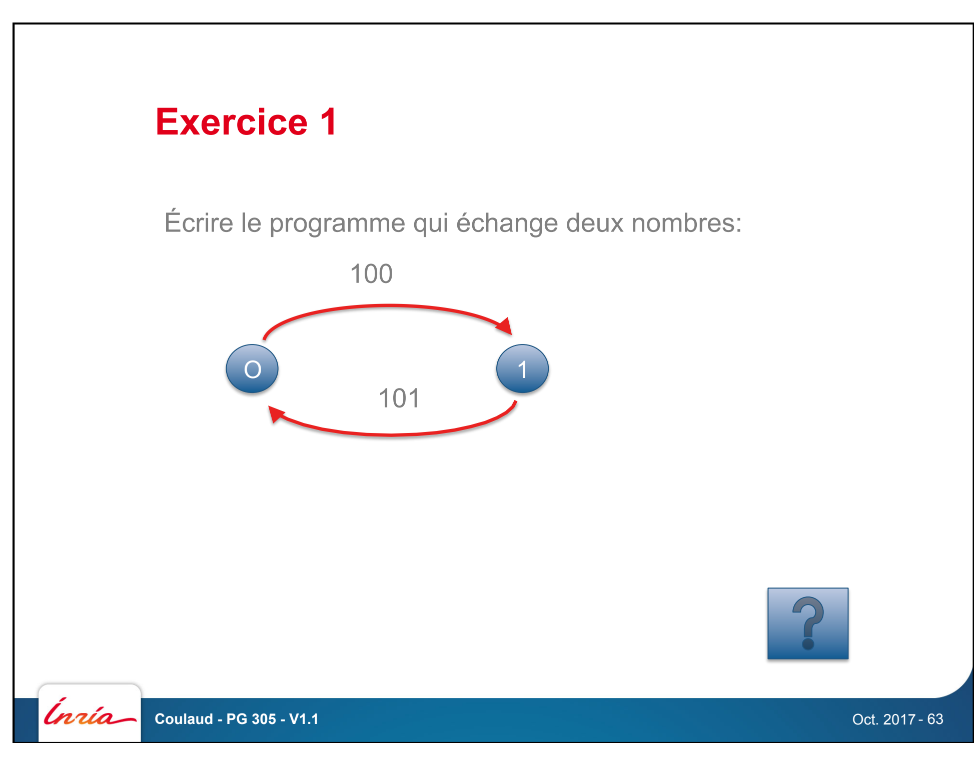
Rendez-vous

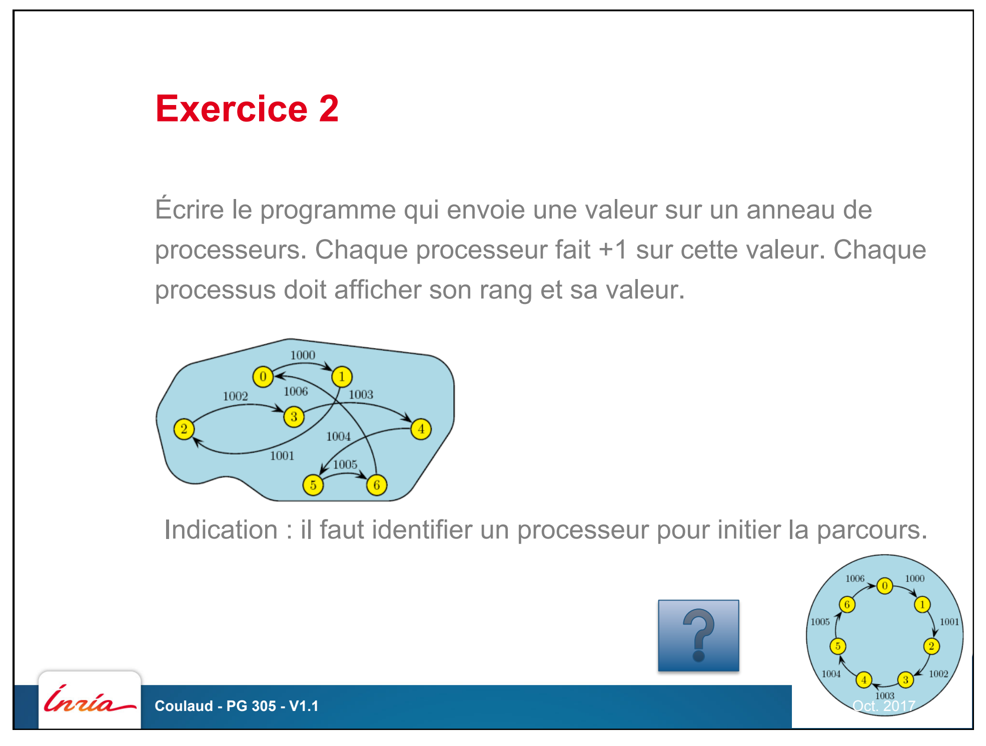
1. Notification d'un message
   P1 prépare la réception
2. P1 prévient qu'il est prêt
3. P0 envoie les données



Ex : Evaluer le coût de la communication dans les deux protocoles.

$T_{com} = \alpha + \beta$ m avec m la taille du message, α la latence, $1/\beta$ le débit


Exercice 1
Écrire le programme qui échange deux nombres:
100
O
1
101
Coulaud - PG 305 - V1.1
Oct. 2017 - 63


Exercice 2
Écrire le programme qui envoie une valeur sur un anneau de processeurs. Chaque processeur fait +1 sur cette valeur. Chaque processus doit afficher son rang et sa valeur.
Indication : il faut identifier un processeur pour initier la parcours.
Coulaud - PG 305 - V1.1
Oct. 2017

Communications non bloquantes

Le processus sort de la fonction dès que possible et avant qu'un transfert ne commence.

Tous les modes de communication sont supportés

standard, bufférisé, synchrone, prêt

Il faut tester le statut de la terminaison.

```
int MPI_Isend( void* buf, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm,
               MPI_Request *request )
int MPI_Irecv( void* buf, int count, MPI_Datatype datatype,
               int source, int tag, MPI_Comm comm,
               MPI_Request *request )
```

Un objet request est utilisé pour identifier la requête et tester la terminaison.

Communications non bloquantes

Avantages

- Permet d'éviter des inter blocages ;
- Permet de recouvrir les communications par du calcul.

Inconvénients

- Augmente la complexité des codes ;
- Difficile à déboguer et à valider.

Quelques fonctions pour

- Synchroniser les processus (MPI_Wait, …) ;
- Vérifier si une requête est terminée (MPI_Test,…) ;
- Contrôler avant la réception si un message est arrivé (MPI_Probe,…)
- Supprimer une communication (MPI_Cancel) (À ÉVITER)

Exemple 3

```
MPI_Status status;
MPI_Request request;

if (myrank == 0){
   MPI_Isend(buff1, msize, MPI_FLOAT, 1, tag1, MPI_COMM_WORLD,&request);
          ....  des calculs pour masquer la latence
    MPI_Wait(&request,&status) ;
}
else if (myrank == 1) /* code for process one */ {
   MPI_Irecv(buff1, msize, MPI_FLOAT, 0, tag1, MPI_COMM_WORLD, &request);
    ....  des calculs pour masquer la latence
    MPI_Wait(&request,&status) ;
}
```


Terminaison de la communication

Terminaison simple
- la terminaison d'une **opération d'envoi** indique que l'envoyeur peut updater le buffer. Cela ne dit pas que le message est reçu.
- la terminaison d'une **opération de réception** indique que le buffer contient le message reçu.

```
int MPI_Test( MPI_Request *request,   int *flag, MPI_Status *status)

int MPI_Wait( MPI_Request *request, MPI_Status *status )
```

Terminaison de la communication

Terminaisons multiples (ANY)

Un appel à MPI_WAITANY or MPI_TESTANY peut être utilisé pour attendre la fin d'une communication parmi plusieurs.

Index : le numéro d'une requête terminée.

```
int MPI_Waitany( int count, MPI_Request *array_of_requests,
                 int *index, MPI_Status *status )

int MPI_Testany( int count, MPI_Request *array_of_requests,
                 int *index, int *flag, MPI_Status *status )
```


Terminaison de la communication

Terminaison multiple (SOME)

Un appel à MPI_WAITSOME or MPI_TESTSOME peut être utilisé pour attendre la terminaison d'au moins une requête dans la liste des requêtes.

outcome nombre de requêtes terminées.

Plus performant que ANY.

```
int MPI_Waitsome( int incount, MPI_Request *array_of_requests,
                  int *outcount, int *array_of_indices,
                  MPI_Status *array_of_statuses )
int MPI_Testsome( int incount, MPI_Request *array_of_requests,
                  int *outcount, int *array_of_indices,
                  MPI_Status *array_of_statuses )
```

Terminaison de la communication

Terminaison multiple (ALL)

- Un appel à MPI_WAITALL or MPI_TESTALL peut être utilisé pour tester la terminaison de **toutes** les opérations.

```
int MPI_Waitall( int count, MPI_Request *array_of_requests,
                 MPI_Status *array_of_statuses )

int MPI_Testall( int count, MPI_Request *array_of_requests,
                 int *flag, MPI_Status *array_of_statuses )
```

Ordre et progression

Les communications non bloquantes préservent l'ordre des appels comme pour les communications bloquantes.
Ordre des appels qui initie les communications MPI

Communications Persistantes

Une communication avec la même liste d'argument exécutée à l'intérieur d'une boucle d'un calcul parallèle.

Autorise aux implémentations de MPI d'optimiser les transferts de données.

Tous les modes de communication (buffered, synchronous and ready) peuvent être utilisés

```
int MPI_Send_init( void* buf, int count, MPI_Datatype datatype,
                   int dest, int tag, MPI_Comm comm,
                   MPI_Request *request )
int MPI_Recv_init( void* buf, int count, MPI_Datatype datatype,
                   int source, int tag, MPI_Comm comm,
                   MPI_Request *request )
```

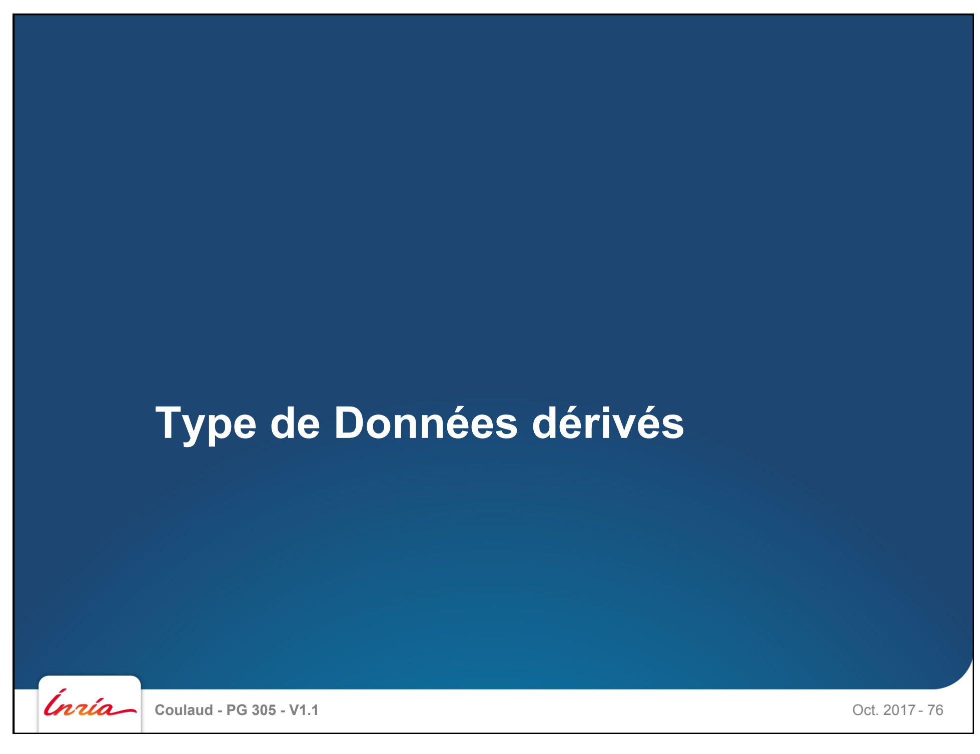

Communications Persistantes

4 étapes

1. Initialisation MPI_Send_init : enregistre une requête persistante.
2. Démarre la communication
   MPI_Start(requête) MPI_Startall(…)
3. Attente de la terminaison de la communication MPI_Wait
   MPI_Wait(requête)
4. Destruction des objets de la requête persistante
   MPI_Request_free(requête)

Communications Persistantes (3)

```
MPI_Ssend_init(...,&s_request);
MPI_Recv_init(...,&r_request);                    Initialisation
----------------------------------------------------------------
Loop:
----------------------------------------------------------------
   MPI_Start(&s_request);
   MPI_Start(&r_request);                         Initie la comm
----------------------------------------------------------------
.           ...... // calculs
----------------------------------------------------------------
     MPI_Wait(&s_request,...);
     MPI_Wait(&r_request,...);                    Attente fin de comm
----------------------------------------------------------------
EndLoop
----------------------------------------------------------------
 MPI_Request_free(&s_request);
 MPI_Request_free(&r_request);                    Libère
```

Type de données MPI

Représentation abstraite des types de données sous-jacents

Type d'objet : MPI_Datatype

Types prédéfinis pour les types de base

- C: MPI_INT, MPI_FLOAT, MPI_DOUBLE
- Fortran: MPI_INTEGER, MPI_REAL
- C++: MPI::BOOL

Types dérivés définis par l'utilisateur

- E.g., arbitraire / structure C

Interopérabilité entre plate-forme

Différentes représentations des données

- Longueur 32 vs. 64 bits
- little-endian versus big-endian

Problèmes

- Pas de standard sur la longueur des données dans les langages de programmation (C/C++/Fortran)
- Pas de représentation standard pour les flottants :
  - Standard IEEE 754 Floating Point Numbers
    Subnormals, infinities, NANs …
  - Même représentation mais différentes longueurs

Coté Performance

Ancienne approche

- Copier manuellement les données dans un buffer ;
- Utilise manuellement MPI_PACK and MPI_UNPACK.

Nouvelle Façon

- Avoir confiance dans les bibliothèques modernes ;
- Type MPI très performant.

Types de données MPI Datatypes

MPI utilise des « datatypes » pour :

- Représenter et transférer efficacement des données ;
- Minimiser l'utilisation de la mémoire.

Même entre des systèmes hétérogènes

- Utilisable dans la plupart des communications (MPI_SEND, MPI_RECV, etc.) ;
- Opérations sur les fichiers.

MPI contient un grand nombre de types prédéfinis

Quelques types predéfinis

MPI_Datatype	C datatype	Fortran datatype
MPI_CHAR	signed char	CHARACTER
MPI_SHORT	signed short int	INTEGER(2)
MPI_INT	signed int	INTEGER
MPI_LONG	signed long int	
MPI_UNSIGNED_CHAR	unsigned char	
MPI_UNSIGNED_SHORT	unsigned short	
MPI_UNSIGNED	unsigned int	
MPI_UNSIGNED_LONG	unsigned long int	
MPI_FLOAT	float	REAL(4)
MPI_DOUBLE	double	REAL(8)
MPI_LONG_DOUBLE	long double	DOUBLE PRECISION*8

Datatype Matching

Deux conditions à satisfaire :

- Chaque type de données dans le buffer d'envoi/réception doit correspondre au type spécifié à l'envoi/réception ;
- Type spécifié par l'opération d'envoi doit correspondre au type spécifié à la réception.

Solution :

- Faire correspondre le type avec le langage ;
- Faire correspondre les types à l'envoi et à la réception.

Conversion des types

"Données envoyées = données reçues"

Deux types de conversions:

1. Changement de représentation : change la représentation binaire (e.g., hex floating point to IEEE floating point)
2. Conversion de Type :
   Convertir deux types (e.g., int to float)

➔ Seulement le changement de représentation est autorisé.

Conversion des types

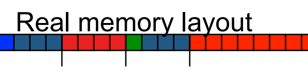
```
if( my_rank == root )
   MPI_Send( ai, 1, MPI_INT, … )
else
   MPI_Recv( ai, 1, MPI_INT, … )
```

```
if( my_rank == root )
   MPI_Send( ai, 1, MPI_INT, … )
else
   MPI_Recv( af, 1, MPI_FLOAT, … )
```

Agencement mémoire

Comment décrire un agencement mémoire ?

```
struct {
  char c1;
  int i;
  char c2;
  double d;
}
```



Liste d'adresses

<pointer to memory, length>

<baseaddr c1, 1>,<addr_of_i, 4>, <addr_of_c2, 1>, <addr_of_d, 8>

- Gaspillage mémoire ;
- Non portable …

Utilise des déplacements par rapport à une adresse de base

<displacement, length>

<0, 1>, <4, 4>, <8, 1>, <12, 8>

- plus efficace en mémoire ;
- presque portable.

Types utilisateurs ou types dérivés

Type dérivé est un objet opaque crée par l'utilisateur qui spécifie

- Une suite de types de base
- Une suite de déplacements en octet (<0 ,>0, ordre quelconque)

Type signature

- Utilisé pour le « matching » des messages

$\{ type_0, type_1, \ldots, type_n \}$

Type map

- Utilisé pour les opérations locales

$\{ (type_0, disp_0), (type_1, disp_1), \ldots, (type_n, disp_n) \}$

Buffer de communication = Type map + adresse de départ (*buf*)

→ $i^{ème}$ entrée = *buf* + $disp_i$ et de type $type_i$

Des fonctions prévoient des modèles contigus, déplacement à pas constants, à pas variables, …

Marqueurs Lower-Bound et Upper-Bound

Typemap = { ($type_0$, $disp_0$), …, ($type_n$, $disp_n$) }

Limite inférieure (Lower bound) est le plus petit déplacement dans le type. Adresse relative du premier byte des données.

$$Lower_bound(Typemap) = Min_j \; disp_j$$

En général = 0

Limite supérieure (Upper bound) est l'adresse relative du dernier byte des données dans le type

$$Upper_bound(Typemap) = Max_j \; disp_j + sizeof(type_j) + align$$

Longueur (Extent) du type

Extent(typemap) = Upper_bound(Typemap) - Lower_bound(Typemap)

Manipulation des datatypes

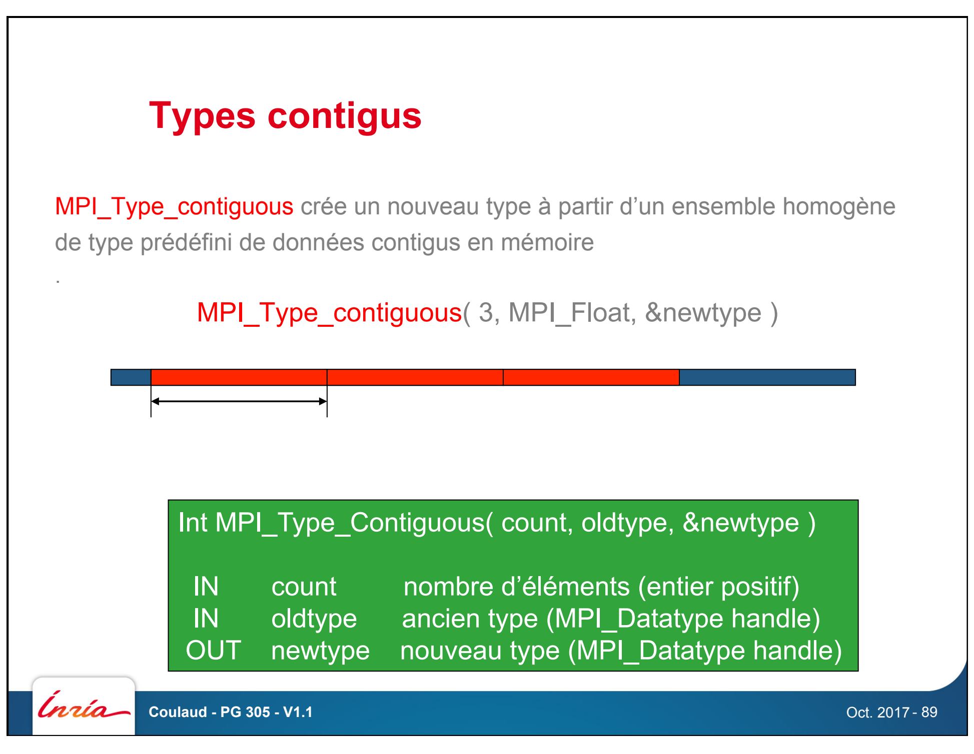
MPI impose que tous les types de données utilisées pour les communications et les opérations sur les fichiers doivent être enregistrés

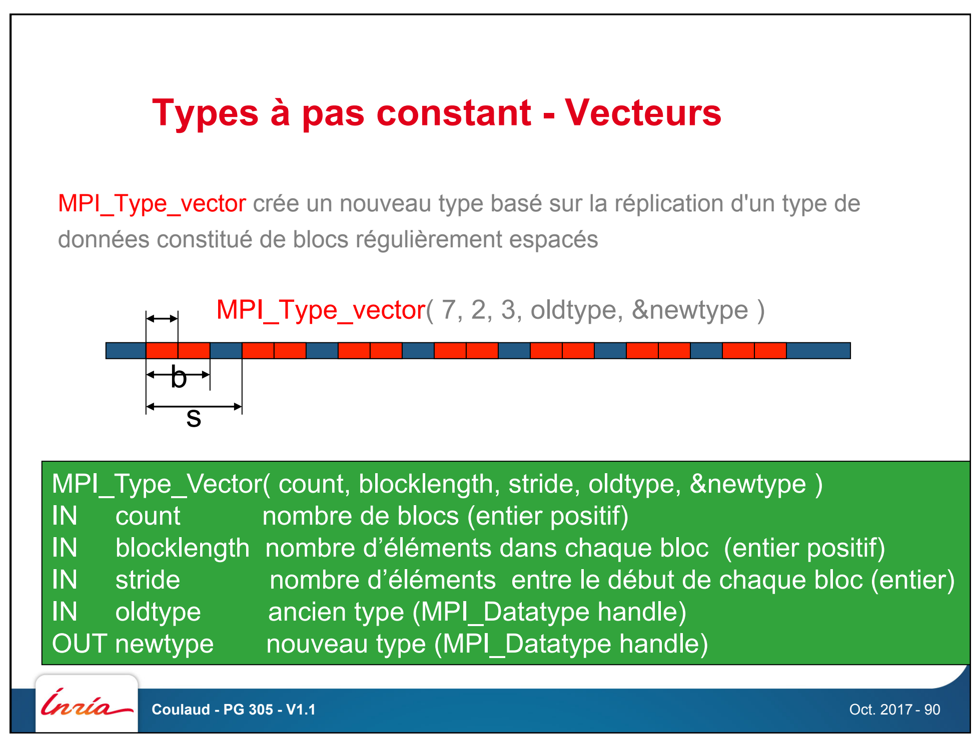
Permet d'optimiser la représentation des types

```
int MPI_Type_commit( MPI_Datatype* )
```

```
int MPI_Type_free( MPI_Datatype* )
```

Les types utilisés durant des étapes intermédiaires et jamais utilisés dans les communications n'ont pas besoin d'être enregistrés.


Types contigus
MPI_Type_contiguous crée un nouveau type à partir d'un ensemble homogène de type prédéfini de données contigus en mémoire
.
MPI_Type_contiguous(3, MPI_Float, &newtype)
Int MPI_Type_Contiguous(count, oldtype, &newtype)
IN count nombre d'éléments (entier positif)
IN oldtype ancien type (MPI_Datatype handle)
OUT newtype nouveau type (MPI_Datatype handle)
Coulaud - PG 305 - V1.1
Oct. 2017 - 89


Types à pas constant - Vecteurs
MPI_Type_vector crée un nouveau type basé sur la réplication d'un type de données constitué de blocs régulièrement espacés
MPI_Type_vector(7, 2, 3, oldtype, &newtype)
b
s
MPI_Type_Vector(count, blocklength, stride, oldtype, &newtype)
IN count nombre de blocs (entier positif)
IN blocklength nombre d'éléments dans chaque bloc (entier positif)
IN stride nombre d'éléments entre le début de chaque bloc (entier)
IN oldtype ancien type (MPI_Datatype handle)
OUT newtype nouveau type (MPI_Datatype handle)
Coulaud - PG 305 - V1.1
Oct. 2017 - 90

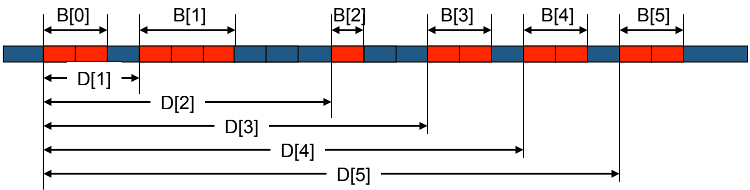
Types homogènes à pas variable

Réplication d'un type de données en une suite de blocs.
Chaque bloc peut
- Contenir un nombre différent d'éléments ;
- Avoir un espacement différent.

MPI_Type_Indexed (count, array_of_blocks, array_of_displs, oldtype,&newtype)

IN	count	nombre de blocs (entier positif)
IN	a_of_b	nombre d'éléments dans chaque bloc (tableau d'entiers)
IN	a_of_d	espacement entre chaque depuis le debut en unité de l'ancien type (tableau d'entiers)
IN	oldtype	ancien type (MPI_Datatype handle)
OUT	newtype	nouveau type (MPI_Datatype handle)

Types homogènes à pas variable



array_of_blocklengths[] = B[] = { 2, 3, 1, 2, 2, 2 }
array_of_displs[] = D[] = { 0, 3, 10, 13, 16, 19 }

MPI_Type_indexed(6, B, D, oldtype, newtype)

Vue hiérarchique des types

MPI_TYPE_CREATE_STRUCT

MPI_TYPE_[CREATE_H]INDEXED

MPI_TYPE_[CREATE_H]VECTOR

MPI_TYPE_CONTIGUOUS

MPI_REAL, MPI_INTEGER, MPI_LOGICAL

Étendre la description des types en autorisant des agencements mémoire plus complexes . Toutes les structures de données

- Ne s'inscrivent pas dans la forme des modèles communs ;
- Ne sont pas des compositions de type.

Les fonctions « H »

Si le déplacement entre deux éléments n'est pas un multiple de l'ancien type.

Au lieu de cela, le déplacement est en octets

- MPI_TYPE_CREATE_HVECTOR
- MPI_TYPE_CREATE_HINDEXED

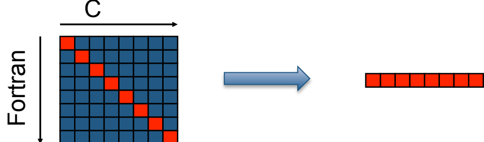
Autrement similaire à la variante sans « H »

Exercices

1) Extraire la diagonale d'un bloc

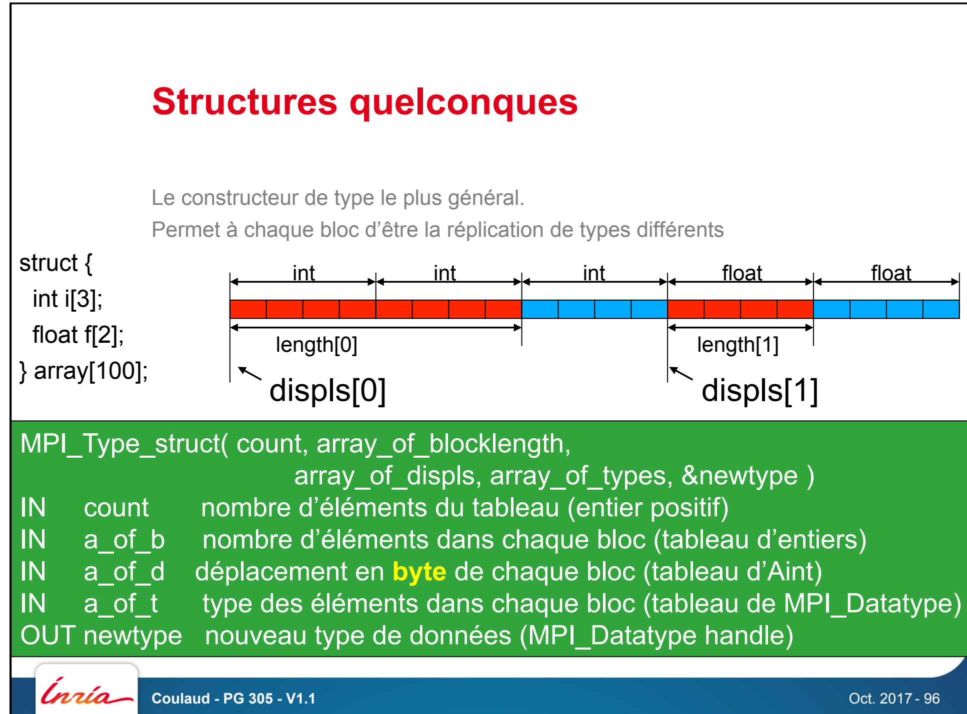
   Chaque processus possède un bloc NxN.

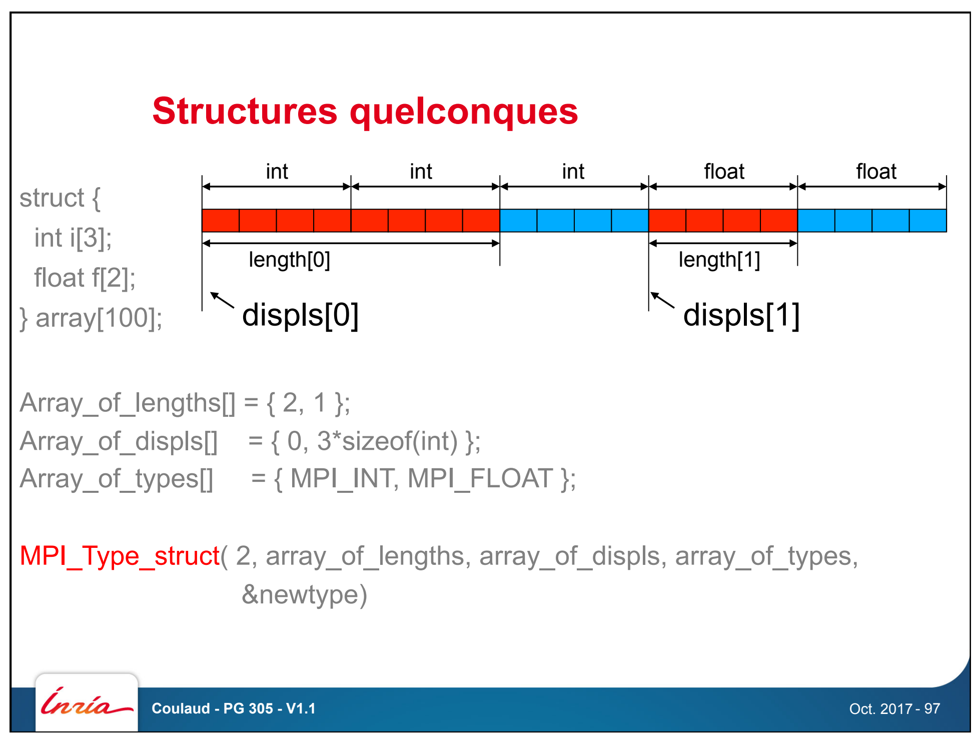
   a) Le processus 0 envoie directement la diagonale de son bloc A au processus 1 qui le stocke dans un tableau de dimension N.



   b) Le processus 1 envoie la diagonale du bloc B et le processus 0 la stocke directement dans son bloc A.

2) Créer un type décrivant une matrice triangulaire supérieure sans la diagonale.





Portable versus non portable

La portabilité est liée aux architectures hétérogènes

Constructeurs de types de données non portables :

- Tous les constructeurs basés sur des espacements en bytes ;
- Tous les constructeurs avec H<type>, MPI_Type_struct.

Limitations pour les types non portables

- Opérations écriture en mémoire « One sided operations » ;
- Opérations sur les I/O parallèles – MPI-I/O.

MPI_GET_ADDRESS

Disponible dans tous les langages pour calculer les espacements

- Nécessaire en Fortran
- En “&foo” → adresse de foo

NON PORTABLE

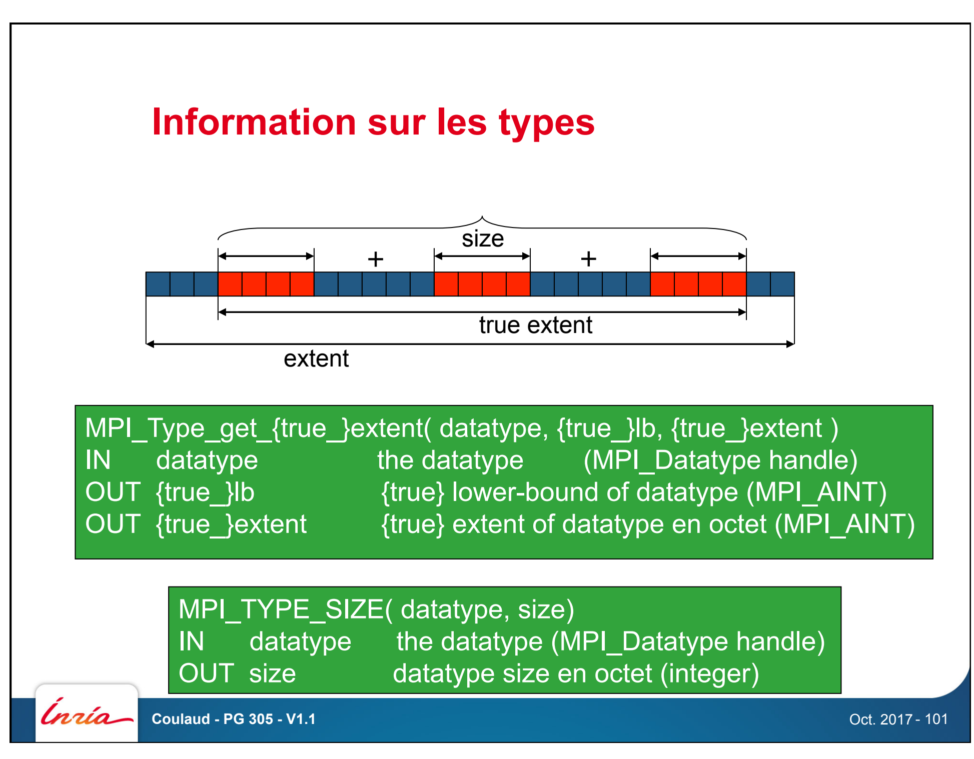
MPI_Get_address(location, address)
IN   location   location in the caller memory (void *)
OUT   address   address of location (address integer MPI_Aint)

Exemple

```
Struct PartStruct
{
  char    class;
  double  x[6];
  char[7]  name;
}
```

```
MPI_Datatype type[3] = { MPI_CHAR, MPI_DOUBLE, MPI_CHAR };
int blocklen[3]             = { 1, 6, 7 };
MPI_Aint i1,i2 ;

MPI_Get_address(&particle[0], &i1);
MPI_Get_address(&particle[0].class, &i2);  disp[0] = i2 - i1 ;
MPI_Get_address(&particle[0].x, &i2);        disp[1] = i2-  i1 ;
MPI_Get_address(&particle[0].name, &i2);  disp[2] = i2 - i1 ;
//
MPI_Type_struct(3, blocklen, disp, type, &Particletype);
```

type : 0 taille du bloc 1 disp 0.
type : 1 taille du bloc 6 disp 8.
type : 2 taille du bloc 7 disp 56.



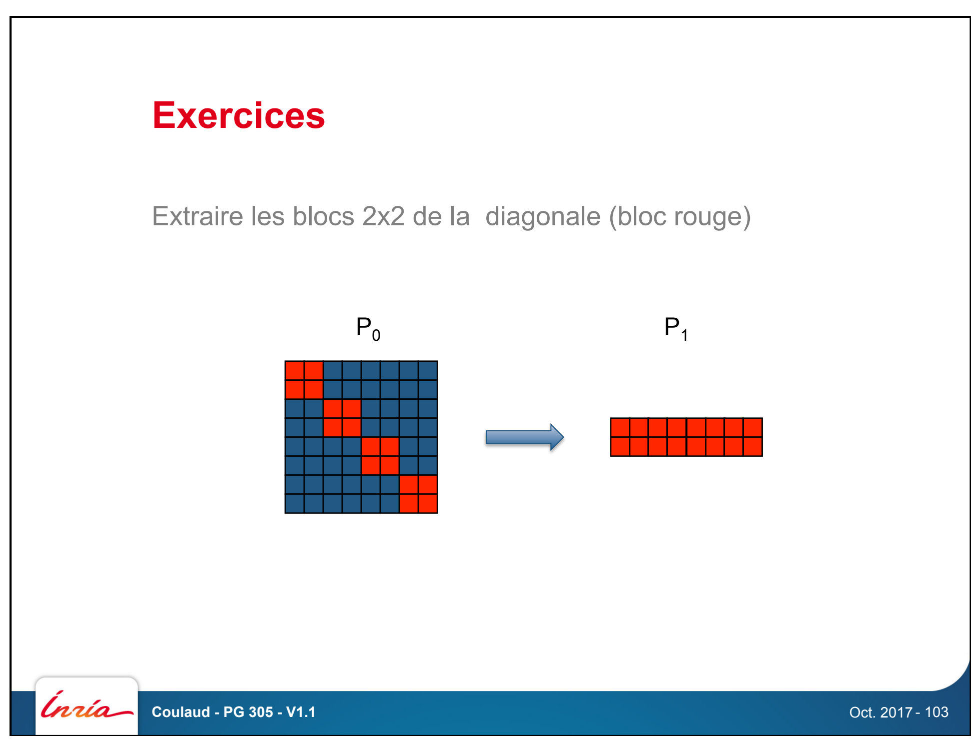
Exemple

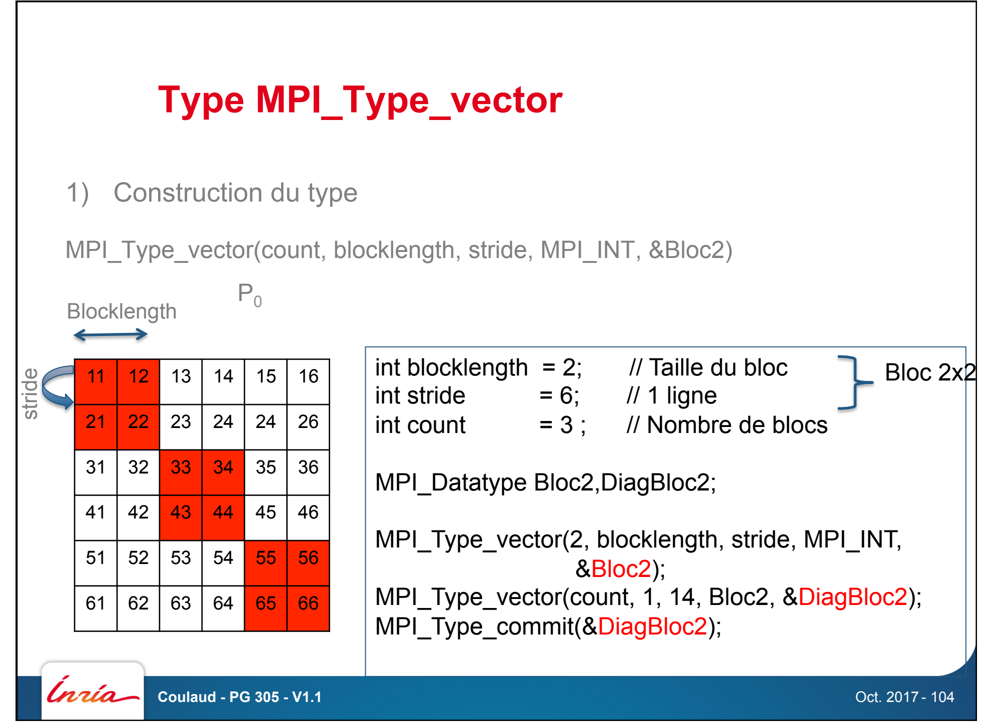
```
Struct PartStruct
{
  char    class;
  double  x[6];
  char[7]  name;
}
```

```
MPI_Datatype type[3] = { MPI_CHAR, MPI_DOUBLE, MPI_CHAR };
int blocklen[3]          = { 1, 6, 7 };
MPI_Aint i1,i2 ;

MPI_Get_address(&particle[0], &i1);
MPI_Get_address(&particle[0].class, &i2);  disp[0] = i2 - i1 ;
MPI_Get_address(&particle[0].x, &i2);       disp[1] = i2-i1 ;
MPI_Get_address(&particle[0].name, &i2); disp[2] = i2-i1 ;
 //
MPI_Type_struct(3, blocklen, disp, type, &Particletype);
```

type : 0 taille du bloc 1 disp 0.
type : 1 taille du bloc 6 disp 8.
type : 2 taille du bloc 7 disp 56.

type Particletype extent 64.
true type Particletype extent 63.


Exercices
Extraire les blocs 2x2 de la diagonale (bloc rouge)
P0
P1
Coulaud - PG 305 - V1.1
Oct. 2017 - 103


Type MPI_Type_vector
1) Construction du type
MPI_Type_vector(count, blocklength, stride, MPI_INT, &Bloc2)
P0
Blocklength
stride
11 12 13 14 15 16
21 22 23 24 24 26
31 32 33 34 35 36
41 42 43 44 45 46
51 52 53 54 55 56
61 62 63 64 65 66
int blocklength = 2; // Taille du bloc
int stride = 6; // 1 ligne
int count = 3 ; // Nombre de blocs
Bloc 2x2
MPI_Datatype Bloc2,DiagBloc2;
MPI_Type_vector(2, blocklength, stride, MPI_INT,
&Bloc2);
MPI_Type_vector(count, 1, 14, Bloc2, &DiagBloc2);
MPI_Type_commit(&DiagBloc2);
Coulaud - PG 305 - V1.1
Oct. 2017 - 104

matrix:

11 12 13 14 15 16
21 22 23 24 25 26
31 32 33 34 35 36
41 42 43 44 45 46
51 52 53 54 55 56
61 62 63 64 65 66

I am process 1

i receive the bloc 2x2 diagonal elements from proc 0

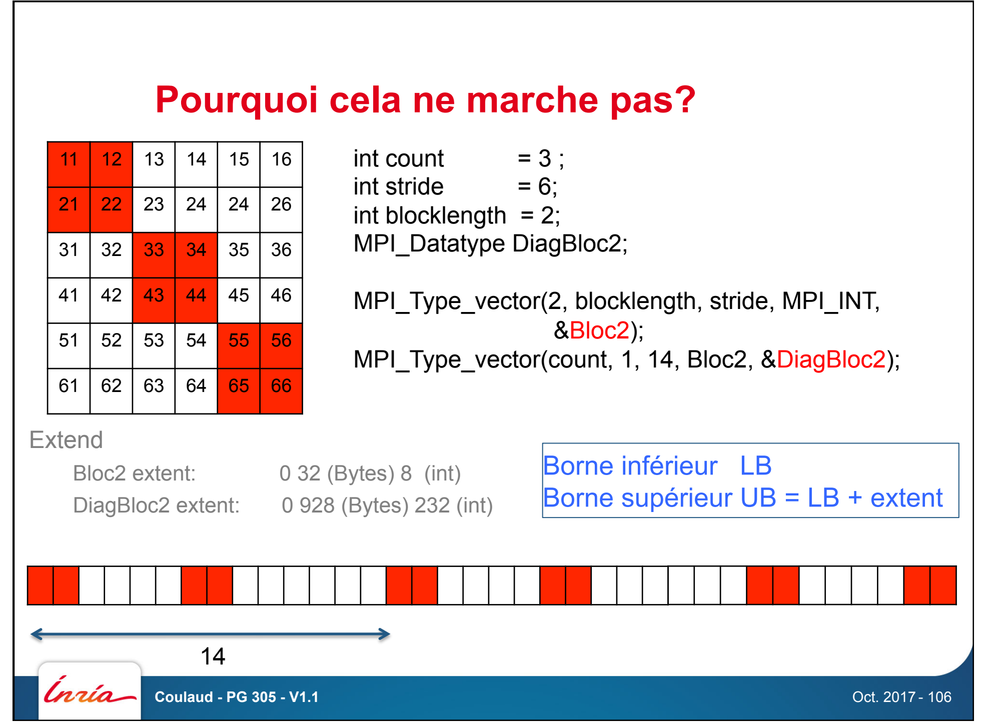
Blocs:

11 12 21 22 1360573467 32767 1360573519 32767 1360576117 32767 1360576194 32767

```
MPI_Type_vector(3, 1, 4, Bloc2, &DiagBloc2);
```

Blocs:

11 12 21 22 63 64 168 0 1581776168 32767 0 0

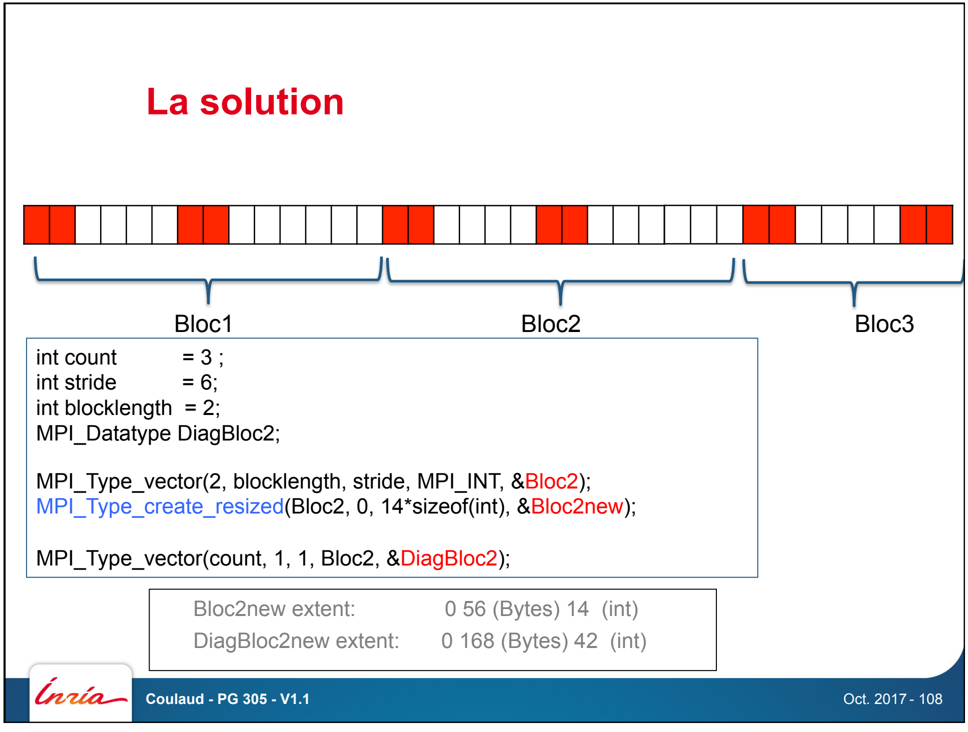


Redimensionner un type

MPI_Type_create_resized(oldtype, lb, extent, *newtype)

IN oldtype le datatype à redimensionner (MPI_Datatype)
IN lb borne inferieur lower du datatype (MPI_Aint)
IN extent extent du datatype en octet (MPI_AInt)
OUT newtype le nouveau (MPI_Datatype)

La solution




```
int count       = 3 ;
int stride      = 6;
int blocklength  = 2;
MPI_Datatype DiagBloc2;

MPI_Type_vector(2, blocklength, stride, MPI_INT, &Bloc2);
MPI_Type_create_resized(Bloc2, 0, 14*sizeof(int), &Bloc2new);

MPI_Type_vector(count, 1, 1, Bloc2, &DiagBloc2);
```

```
Bloc2new extent:        0 56 (Bytes) 14 (int)
DiagBloc2new extent:    0 168 (Bytes) 42 (int)
```

MPI_Recv (&blocs[0], 2*sizeMat, MPI_INT, 0, tag, MPI_COMM_WORLD, &status)

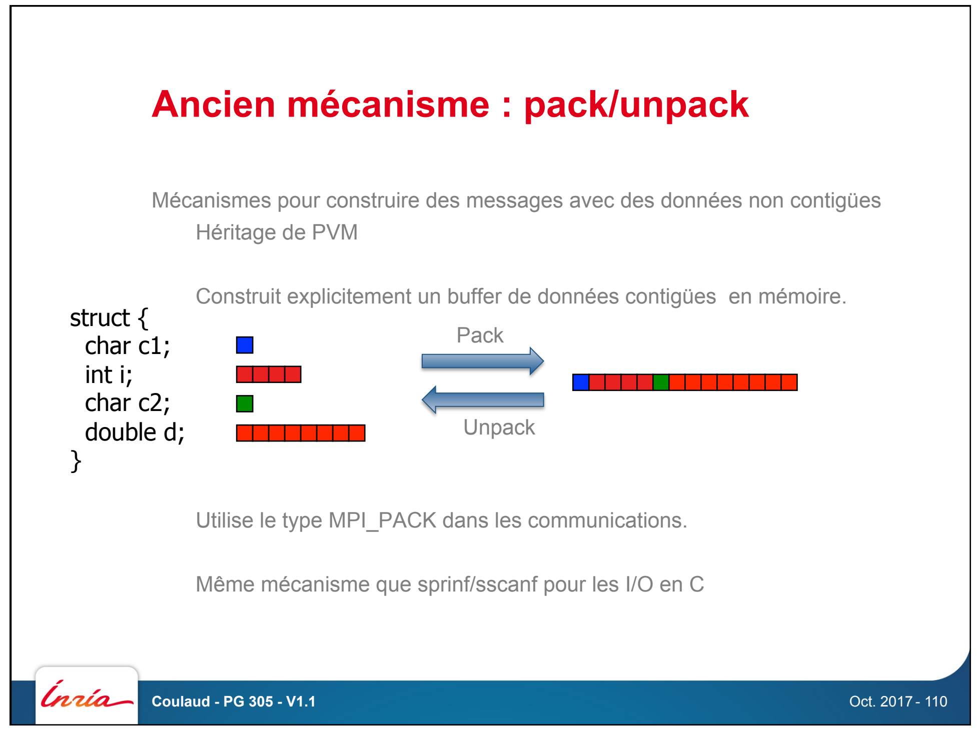
i receive the lbloc 2x2 diagonal elements from 0
Blocs:
11 12 21 22 33 34 43 44 55 56 65 66

11	12	13	14	15	16
21	22	23	24	24	26
31	32	33	34	35	36
41	42	43	44	45	46
51	52	53	54	55	56
61	62	63	64	65	66

MPI_Recv (&a[0][0], 1, DiagBloc2new, 0, tag, MPI_COMM_WORLD, &status);

I am process 1
i receive the bloc 2x2 diagonal elements from 0
matrix::

```
11  12  0  0  0  0
21  22  0  0  0  0
 0   0 33 34  0  0
 0   0 43 44  0  0
 0   0  0  0 55 56
 0   0  0  0 65 66
```

Ancien mécanisme : pack/unpack

MPI_Pack(inbuf, incount, datatype,outbud, outsize, position comm)

IN	inbuf	buffer d'input
IN	incount	nombre d'éléments (entier positif)
IN	datatype	type des données (MPI_Datatype handle)
OUT	outbuf	buffer d'output
IN	incount	nombre d'éléments (entier positif)
INOUT	position	position courrante dans le buffer en byte (entier)
IN	comm	communicateur pour les messages utilisant le type PACK

Utilise un communicateur pour optimiser la conversion de données
- Si machine hétérogène : XDR
- Si le communicateur n'est pas précisé on encode en XDR.

Types versus pack/unpack

Programmation pack/unpack plus facile. On construit le buffer quand on en a besoin.

Généralement
- le buffer utilisé par le pack est plus gourmand en mémoire ;
- Le type est plus rapide dans la plupart des implémentations ;
- Même taille des messages et temps de communication.

Utiliser le type MPI pour améliorer la performance mémoire et CPU

Mais Pack/Unpack doit être utilisé pour des types compliqués, dynamiques, …

Exercice

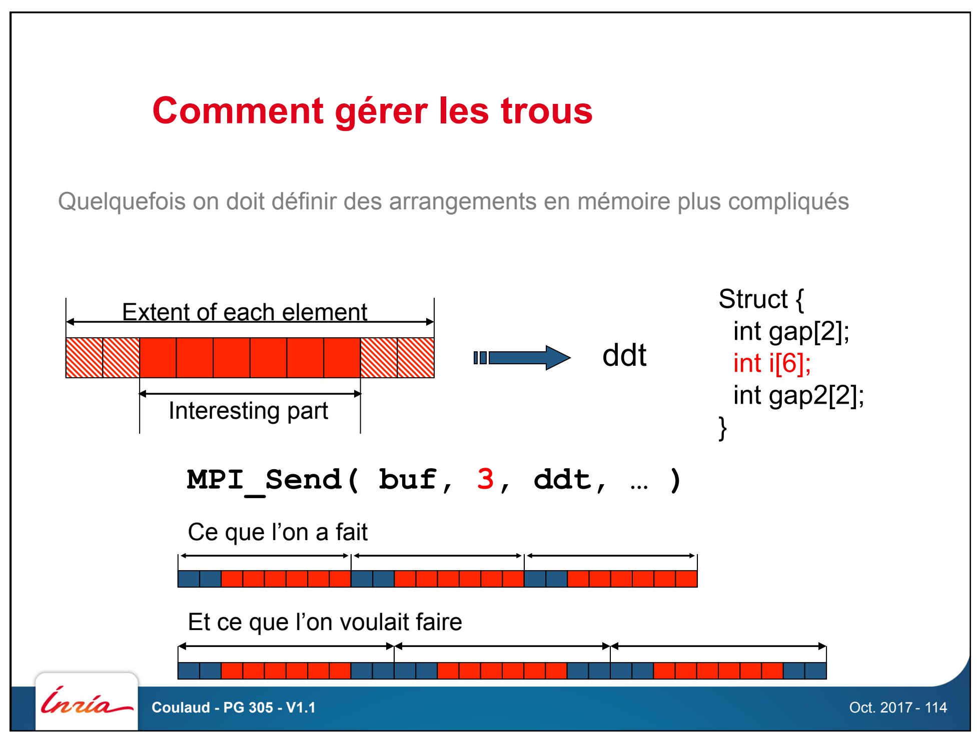
```
typedef struct {
        double pos[3];
        double v[3];
        double f[3];
        double mass;
        int    globalID;
        char   name[4] ;
} Particule;
```

Écrire le type de données pour envoyer un objet Particule.

En général, pour évaluer les interactions entre deux particules, nous n'avons besoin que des positions. Optimiser le type, pour n'envoyer que les positions.

Comment gérer les trous

Quelquefois on doit définir des arrangements en mémoire plus compliqués



Marqueurs Lower-Bound et Upper-Bound

Définir des types avec des trous au début ou à la fin

- 2 pseudo-types : MPI_LB and MPI_UB
- utilisé avec MPI_TYPE_STRUCT si

$$\text{Typemap} = \{ (\text{type}_0, \text{disp}_0), \ldots, (\text{type}_n, \text{disp}_n) \}$$

$$\text{lb(Typemap)} \begin{cases} \text{Min}_j \text{ disp}_j & \text{si aucune entré n'a le type lb} \\ \text{min}_j(\{\text{disp}_j \text{ such that type}_j = \text{lb}) & \text{autrement} \end{cases}$$

$$\text{ub(Typemap)} \begin{cases} \text{Max}_j \text{ disp}_j + \text{sizeof(type}_j) + \text{align} & \text{si aucune entré n'a le type ub} \\ \text{Max}_j\{\text{disp}_j \text{ such that type}_j = \text{ub}\} & \text{autrement} \end{cases}$$

MPI_LB et MPI_UB

```
displs       = (-3, 0, 6 )
blocklengths = ( 1, 1, 1 )
types        = ( MPI_LB, MPI_INT, MPI_UB )
MPI_Type_struct( 3, displs, blocklengths, types, type1 )
```

Typemap= { (lb, -3), (int, 0), (ub, 6) }

```
MPI_Type_contiguous( 3, type1, type2 )
```

Typemap= { (lb, -3), (int, 0), (int, 9), (int, 18), (ub, 24) }

True Lower-Bound and True Upper-Bound Markers

Define the real extent of the datatype: the amount of memory needed to copy the datatype inside
TRUE_LB define the lower-bound ignoring all the MPI_LB markers.

Typemap = { ($type_0$, $disp_0$), …, ($type_n$, $disp_n$) }

true_lb(Typemap) = min_j { $disp_j$: $type_j$!= lb }
true_ub(Typemap) = max_j { $disp_j$ + sizeof($type_j$) : $type_j$!= ub }

Les communicateurs Intra et Inter

On peut vouloir diviser les processus

- Travailler sur des jeux de données différents
- Communication collective sur un sous ensemble de processus

- Eviter des conflits entre des modules/bibliothèque
  Communication bloquante prise par le mauvais module
- Localiser vos opérations MPI à vos codes (+ sûr)

Deux notions

1. Groupe : ensemble ordonné de processus ;
2. Communicateur : groupe mais spécifie en plus un domaine de communication.

Groupes

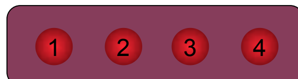
Un groupe est un ensemble de processus ordonnés

- Le groupe a une taille
- Chaque processus a un rang

Sa création est une opération locale

Pourquoi a-t-on besoin d'un groupe ?

- Pour faire une distinction claire entre processus
- Pour autoriser des communications à l'intérieur et entre des sous ensembles de processus
- Pour créer des intra et inter communicateurs …



Création de Groupes

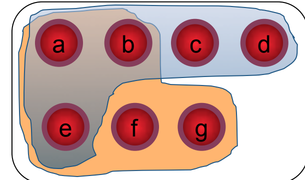
MPI_Group_*(group1, group2, newgroup)

- Où * ∈ {union, intersection, difference }
- Newgroup contient les processus satisfaisants l'opération * d'abord ordonnés selon l'ordre dans le groupe 1 et ensuite en fonction de l'ordre dans le groupe 2.
- Dans le nouveau groupe chaque processus sera présent seulement 1 fois.

Groupe spécial sans aucun processus MPI_GROUP_EMPTY.

MPI_GROUP_NULL signifie que l'objet groupe n'existe pas.

Exemple



group1 = {a, b, c, d, e}
group2 = {e, f, g, b, a}

Union
- newgroup = {a, b, c, d, e, f, g}

Différence
- newgroup = {c, d}

Intersection
- newgroup = {a, b, e}

Groupes

MPI_Group_*(group, n, ranges, newgroup)

- Où * ∈ {incl, excl}
- n est la taille du tableau ranges
- ranges est un tableau de rank impliqué dans l'opération

Pour incl

- n est aussi la taille de newgroup
- ranges : tableaux des rangs qui seront dans newgroup ;
- L'ordre dans newgroup dépend de l'ordre dans ranges.

Pour excl

- ranges : tableaux des rangs qui ne seront pas dans newgroup ;
- L'ordre dans newgroup dépend de l'ordre initiale.

Exemple 2

Groupe = {a, b, c, d, e, f, g, h, i, j}

n = 4, ranks = {2, 3, 0, 4}

INCL : MPI_Group_incl (Groupe, n, ranks, newgroup)

- Newgroup = {c, d, a, e}

EXCL : MPI_Group_excl(Groupe, n, ranks, newgroup)

- Newgroup = {b, f, g, h, i, j}

Groupes

MPI_Group_range_*(group, n, ranges, newgroup)

- Où * ∈ { incl, excl }
- n est la taille du tableau ranges
- ranges est un tableau de triplet d'entier (start, end, stride)

Pour incl

- ranges : tableaux de triplet d'entier définissant les processus qui seront dans newgroup ;
- l'ordre dans newgroup dépend de l'ordre dans ranges.

Pour excl

- ranges : tableaux de triplet d'entier définissant les processus qui ne seront pas dans newgroup ;
- l'ordre dans newgroup dépend de l'ordre initial.

Groupes

Groupe = {a, b, c, d, e, f, g, h, i, j}
n = 3 ;
ranges = ((6, 7, 1), (1, 6, 2), (0, 9, 4))

Quels sont les processus désignés par ranges

- (6, 7, 1) => {g, h} (ranks (6, 7))
- (1, 6, 2) => {b, d, f} (ranks (1, 3, 5))
- (0, 9, 4) => {a, e, i} (ranks (0, 4, 8))

incl

- Newgroup = {g, h, b, d, f, a, e, i}

excl

- Newgroup = {c, j}

Quelques méthodes supplémentaires

Un constructeur à partir du communicateur

- MPI_Comm_group(comm, group)
- MPI_Group_size(group , size)
- MPI_Group_rank(group , rank)

rank du processus dans le groupe sinon MPI_UNDEFINED

Correspondance entre les rangs des processus du group1 et leur rang dans group2

- MPI_Group_translate_ranks(group1, n, rank1, group2, rank2)

Comparer deux groupes MPI_Group_compare(group1,group2, result). Le résultat est

- MPI_IDENT si les éléments des groupes sont les mêmes ainsi les rangs ;
- MPI_SIMILAR si les éléments des groupes sont les mêmes mais avec des rangs différents ;
- MPI_UNEQUAL sinon

Supprimer un groupe : MPI_Group_free(group)

Les communicateurs

Un canal entre des processus pour échanger des messages.

Les opérations pour

- créer les communicateurs sont **collectives** ;
- accéder à une information d'un communicateur sont **locales**.

Communicateurs spéciaux :

MPI_COMM_WORLD, MPI_COMM_NULL, MPI_COMM_SELF

Créer une copie identique (groupe, topologie)

MPI_Comm_dup(comm, newcomm)

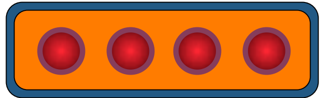
Permet d'échanger des messages entre le même ensemble de processus en utilisant des étiquettes identiques (utilespour développer des bibliothèques).

Deux types de communicateurs :

- **Intracommunicateur** : communication dans un groupe.
- **Intercommunicateur** : communication entre deux groupes.

Intracommunicateurs

Qu'est ce qu'un communicateur?



- Des processus
- Un groupe
- Un communicateur
- Une topologie (optionnel)

Objet opaque interrogeable par :

MPI_COMM_SIZE, MPI_COMM_RANK

MPI_Comm_compare(conn1, comm2, result)
- MPI_IDENT: comm1 and comm2 représente le même communicateur
- MPI_CONGRUENT: même processus, même rangs
- MPI_SIMILAR: même processus, rangs différents
- MPI_UNEQUAL: sinon

Intracommunicateurs

MPI_Comm_create(comm, group, newcomm)
- Créer un nouveau communicateur sur tous les processus du communicateur comm qui sont définis sur le groupe ;
- Tous les autres processus ont MPI_COMM_NULL.

Exemple

0	1	2	3	4	5	6	7	8	9

0	2	4	6	8

1	3	5	7	9

```
MPI_Group_range_incl( group, 1, (0, 9, 2), even_group);
MPI_Group_range_excl( group, 1, (0, 9, 2), odd_group );

MPI_Comm_create( comm, even_group, even_comm );
MPI_Comm_create( comm, odd_group,  odd_comm  );
```

```
int main(int argc, char **argv) {
  MPI_Group   group,even_group ,odd_gr
  int rank,grank;
```

```
MPI_Group_rank(even_group , &grank);
printf(" rank %d Groupe %d\n ",rank, grank)
```

rank 3 Groupe -32766
rank 4 Groupe 2
rank 5 Groupe -32766

```
MPI_Group_rank(odd_group , &grank);
printf(" rank %d Groupe %d\n ",rank, grank);
```

rank 3 Groupe 1
rank 4 Groupe -32766
rank 5 Groupe 2
rank 7 Groupe 3
rank 6 Groupe -32766
rank 1 Groupe 0
rank 2 Groupe -32766
rank 9 Groupe 4
rank 0 Groupe -32766
rank 8 Groupe -32766

Intracommunicateurs

MPI_Comm_splt(comm, color, key, newcomm)

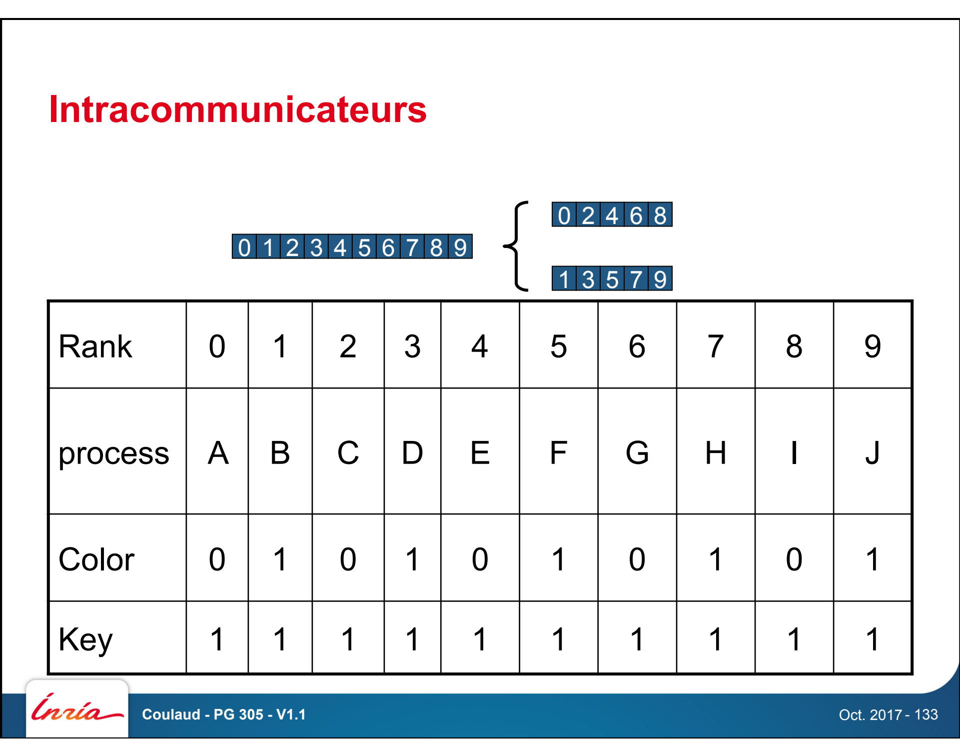
- color : contrôle l'affectation du sous ensemble ;
- key : contrôle l'affectation du rang.

rank	0	1	2	3	4	5	6	7	8	9
process	A	B	C	D	E	F	G	H	I	J
color	0	⊥	3	0	3	0	0	5	3	⊥
key	3	1	2	5	1	1	1	2	1	0

3 différentes couleurs => 3 communicateurs

1. {A, D, F, G} avec les rangs {3, 5, 1, 1} => {F, G, A, D}
2. {C, E, I} avec les rangs {2, 1, 3} => {E, I, C}
3. {H} avec les rangs {1} => {H}

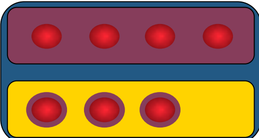
B et J ont **MPI_COMM_NULL** comme ils ont une couleur non définie (MPI_UNDEFINED)



Rank	0	1	2	3	4	5	6	7	8	9
process	A	B	C	D	E	F	G	H	I	J
Color	0	1	0	1	0	1	0	1	0	1
Key	1	1	1	1	1	1	1	1	1	1

Intercommunicateurs

Qu'est-ce qu'un intercommunicateur?



- Plus de processus
- **Deux** groupes
- Un communicateur

Objet qui construit un domaine de communication entre deux groupes disjoints de processus.

Autorise des communications entre les groupes

Très utile pour les couplages de codes, les simulations client/serveur

Intercommunicateurs

Propriétés :

- La syntaxe pour les communications est la même ;
- Le processus destinataire est identifié par son rang dans le groupe distant ;
- Pas de conflit garanti avec une autre communication qui utilise un communicateur différent ;

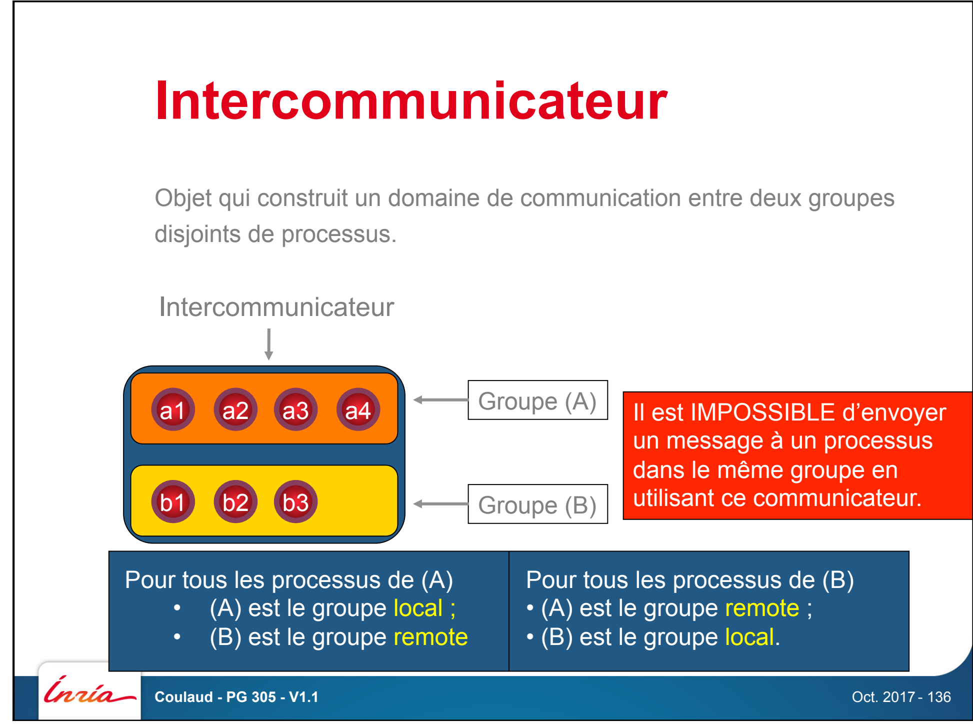
Accéder aux informations

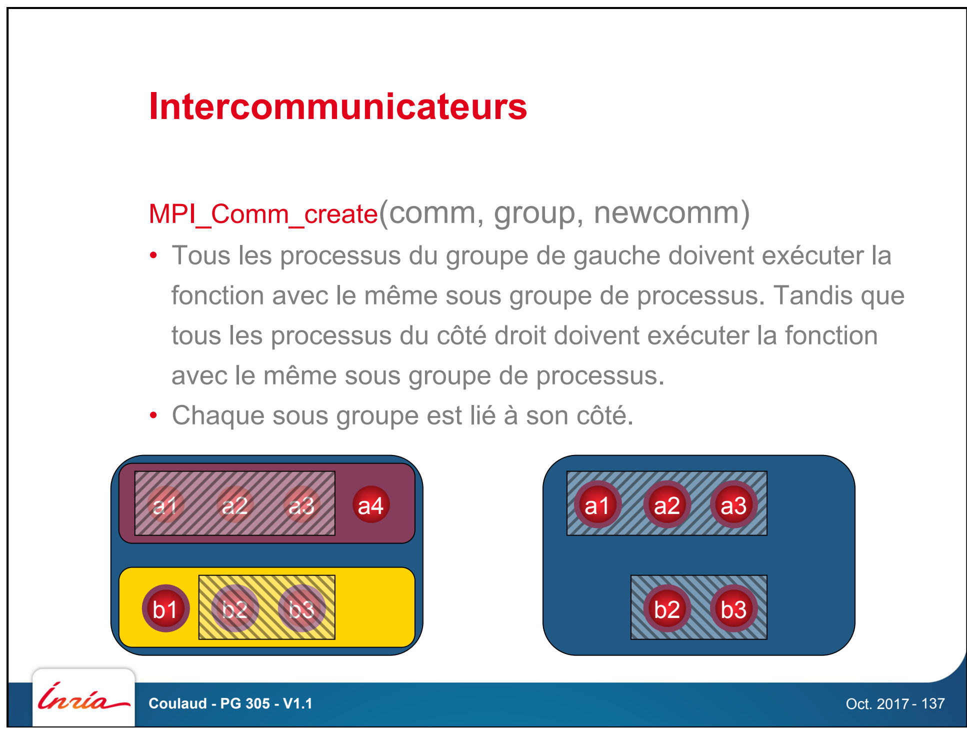
- **Locale**
  MPI_Comm_size, MPI_Comm_rank
- **Remote**
  MPI_Comm_remote_size, MPI_Comm_remote_rank

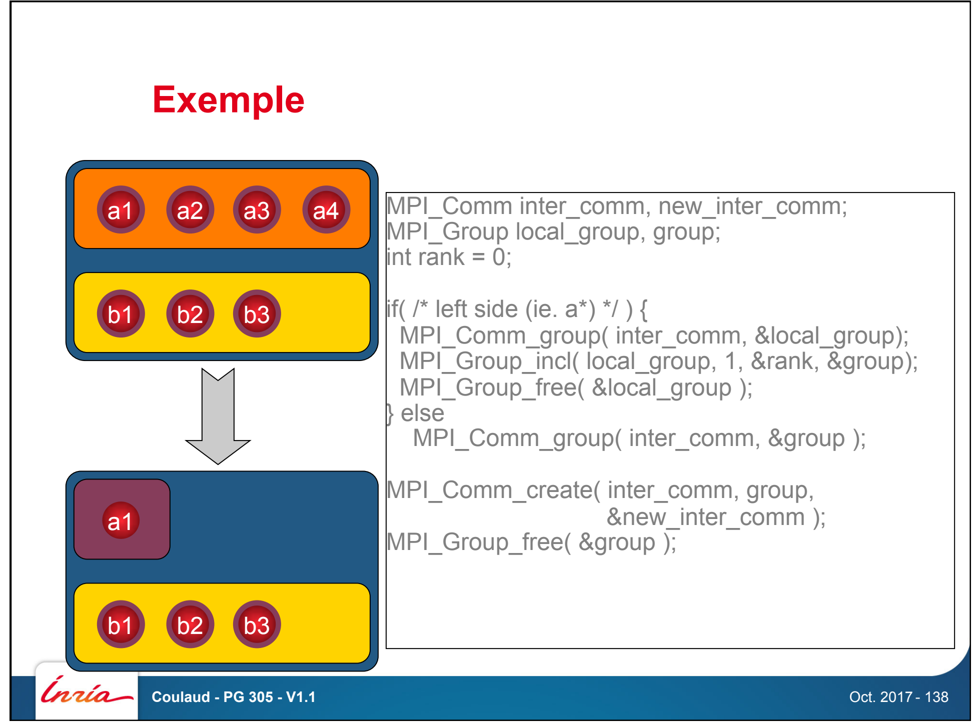
Tester la nature du communicateur
MPI_Comm_Test_inter(comm, flag)

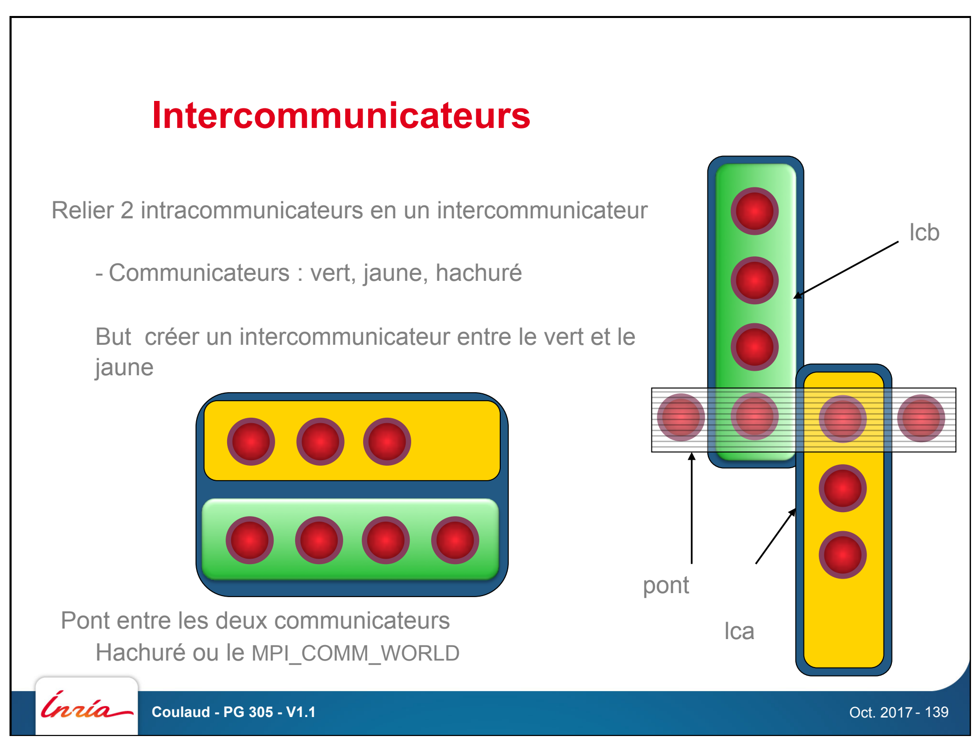
Intercommunicateur

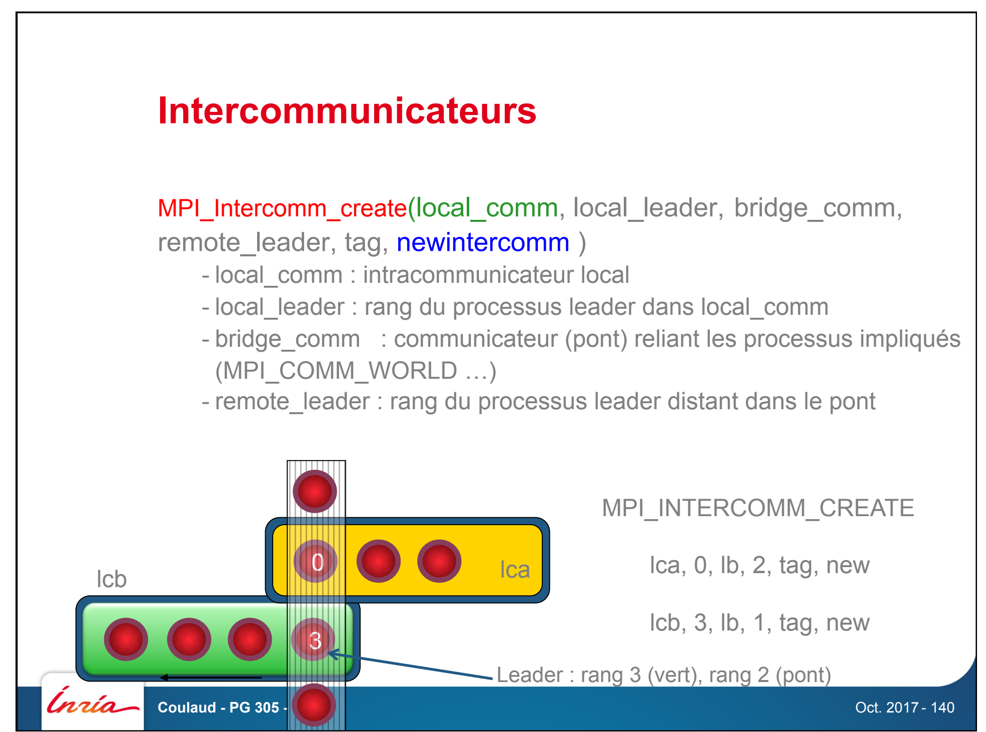
Objet qui construit un domaine de communication entre deux groupes disjoints de processus.

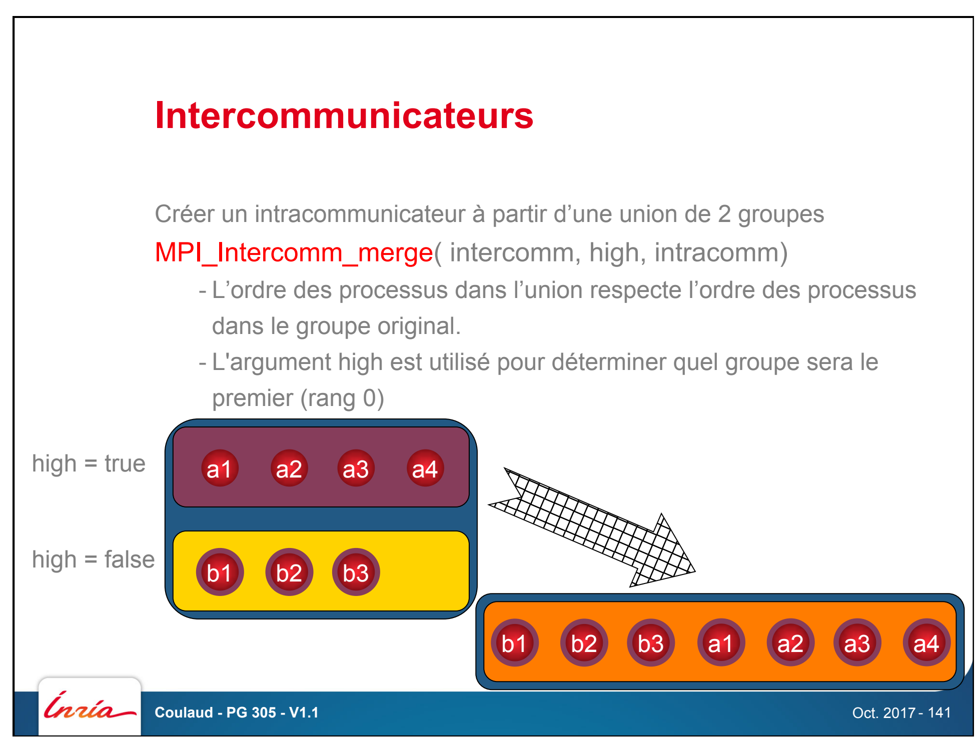


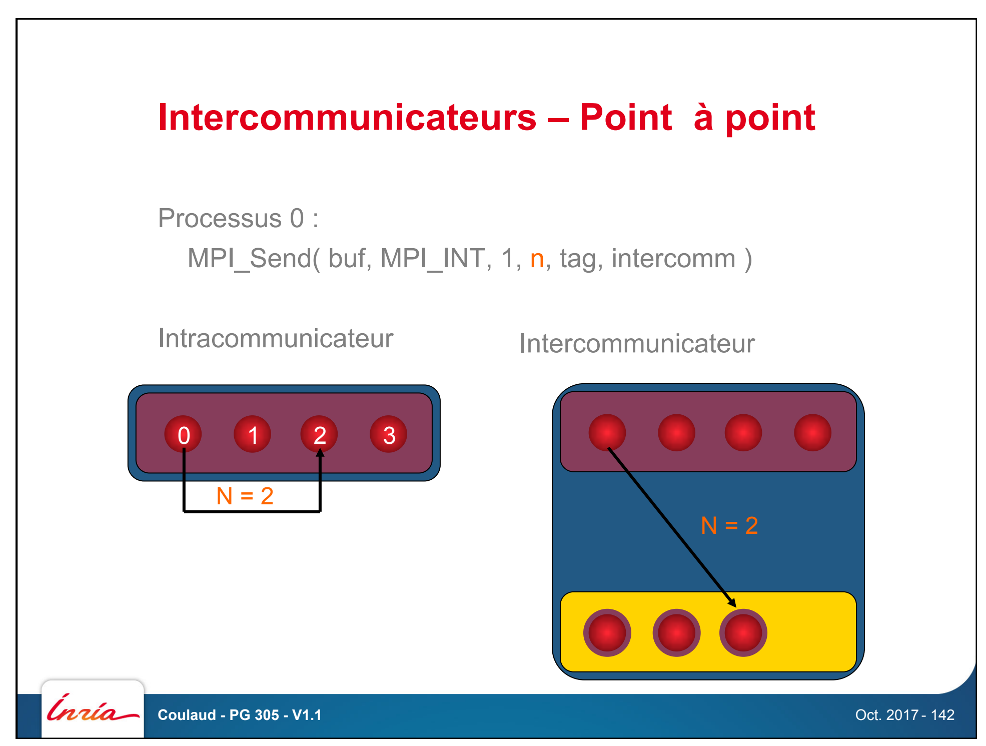

Intercommunicateurs
MPI_Comm_create(comm, group, newcomm)
Tous les processus du groupe de gauche doivent exécuter la fonction avec le même sous groupe de processus. Tandis que tous les processus du côté droit doivent exécuter la fonction avec le même sous groupe de processus.
Chaque sous groupe est lié à son côté.
a1
a2
a3
a4
b1
b2
b3
a1
a2
a3
b2
b3
Coulaud - PG 305 - V1.1
Oct. 2017 - 137

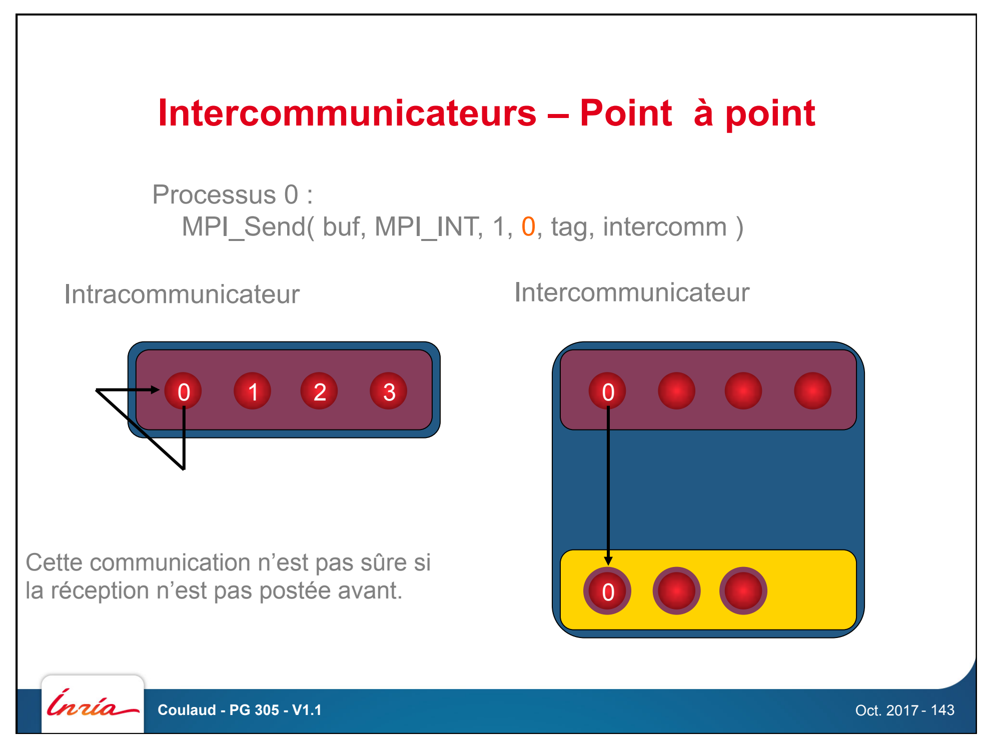

Exemple
a1
a2
a3
a4
b1
b2
b3
a1
b1
b2
b3
MPI_Comm inter_comm, new_inter_comm;
MPI_Group local_group, group;
int rank = 0;
if(/* left side (ie. a*) */) {
MPI_Comm_group(inter_comm, &local_group);
MPI_Group_incl(local_group, 1, &rank, &group);
MPI_Group_free(&local_group);
} else
MPI_Comm_group(inter_comm, &group);
MPI_Comm_create(inter_comm, group,
&new_inter_comm);
MPI_Group_free(&group);
Coulaud - PG 305 - V1.1
Oct. 2017 - 138

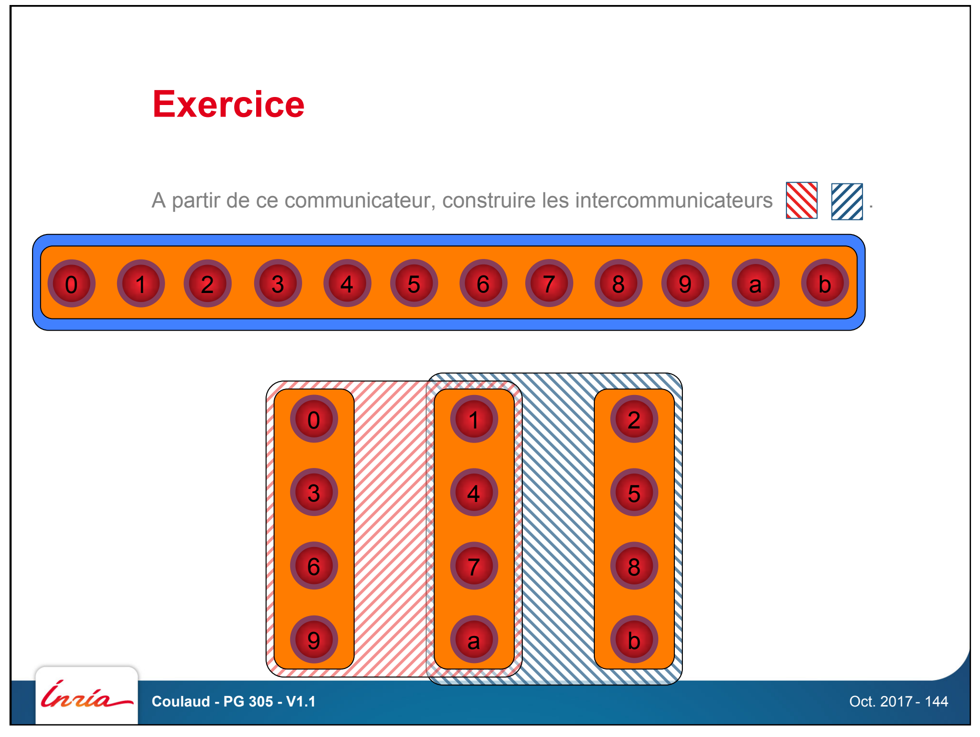

Intercommunicateurs
Relier 2 intracommunicateurs en un intercommunicateur
- Communicateurs : vert, jaune, hachuré
But créer un intercommunicateur entre le vert et le jaune
Pont entre les deux communicateurs
Hachuré ou le MPI_COMM_WORLD
Icb
pont
Ica
Coulaud - PG 305 - V1.1
Oct. 2017 - 139

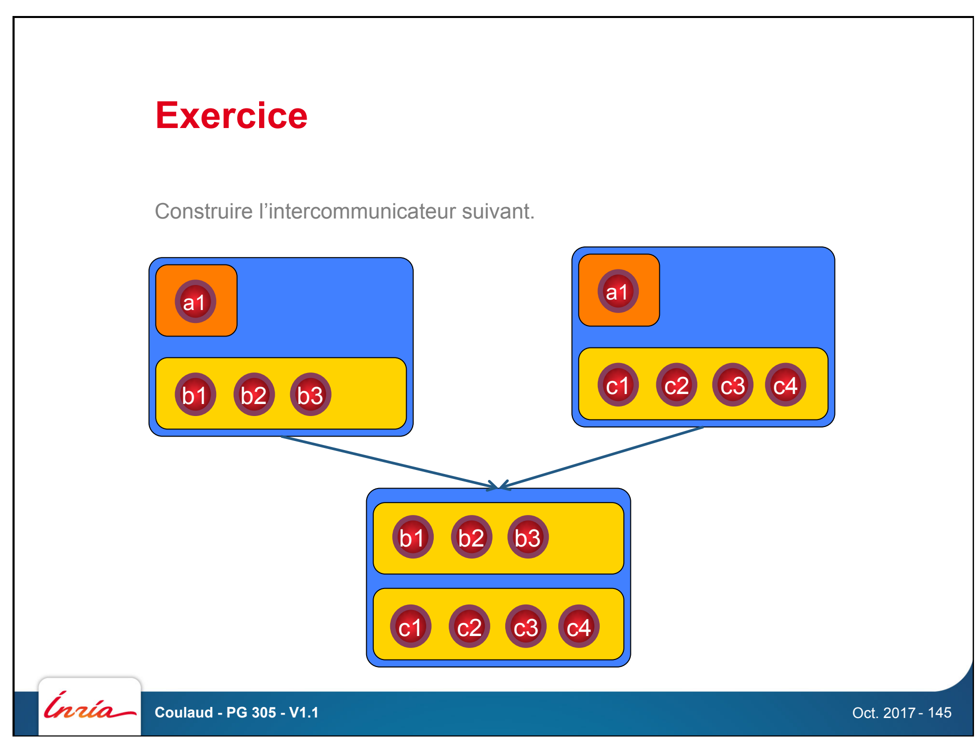

Intercommunicateurs
MPI_Intercomm_create(local_comm, local_leader, bridge_comm, remote_leader, tag, newintercomm)
- local_comm : intracommunicateur local
- local_leader : rang du processus leader dans local_comm
- bridge_comm : communicateur (pont) reliant les processus impliqués (MPI_COMM_WORLD …)
- remote_leader : rang du processus leader distant dans le pont
MPI_INTERCOMM_CREATE
Ica, 0, Ib, 2, tag, new
Icb, 3, Ib, 1, tag, new
Icb
Ica
0
3
Leader : rang 3 (vert), rang 2 (pont)
Coulaud - PG 305 -
Oct. 2017 - 140

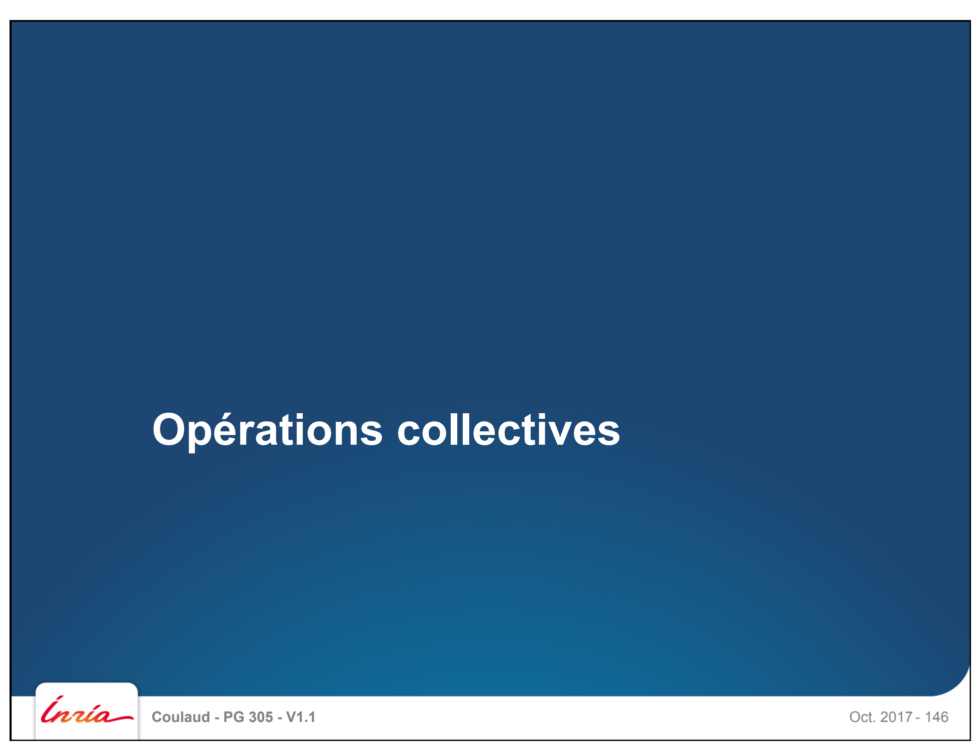

Intercommunicateurs
Créer un intracommunicateur à partir d'une union de 2 groupes
MPI_Intercomm_merge(intercomm, high, intracomm)
- L'ordre des processus dans l'union respecte l'ordre des processus dans le groupe original.
- L'argument high est utilisé pour déterminer quel groupe sera le premier (rang 0)
high = true
a1 a2 a3 a4
high = false
b1 b2 b3
b1 b2 b3 a1 a2 a3 a4
Coulaud - PG 305 - V1.1
Oct. 2017 - 141


Intercommunicateurs – Point à point
Processus 0 :
MPI_Send(buf, MPI_INT, 1, n, tag, intercomm)
Intracommunicateur
Intercommunicateur
0 1 2 3
N = 2
N = 2
Coulaud - PG 305 - V1.1
Oct. 2017 - 142


Intercommunicateurs – Point à point
Processus 0 :
MPI_Send(buf, MPI_INT, 1, 0, tag, intercomm)
Intracommunicateur
Intercommunicateur
0
1
2
3
0
0
Cette communication n'est pas sûre si
la réception n'est pas postée avant.
Coulaud - PG 305 - V1.1
Oct. 2017 - 143


Exercice
A partir de ce communicateur, construire les intercommunicateurs
0 1 2 3 4 5 6 7 8 9 a b
0 3 6 9
1 4 7 a
2 5 8 b
Coulaud - PG 305 - V1.1
Oct. 2017 - 144


Exercice
Construire l'intercommunicateur suivant.
a1
b1 b2 b3
a1
c1 c2 c3 c4
b1 b2 b3
c1 c2 c3 c4
Coulaud - PG 305 - V1.1
Oct. 2017 - 145


Opérations collectives
Coulaud - PG 305 - V1.1
Oct. 2017 - 146

Notions générales

Elles permettent de faire en une opération une série de communications point à point.

Elles **impliquent tous les processus du communicateur** indiqué.

Elles sont plus **restrictives** que les communications points à points

1. Pas d'étiquettes (gérées par le système)
   - Jamais définies explicitement ;
   - Pas d'interférence avec les communications point à point.
2. Accord entre le nombre de données envoyées et les données reçues ;
3. Les appels doivent correspondre exactement à l'ordre d'exécution ;
4. Les communications collectives sont **bloquantes** et il n'y a qu'un seul mode analogue au mode standard du Pt à Pt.

Sémantique

L'appel se termine dès que la participation du processus dans la communication globale est terminée.

- Cela signifie que le processus peut modifier les buffers impliqués dans l'opération collective ;
- N'indique pas que les autres processus ont terminés ou même commencé l'opération collective ;
- Un processus ne peut terminer tant que les processus impliqués dans l'opération collectives ait atteint l'appel de l'invocation de l'opération collective.

Ainsi une opération collective **peut** ou **ne peut pas** avoir un effet de synchronisation sur les processus (dépendant de l'implémentation) sauf pour la barrière.

Permet d'avoir des communications efficaces dans certaines implémentations
Généralement openMPI, MPCH2 les opérations sont synchronisantes

Communicators - Collectives

Classification simple par type d'opérations

One-To-All (simplex mode)
Un processus contribue au résultat. Tous les processus reçoivent le résultat.
- MPI_Bcast
- MPI_Scatter, MPI_Scatterv

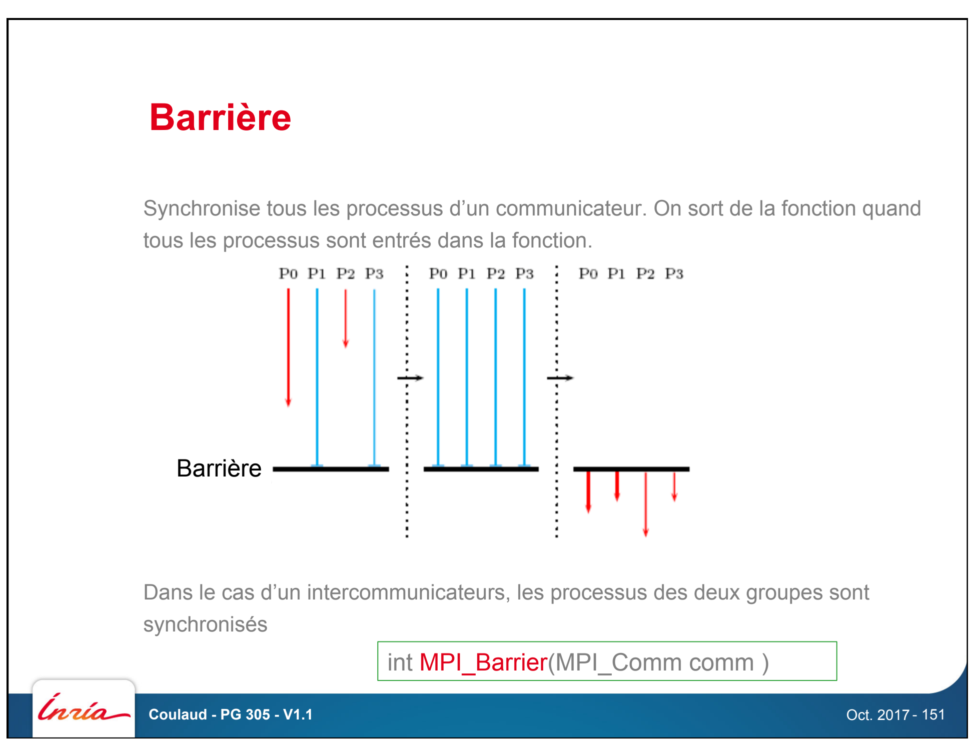
All-To-One (simplex mode)
Tous les processus contribuent au résultat. Un processus reçoit le résultat.
- MPI_Gather, MPI_Gatherv
- MPI_Reduce

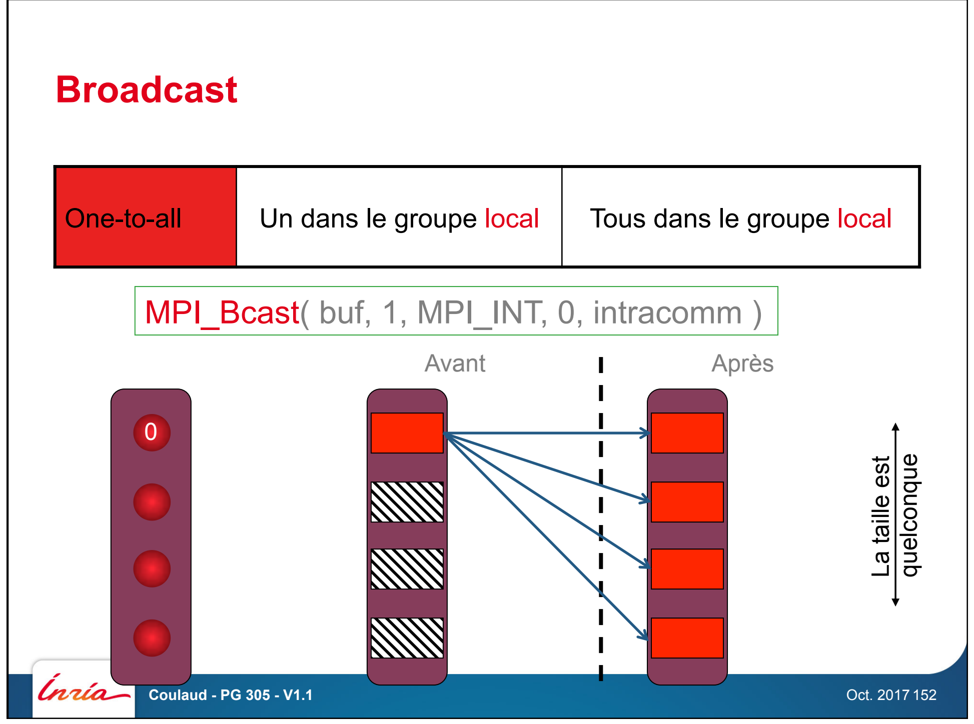
All-To-All (duplex mode)
Tous les processus contribuent au résultat et reçoivent le résultat.
- MPI_Allgather, MPI_Allgatherv
- MPI_Alltoall, MPI_Alltoallv
- MPI_Allreduce, MPI_Reduce_scatter

Other
Opérations collectives qui ne correspondent pas à un cas ci-dessus.
- MPI_Scan
- MPI_Barrier

Collectives

	Qui génère le résultat	Qui reçoit le résultat
One-to-all	Un dans groupe local	Tous dans le groupe local
All-to-one	Tous dans le groupe local	Un dans le groupe local
All-to-all	Tous dans le groupe local	Tous dans le groupe local
Autres	?	?


Barrière
Synchronise tous les processus d'un communicateur. On sort de la fonction quand tous les processus sont entrés dans la fonction.
P0 P1 P2 P3
P0 P1 P2 P3
P0 P1 P2 P3
Barrière
Dans le cas d'un intercommunicateurs, les processus des deux groupes sont synchronisés
int MPI_Barrier(MPI_Comm comm)
Coulaud - PG 305 - V1.1
Oct. 2017 - 151


Broadcast
One-to-all
Un dans le groupe local
Tous dans le groupe local
MPI_Bcast(buf, 1, MPI_INT, 0, intracomm)
Avant
Après
0
La taille est quelconque
Coulaud - PG 305 - V1.1
Oct. 2017 152

Broadcast (exemple)

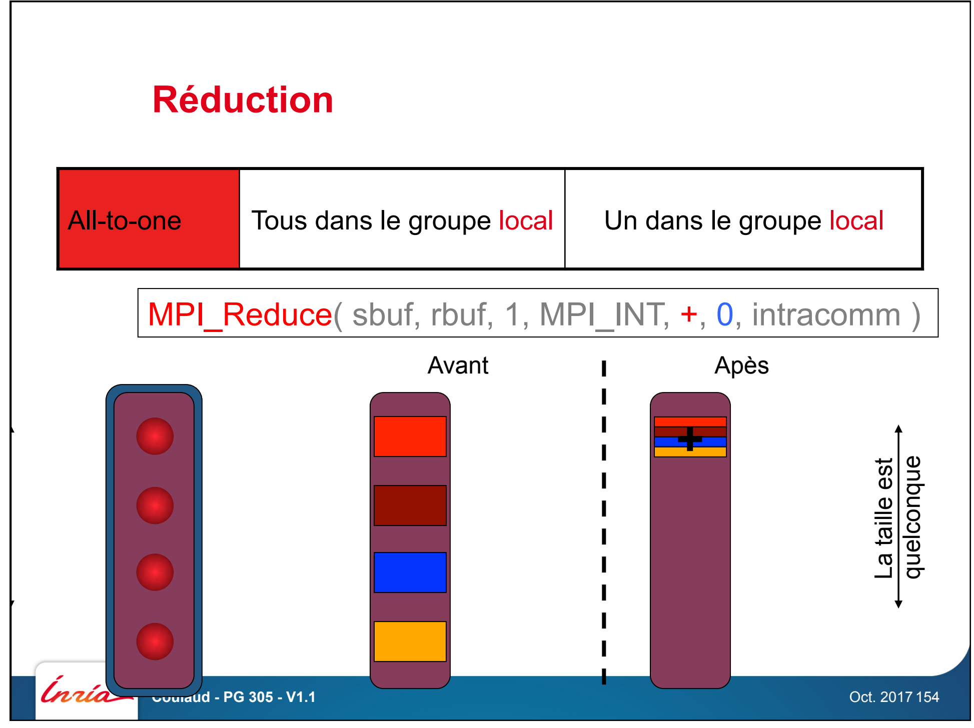
```
If(rank ==0 ){
  MPI_Bcast(&x 1, MPI_INT, 0, comm);
  MPI_Bcast(&y 1, MPI_INT, 0, comm);
 local_work() ;
}
else If(rank ==1 ){
  local_work() ;
  MPI_Bcast(&x 1, MPI_INT, 0, comm);
  MPI_Bcast(&y 1, MPI_INT, 0, comm);
}
else If(rank ==2 ){
  local_work() ;
  MPI_Bcast(&x 1, MPI_INT, 0, comm);
  MPI_Bcast(&y 1, MPI_INT, 0, comm);
}
```

- Tous les processus de comm doivent appeler le broadcast.
- Seul le rank 0 a une valeur dans x et y avant le broadcast.
- Ordre des appels est respectés

Réduction

All-to-one	Tous dans le groupe local	Un dans le groupe local

MPI_Reduce(sbuf, rbuf, 1, MPI_INT, +, 0, intracomm)



Réduction (opérateurs)

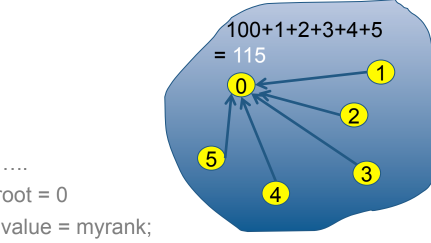
Les opérations prédéfinies

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

On peut définir nos propres opérations avec MPI_Op_create

Réduction (exemple)

On calcule la somme des rangs + 100 → S = 100+size*(size-1)/2




```
…..
root = 0
 value = myrank;
 if (myrank == 0){ value= 100 ; }
 MPI_Reduce(&value,&sum,1,MPI_INT,  MPI_SUM, root, MPI_COMM_WORLD);

 if (myrank == root) {
   printf("La somme calculée est  %d.\n",sum)  ;}
…..
```

Fonction utilisateur pour la réduction

Lier une fonction définie par l'utilisateur à une opération de réduction qui pourra être utilisée dans les fonctions :

- MPI_REDUCE, MPI_ALLREDUCE, MPI_REDUCE_SCATTER,
- MPI_SCAN, MPI_EXSCAN.

La fonction doit être associative.

```
void function ( void* invec, void* inoutvec,
                int* len, MPI_Datatype* datatype)

int MPI_Op_create( MPI_User_function* function,
                   int commute, MPI_Op* op)

int MPI_op_free(MPI_Op *op)
```


Exemple reduceMyOp.c

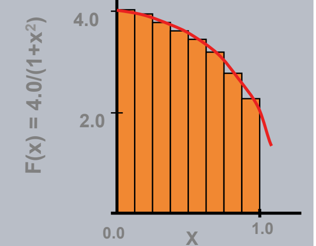
On souhaite calculer $z = \Pi z_i$ avec z_i sur le processeur i

```
typedef struct {
double real,imag; } complex;

void cprod(complex *in, complex *inout, int *len, MPI_Datatype *dptr) {
  int i;
  complex c;
  for (i=0; i<*len; ++i) {
     c.real   = (*in).real * (*inout).real   - (*in).imag * (*inout).imag;
     c.imag  = (*in).real * (*inout).imag + (*in).imag * (*inout).real;
      *inout  =c;
     in++; inout++;
   }
}
```

Exemple reduceMyOp.c (cont)

```
…
MPI_Op myop;
MPI_Datatype ctype;

MPI_Type_contiguous(2,MPI_DOUBLE,&ctype);
MPI_Type_commit(&ctype);
        …
MPI_Op_create(cprod,TRUE,&myop);
MPI_Reduce(&source,&result,1,ctype,myop,root,MPI_COMM_WORLD);

 …
```

Exercice : programme pi



Calculer pi à l'aide de l'intégrale

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

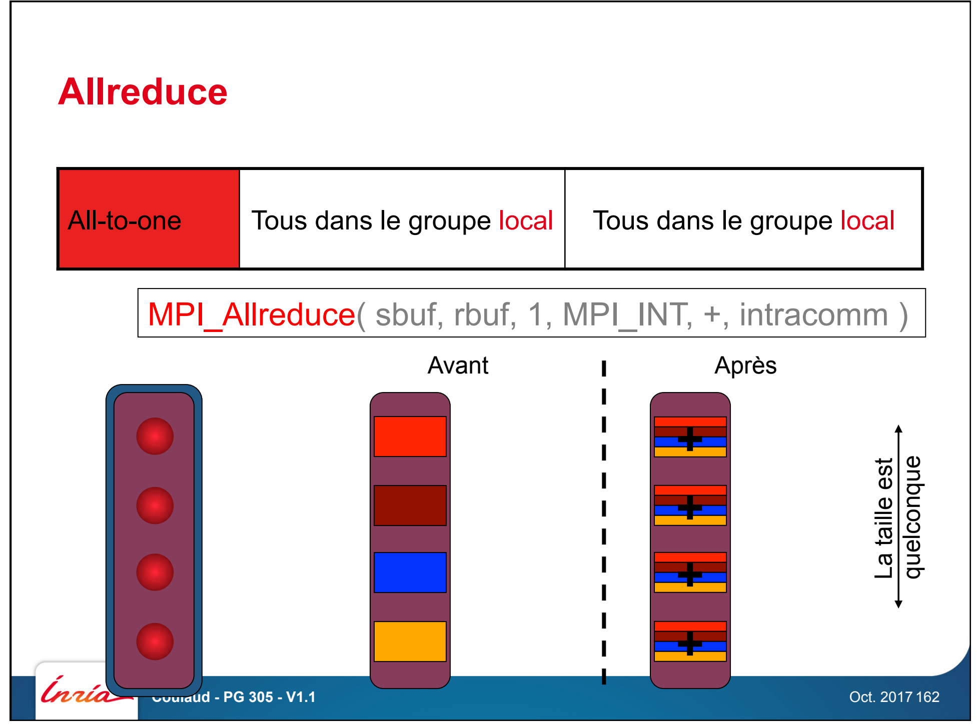
On approche l' intégrale par la somme des aires des rectangles :

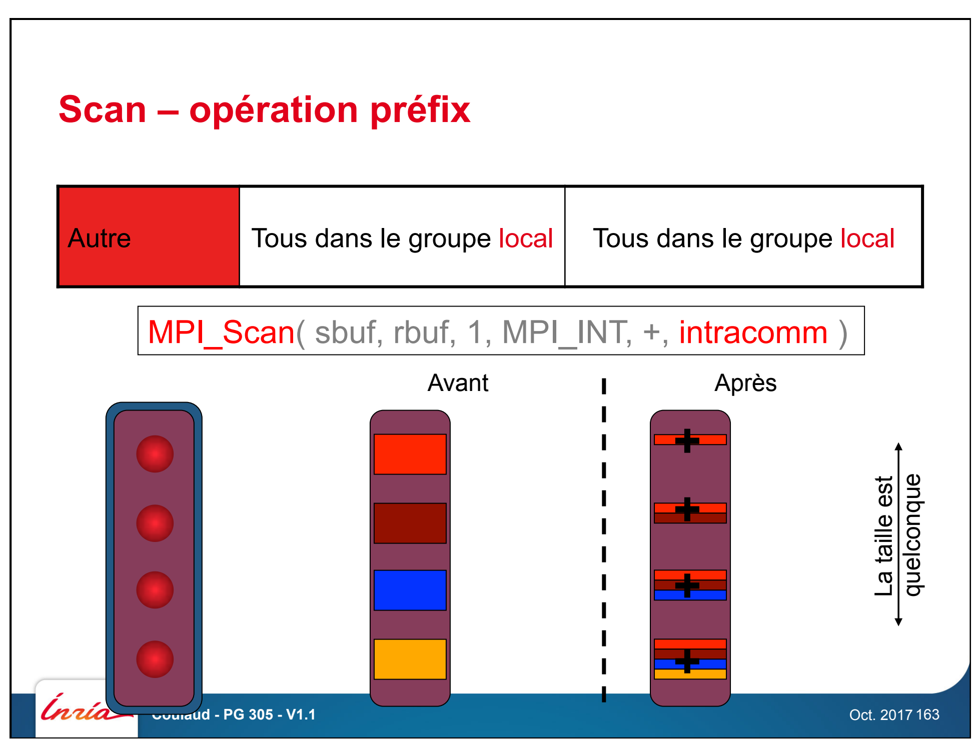
$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

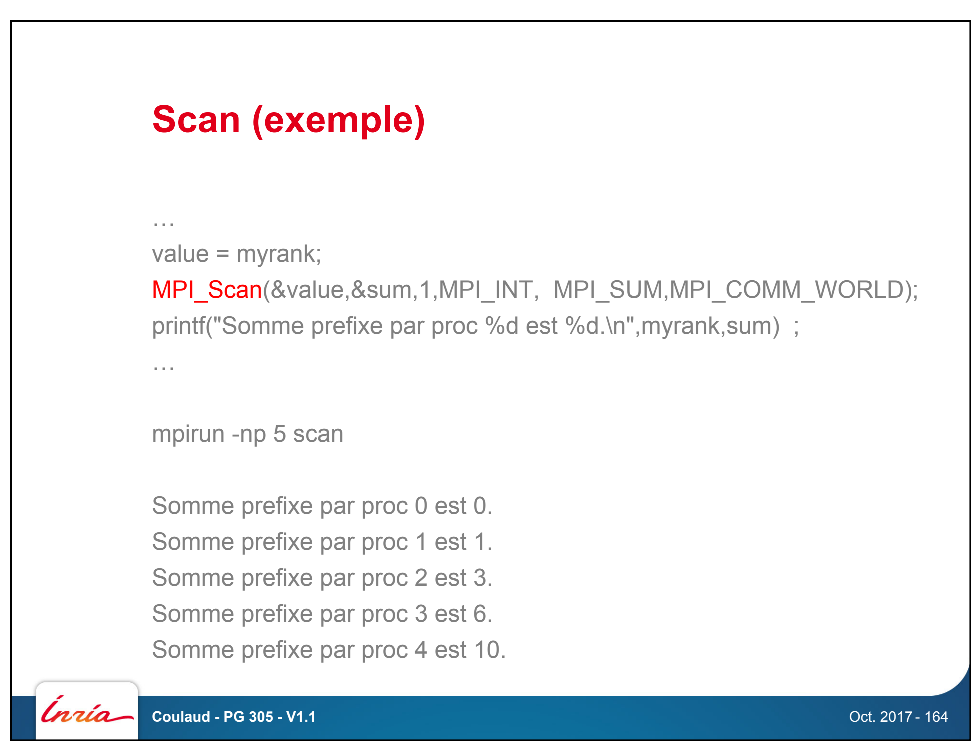
où chaque rectangle a une largeur Δx et une hauteur $F(x_i)$ évaluée au milieu de d' intervalle i.

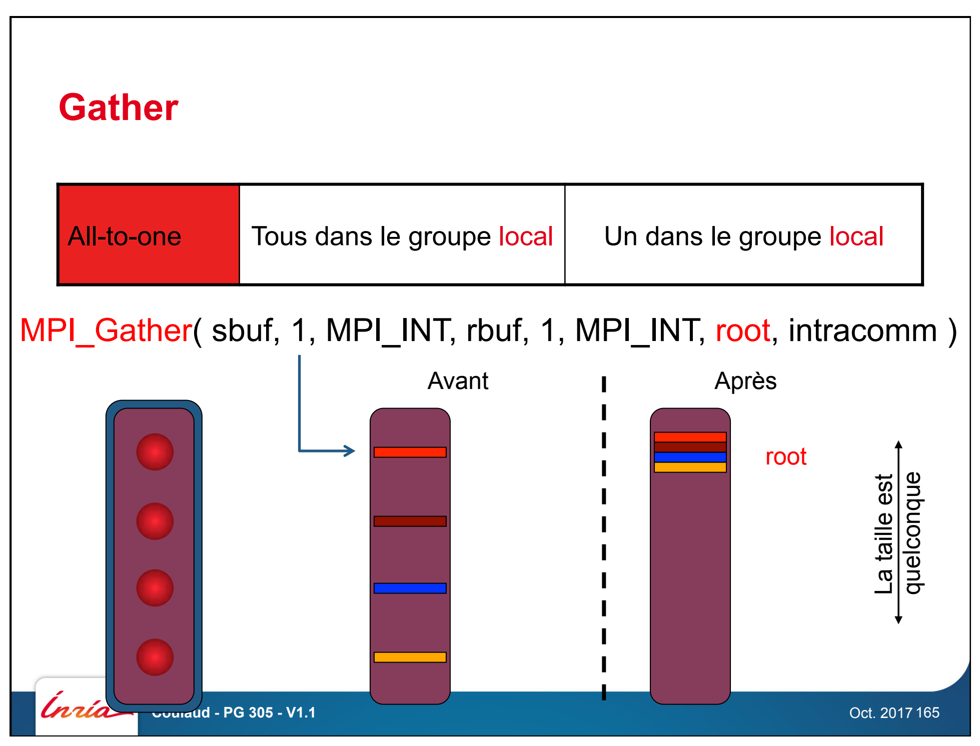
Exercice 2 : programme pi.c

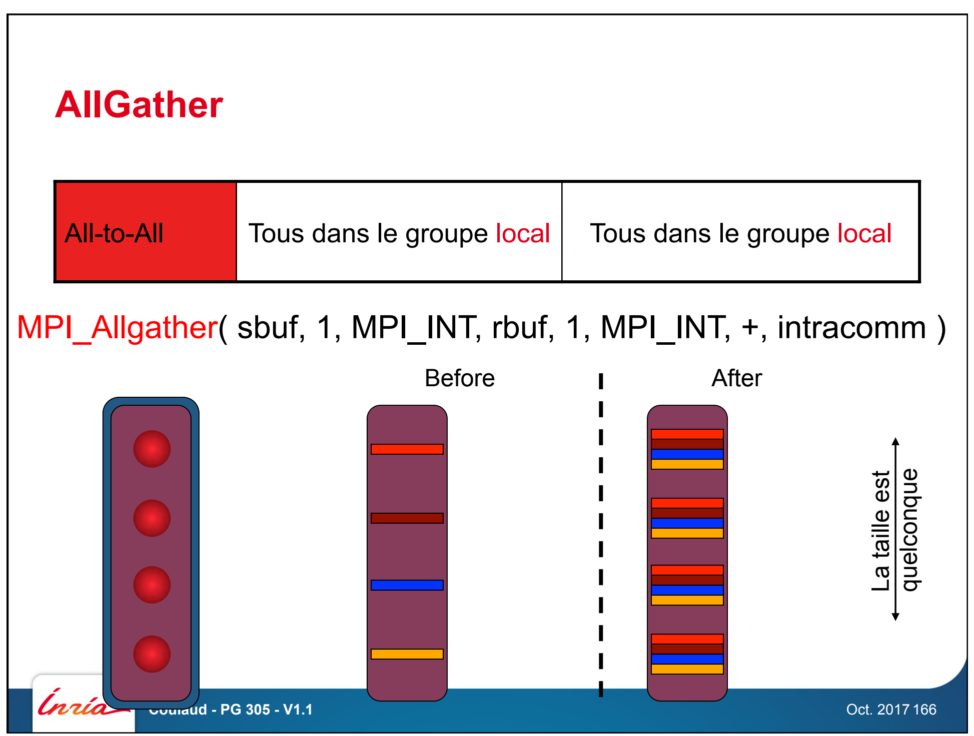
```
double f(double x) {
  return 4.0 / (1 + x*x);
}

int main(int argc, const char** argv) {

  double PI25DT = 3.141592653589793238462643 ;

  const int nIntervals = (argc >= 2) ? atoi(argv[1]) : 100;
  const double invNIntervals = 1.0 / (double) nIntervals;
  double x0, x1 ;
  int i ;

  double surface = 0;
  for( i = 0; i < nIntervals; ++i) {
    const double x = x0 + i * invNIntervals;
    surface += invNIntervals* f(x + invNIntervals*0.5);
  }
  printf("PI %.16f, Error is %e \n", surface, fabs(surface-PI25DT));
}
```

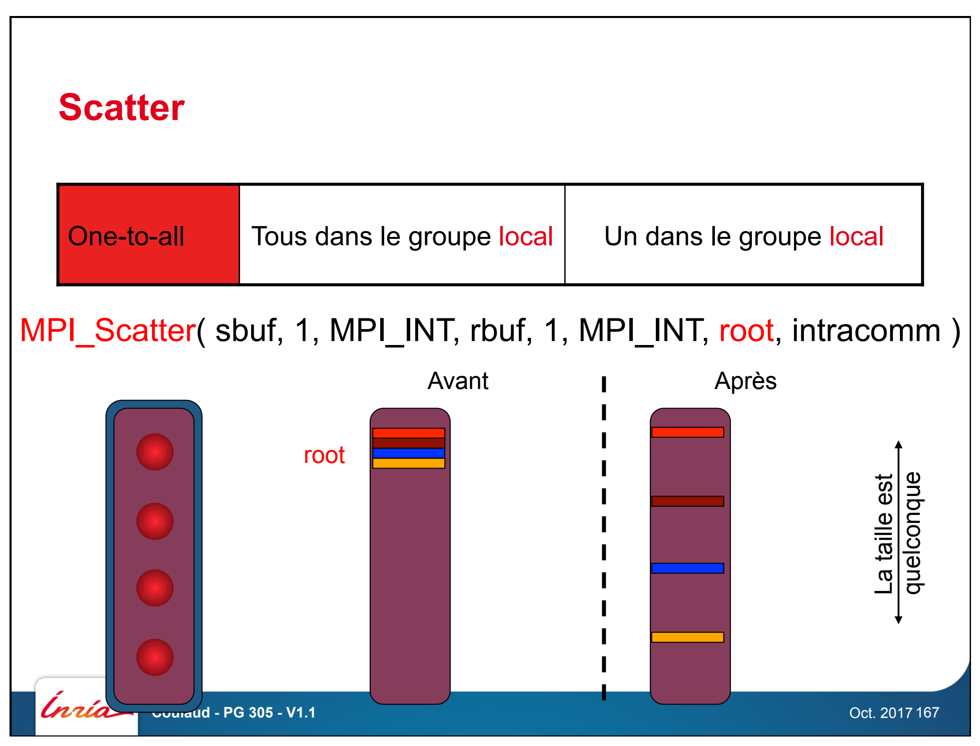



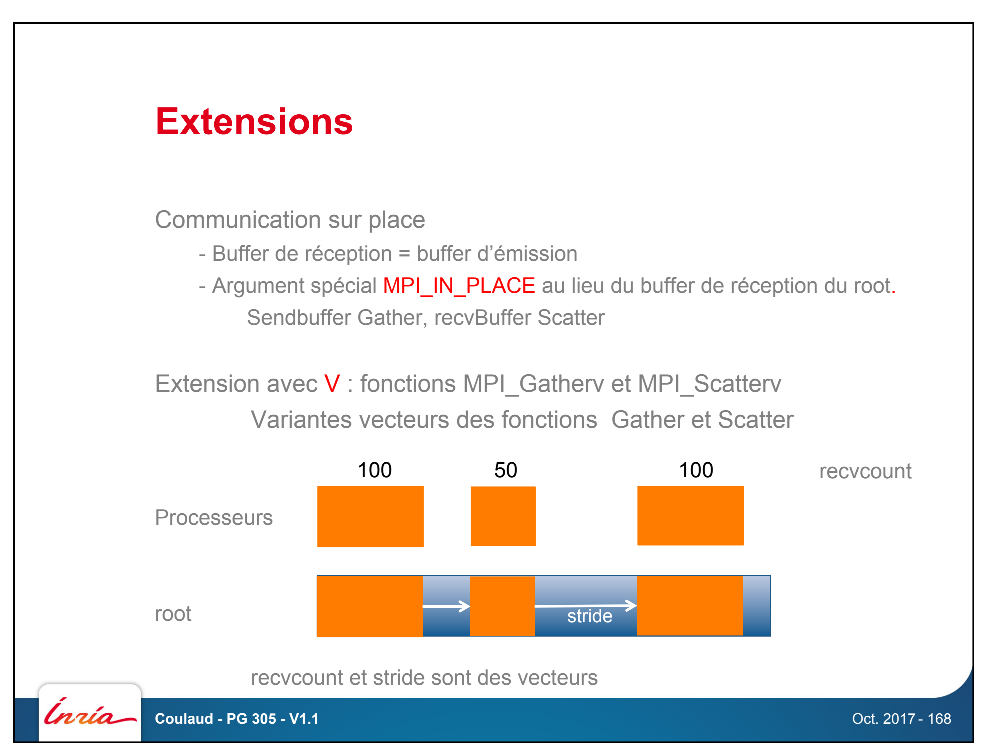

Scan – opération préfix
Autre
Tous dans le groupe local
Tous dans le groupe local
MPI_Scan(sbuf, rbuf, 1, MPI_INT, +, intracomm)
Avant
Après
La taille est quelconque
Coulaud - PG 305 - V1.1
Oct. 2017 163

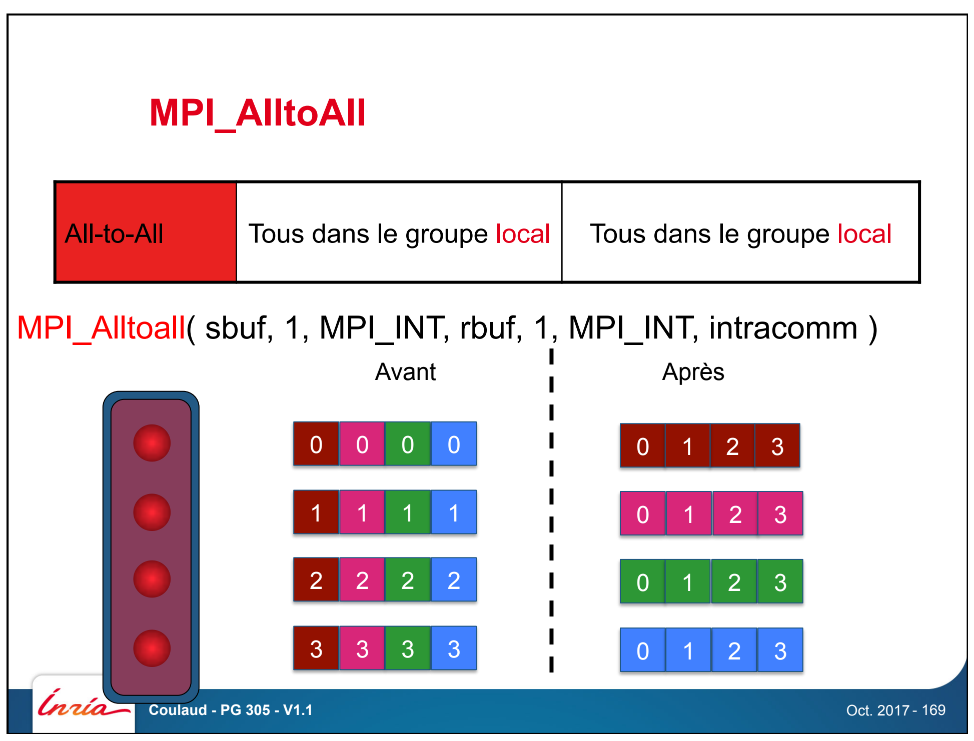

Scan (exemple)
…
value = myrank;
MPI_Scan(&value,&sum,1,MPI_INT, MPI_SUM,MPI_COMM_WORLD);
printf("Somme prefixe par proc %d est %d.\n",myrank,sum) ;
…
mpirun -np 5 scan
Somme prefixe par proc 0 est 0.
Somme prefixe par proc 1 est 1.
Somme prefixe par proc 2 est 3.
Somme prefixe par proc 3 est 6.
Somme prefixe par proc 4 est 10.
Coulaud - PG 305 - V1.1
Oct. 2017 - 164

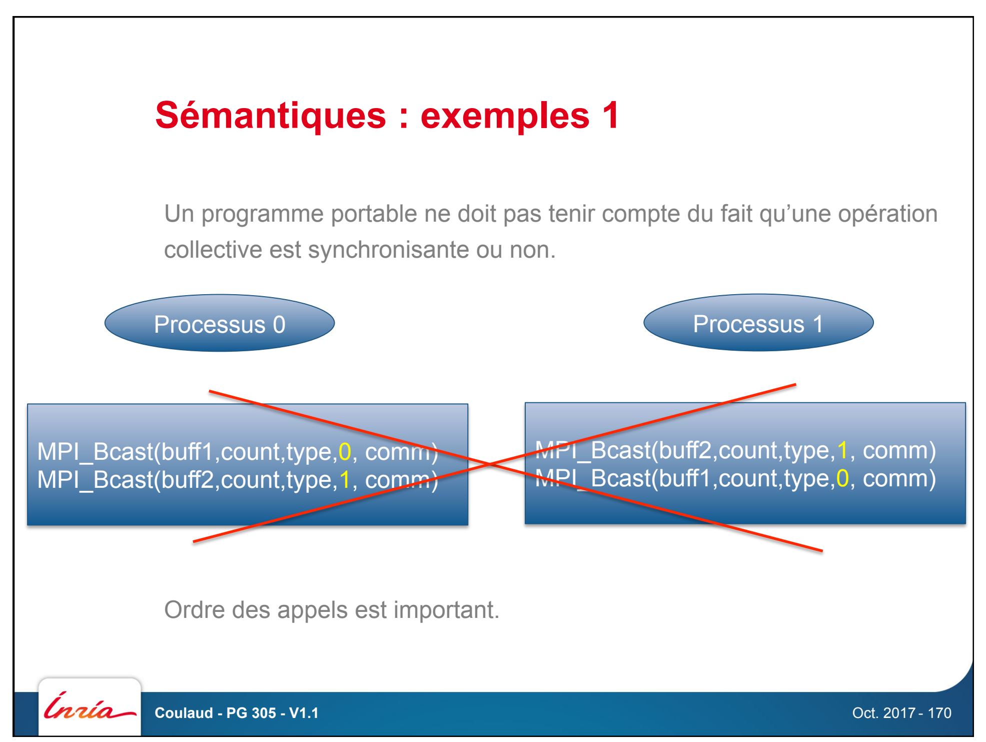

Gather
All-to-one
Tous dans le groupe local
Un dans le groupe local
MPI_Gather(sbuf, 1, MPI_INT, rbuf, 1, MPI_INT, root, intracomm)
Avant
Après
root
La taille est quelconque
Coulaud - PG 305 - V1.1
Oct. 2017 165

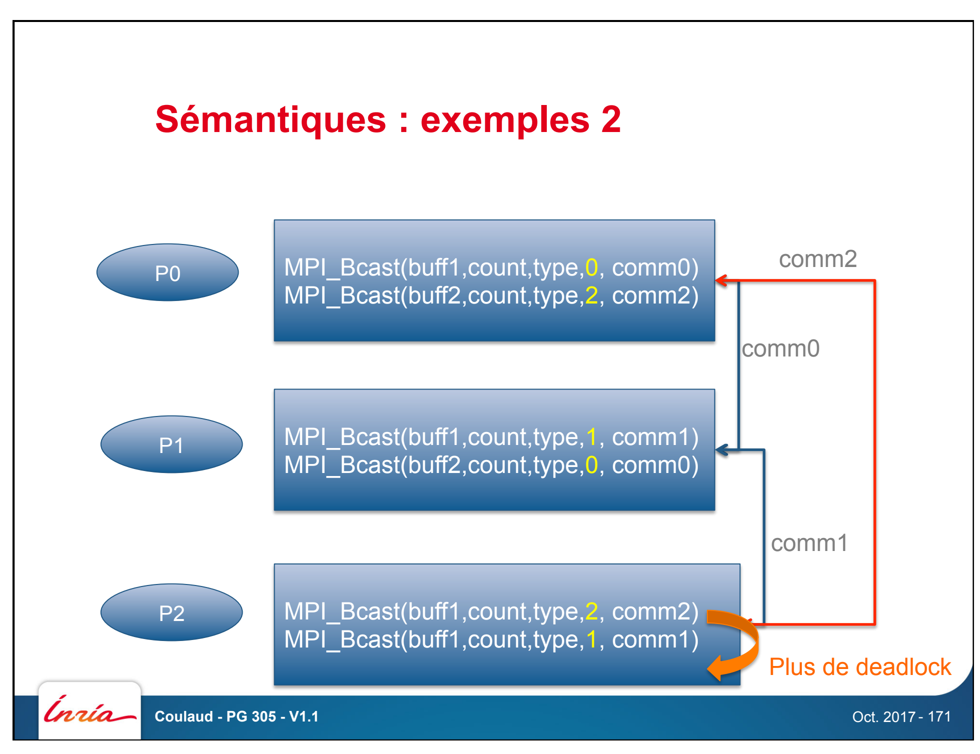

AllGather
All-to-All
Tous dans le groupe local
Tous dans le groupe local
MPI_Allgather(sbuf, 1, MPI_INT, rbuf, 1, MPI_INT, +, intracomm)
Before
After
La taille est quelconque
Coulaud - PG 305 - V1.1
Oct. 2017 166

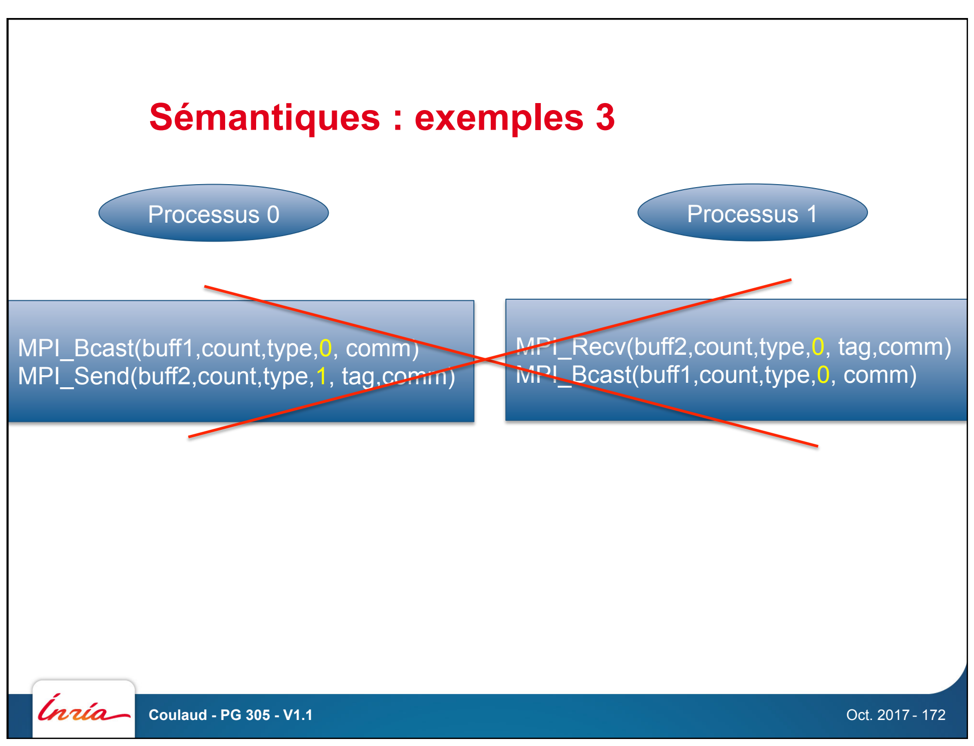

Scatter
One-to-all
Tous dans le groupe local
Un dans le groupe local
MPI_Scatter(sbuf, 1, MPI_INT, rbuf, 1, MPI_INT, root, intracomm)
Avant
Après
root
La taille est quelconque
Coulaud - PG 305 - V1.1
Oct. 2017 167


Extensions
Communication sur place
- Buffer de réception = buffer d'émission
- Argument spécial MPI_IN_PLACE au lieu du buffer de réception du root.
Sendbuffer Gather, recvBuffer Scatter
Extension avec V : fonctions MPI_Gatherv et MPI_Scatterv
Variantes vecteurs des fonctions Gather et Scatter
100
50
100
recvcount
Processeurs
root
stride
recvcount et stride sont des vecteurs
Coulaud - PG 305 - V1.1
Oct. 2017 - 168


MPI_AlltoAll
All-to-All
Tous dans le groupe local
Tous dans le groupe local
MPI_Alltoall(sbuf, 1, MPI_INT, rbuf, 1, MPI_INT, intracomm)
Avant
Après
0 0 0 0
1 1 1 1
2 2 2 2
3 3 3 3
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3
Coulaud - PG 305 - V1.1
Oct. 2017 - 169


Sémantiques : exemples 1
Un programme portable ne doit pas tenir compte du fait qu'une opération collective est synchronisante ou non.
Processus 0
Processus 1
MPI_Bcast(buff1,count,type,0, comm)
MPI_Bcast(buff2,count,type,1, comm)
MPI_Bcast(buff2,count,type,1, comm)
MPI_Bcast(buff1,count,type,0, comm)
Ordre des appels est important.
Coulaud - PG 305 - V1.1
Oct. 2017 - 170


Sémantiques : exemples 2
P0
MPI_Bcast(buff1,count,type,0, comm0)
MPI_Bcast(buff2,count,type,2, comm2)
P1
MPI_Bcast(buff1,count,type,1, comm1)
MPI_Bcast(buff2,count,type,0, comm0)
P2
MPI_Bcast(buff1,count,type,2, comm2)
MPI_Bcast(buff1,count,type,1, comm1)
comm2
comm0
comm1
Plus de deadlock
Coulaud - PG 305 - V1.1
Oct. 2017 - 171


Sémantiques : exemples 3
Processus 0
Processus 1
MPI_Bcast(buff1,count,type,0, comm)
MPI_Send(buff2,count,type,1, tag,comm)
MPI_Recv(buff2,count,type,0, tag,comm)
MPI_Bcast(buff1,count,type,0, comm)
Coulaud - PG 305 - V1.1
Oct. 2017 - 172

Communications collectives étendues

Communications collectives avec les intercommunicateurs
Deux modes

Simplex mode (ie. opérations sur la racine)

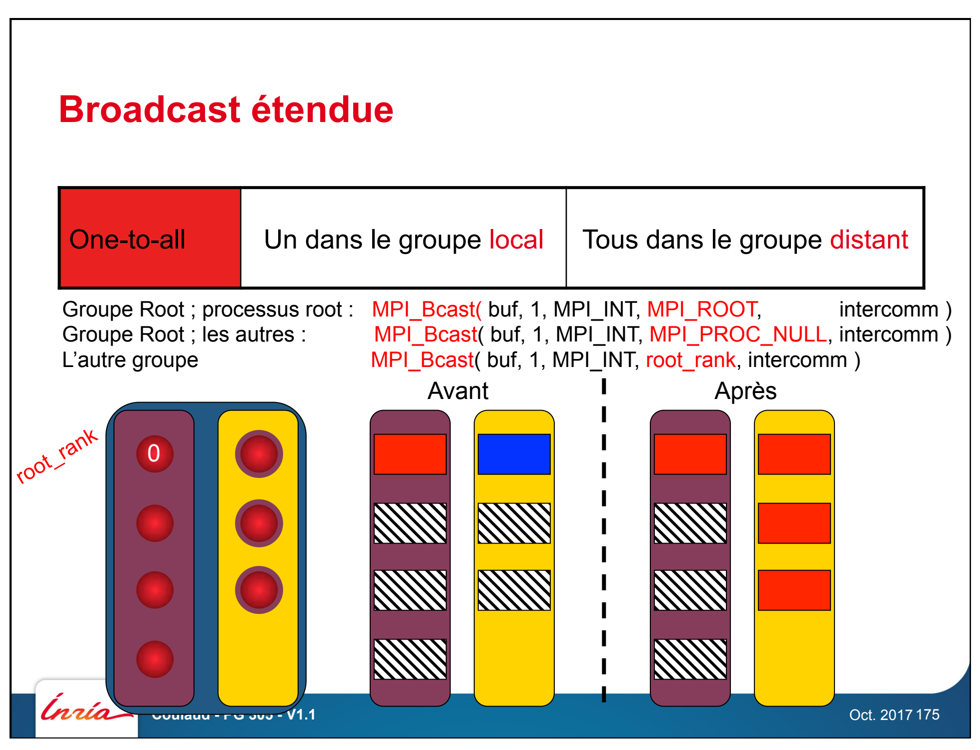
- Dans le groupe qui initie la communication
    - Utilise MPI_ROOT pour le processeur émetteur.
    - Les autres utilisent MPI_PROC_NULL
- Dans le groupe distant
  Tous utilisent le rang du processeur qui envoie dans le groupe distant.

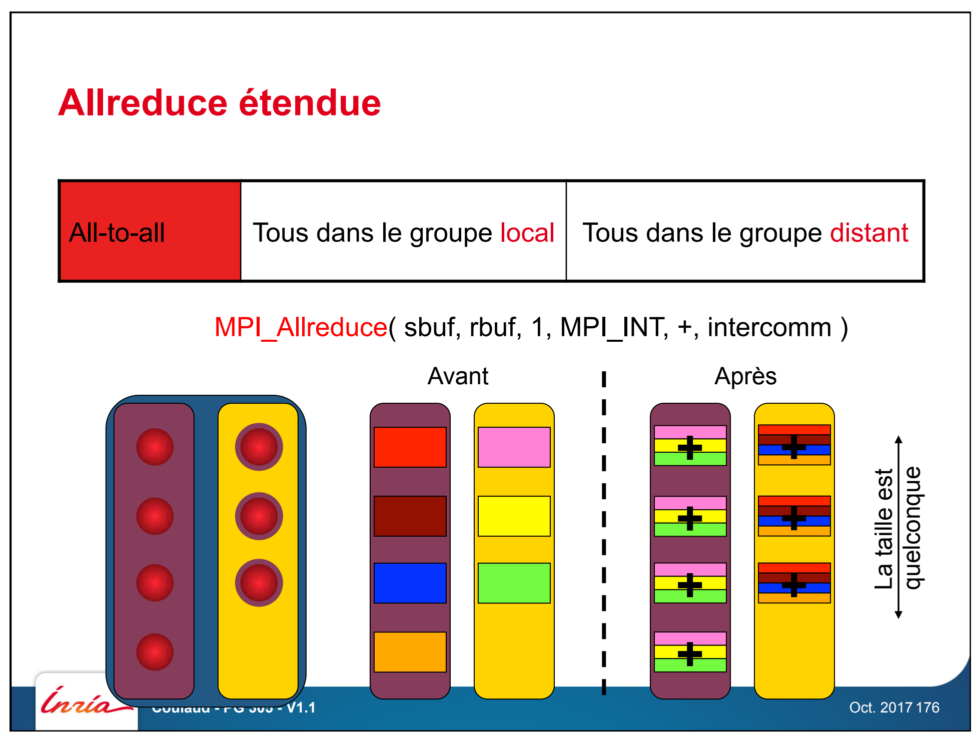
Duplex mode (ie. échange)

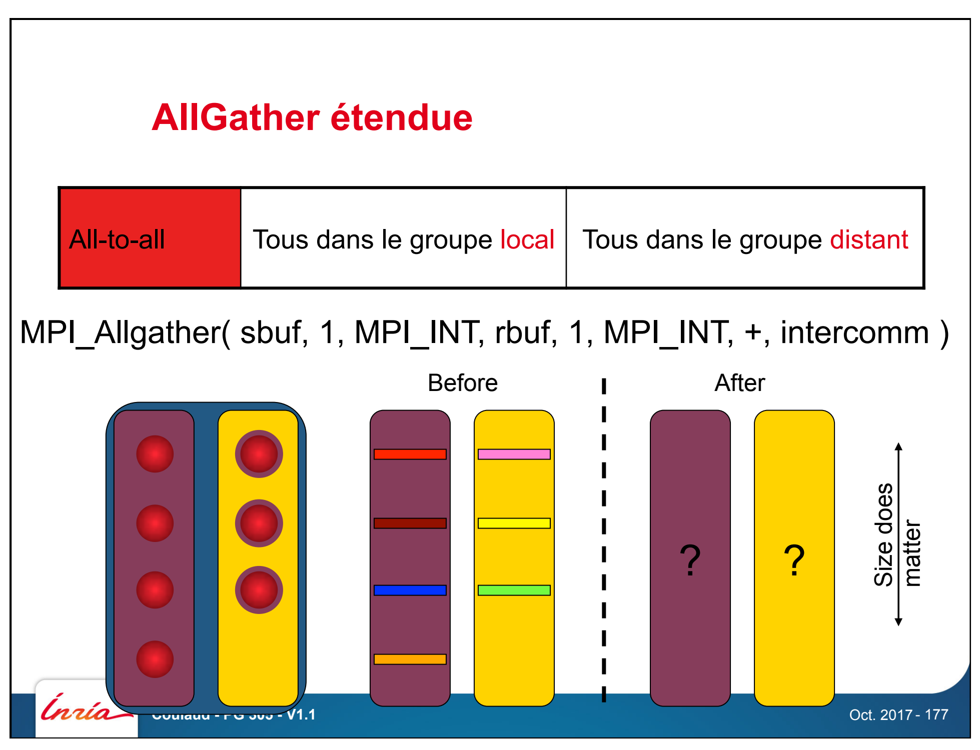
Données envoyées par le processus d'un groupe sont reçues par le processus de l'autre groupe et réciproquement.

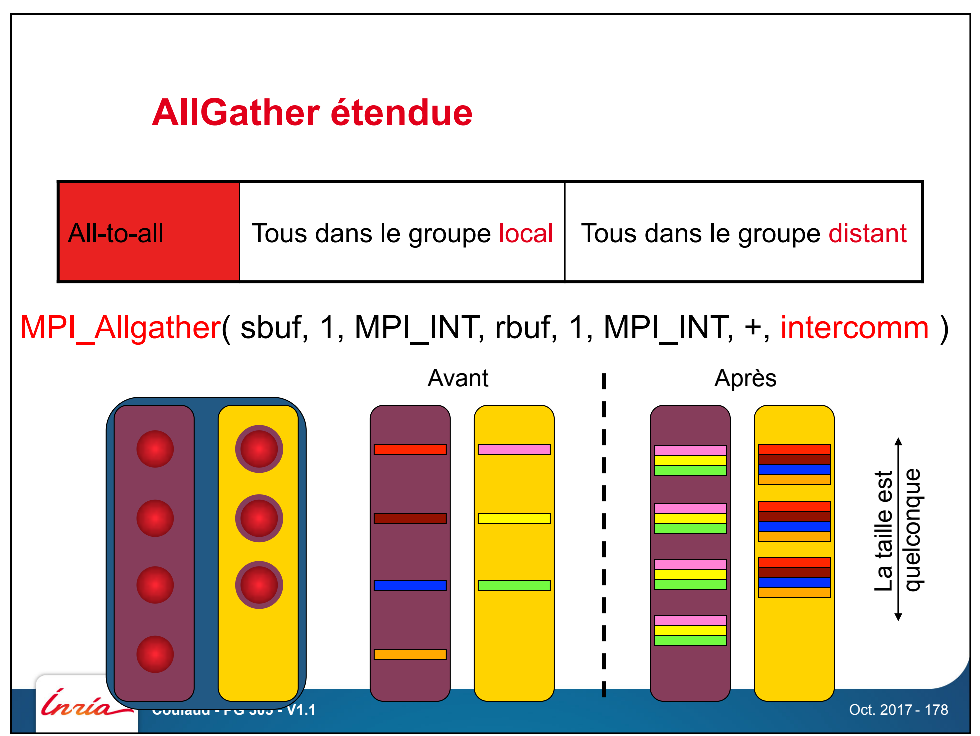
Opérations Collectives étendues

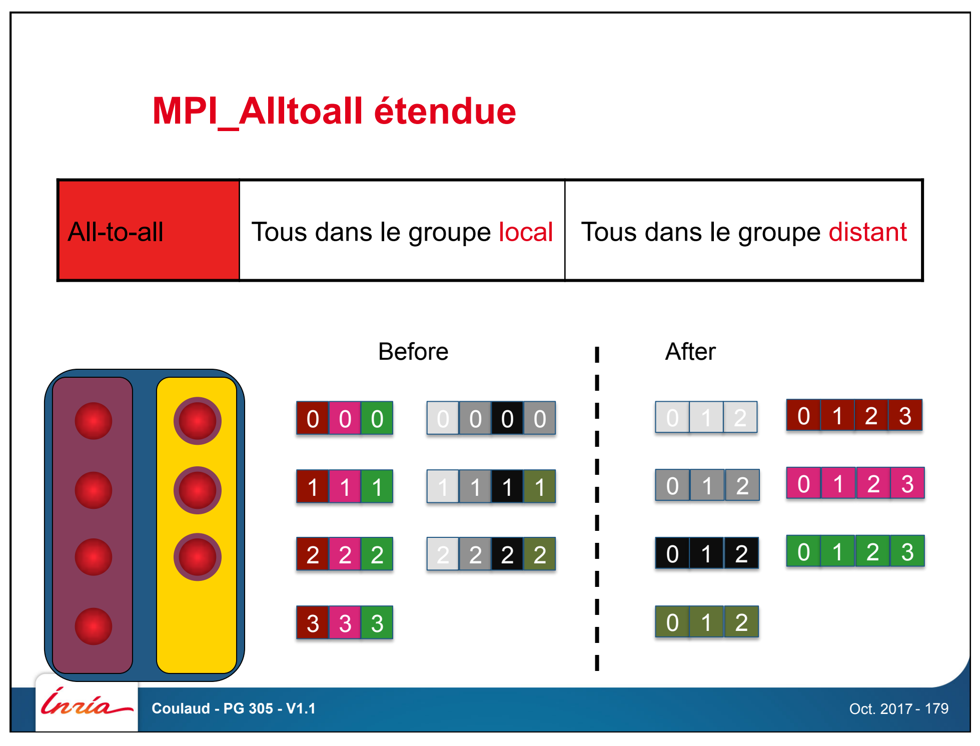
	Qui génère le résultat	Qui reçoit le résultat
One-to-all	Un dans le groupe local	Tous dans le groupe distant
All-to-one	Tous dans le groupe local	Un dans le groupe distant
All-to-all	Tous dans le groupe local	Tous dans le groupe distant
Autres	?	?

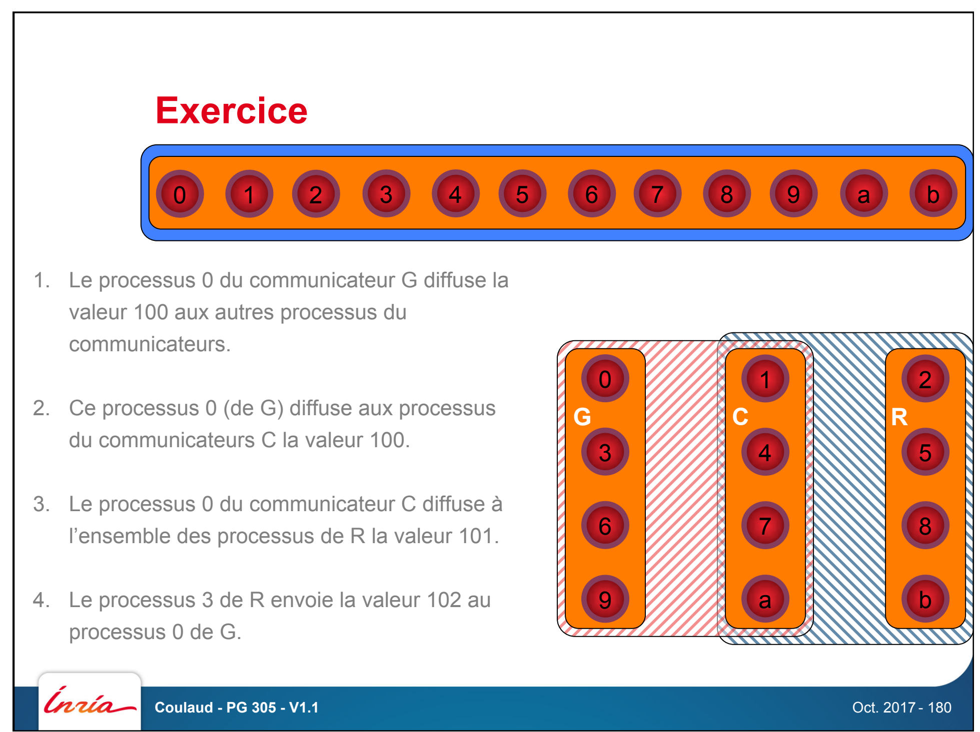

Broadcast étendue
One-to-all
Un dans le groupe local
Tous dans le groupe distant
Groupe Root ; processus root : MPI_Bcast(buf, 1, MPI_INT, MPI_ROOT, intercomm)
Groupe Root ; les autres : MPI_Bcast(buf, 1, MPI_INT, MPI_PROC_NULL, intercomm)
L'autre groupe MPI_Bcast(buf, 1, MPI_INT, root_rank, intercomm)
Avant
Après
root_rank
0
Oct. 2017 175


Allreduce étendue
All-to-all
Tous dans le groupe local
Tous dans le groupe distant
MPI_Allreduce(sbuf, rbuf, 1, MPI_INT, +, intercomm)
Avant
Après
La taille est quelconque
Oct. 2017 176


AllGather étendue
All-to-all
Tous dans le groupe local
Tous dans le groupe distant
MPI_Allgather(sbuf, 1, MPI_INT, rbuf, 1, MPI_INT, +, intercomm)
Before
After
?
?
Size does matter
V1.1
Oct. 2017 - 177


AllGather étendue
All-to-all
Tous dans le groupe local
Tous dans le groupe distant
MPI_Allgather(sbuf, 1, MPI_INT, rbuf, 1, MPI_INT, +, intercomm)
Avant
Après
La taille est quelconque
V1.1
Oct. 2017 - 178


MPI_Alltoall étendue
All-to-all
Tous dans le groupe local
Tous dans le groupe distant
Before
After
Coulaud - PG 305 - V1.1
Oct. 2017 - 179


Exercice
0 1 2 3 4 5 6 7 8 9 a b
1. Le processus 0 du communicateur G diffuse la valeur 100 aux autres processus du communicateurs.
2. Ce processus 0 (de G) diffuse aux processus du communicateurs C la valeur 100.
3. Le processus 0 du communicateur C diffuse à l'ensemble des processus de R la valeur 101.
4. Le processus 3 de R envoie la valeur 102 au processus 0 de G.
G
C
R
Coulaud - PG 305 - V1.1
Oct. 2017 - 180

MPI 3.0

Communications collectives non bloquantes

Communications non bloquantes

- Eviter un deadlock
- Recouvrir communication/calcul

Communications collectives

- Fonctions optimisées

Communications collectives non bloquantes

- Combiner les deux avantages

MPI 3.0

Communications collectives non bloquantes

Sémantique

- Retour rapide (sans terminaison)
- Pas de progression garantie

Pour la terminaison

- Tester la terminaison (non bloquant) : Test, Testany, Testall, Testsome
- Attendre la terminaison : Wait, Waitany, Waitsome, Waitall
  Mettre le plus loin possible dans le code

MPI
3.0

Communications collectives non bloquantes

I pour Immédiat

Variante non bloquante de toutes les communications collectives

MPI_Ibcast(<bcast args>, MPI_Request *req)

Sémantique

- Retour indépendant de ce que la fonction doit faire
- Pas de garantie de la progression (dépend de l'implémentation)
- Terminaison classique (test, wait)
- Terminaison dans le désordre

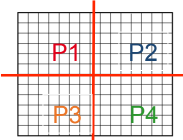
Restrictions:

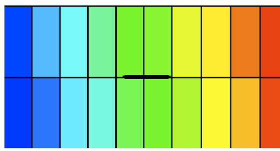
- Pas d'étiquette, matching dans l'ordre
- Send buffer ne doit pas être modifié
- Pas de MPI_Cancel
- Pas de matching avec des collectives bloquantes

Topologies de processus

Topologies de processus

Dans les méthodes de partitionnement de domaine sur des structures régulières 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 de type :

- Cartésien :
  - Chaque processus est défini dans une grille de processus ;
  - la grille peut être périodique ou non ;
  - les processus sont identifiés par leurs coordonnées dans la grille.
- Graphe :

  Généralisation à des topologies plus complexes.

Topologies de processus

Une topologie peut être ajoutée uniquement à un intra communicateur.

Une topologie

- Permet de fournir un mécanisme pour nommer les processus qui s'associe au schéma de communications ;
- Peut aider le système d'exécution à placer en adéquation la topologie des processus sur la topologie des processeurs ;
- Permet d'optimiser les communications ;
- Simplifier l'écriture du code.

Topologies cartésiennes

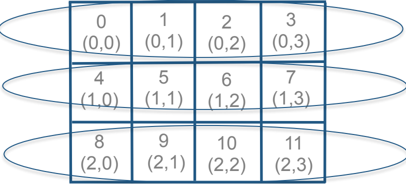
Les coordonnées commencent à 0

Numérotation « row-major »

Exemples

Grille 3x4

non périodique sur les colonnes, périodique sur les lignes



Topologies cartésiennes

Une topologie cartésienne est définie lorsqu'un ensemble de processus appartenant à un communicateur donné comm ancien appelle le sous-programme

MPI_Cart_create(comm, ndims, dims,period, reorder, GRIDcomm)

IN	comm	communicateur
IN	ndims	nombre de dimensions de la grille
IN	dims	tableaux des dimensions de la grille
IN	period	tableaux spécifiant la périodicité
IN	reorder	si true renumérotation des processus
IN	CARTcomm	communicateur avec la topologie cartésienne

Si reorder = false le rang des processus est le même dans comm et dans GRIDcomm.

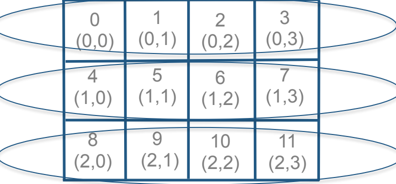
= true l'implémentation MPI choisit l'ordre des processus.

Numérotation row-major

Topologie exemple

Construire la grille cartésienne comportant 3 lignes et 4 colonnes, périodique dans la deuxième dimension.

dims[0]= 3 ;
dims[1]= 4 ;
periods[0]= false ;
periods[1]= true ;
reorder = false ;



MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, grid2D)

Fonctions associées à la grille

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_cart, coords, rang)

- MPI_Cart_coords renvoie les coordonnées d'un processus de rang donné ;

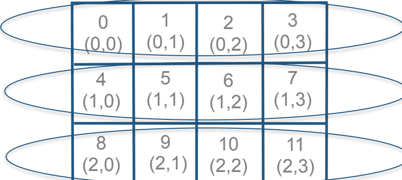
MPI_Cart_coords(comm_cart, rank, dim, coords)

- MPI_Cart_shift donne le processeur avant et après dans la direction d

MPI_Cart_shift(comm_cart, d, pas, rang_precedent, rang_suivant)

pas : longueur du shift - et +
d : direction du shift
Si on sort de la topologie MPI_PROC_NULL est retourné.

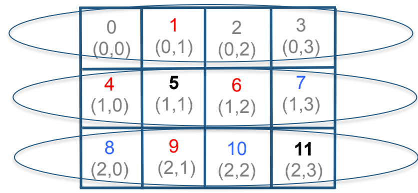
Exemple




```
dims[0]= 3 ;
dims[1]= 4 ;
periods[0]= 0 ;
periods[1]= 1 ;
reorder   = 1 ;

MPI_Cart_create(MPI_COMM_WORLD,2,dims,periods,reorder,&grid_2D);

MPI_Cart_coords(grid_2D,rank,2,coords) ;
MPI_Cart_rank( grid_2D, coords, &rank2d );

MPI_Cart_shift(grid_2D, 0, 1, &dessous, &dessus);
MPI_Cart_shift(grid_2D, 1, 1, &avant, &apres);
```


Exemple

Rang **5**. Mes coordonnées (1,1).
Mes voisins
avant 4 et après 6 ;
dessus 1 dessous 9
Rang **11.** Mes coordonnées (2,3).
Mon voisin
avant 10 et après 8 ;
dessus 7 dessous -2



Dans cette implémentation MPI_PROC_NULL = -2

La valeur MPI_PROC_NULL peut être utilisée pour une source ou pour une destination dans une communication.

Une communication avec MPI_PROC_NULL n'a aucun effet.

Un envoi à MPI_PROC_NULL réussit et retourne dès que possible.

Une réception sur MPI_PROC_NULL réussit et retourne dès que possible sans modification du buffer de réception.
Quand une réception avec source = MPI_PROC_NULL est exécutée l'objet status est positionné à

source = MPI_PROC_NULL,
tag   = MPI_ANY_TAG,
count   = 0.

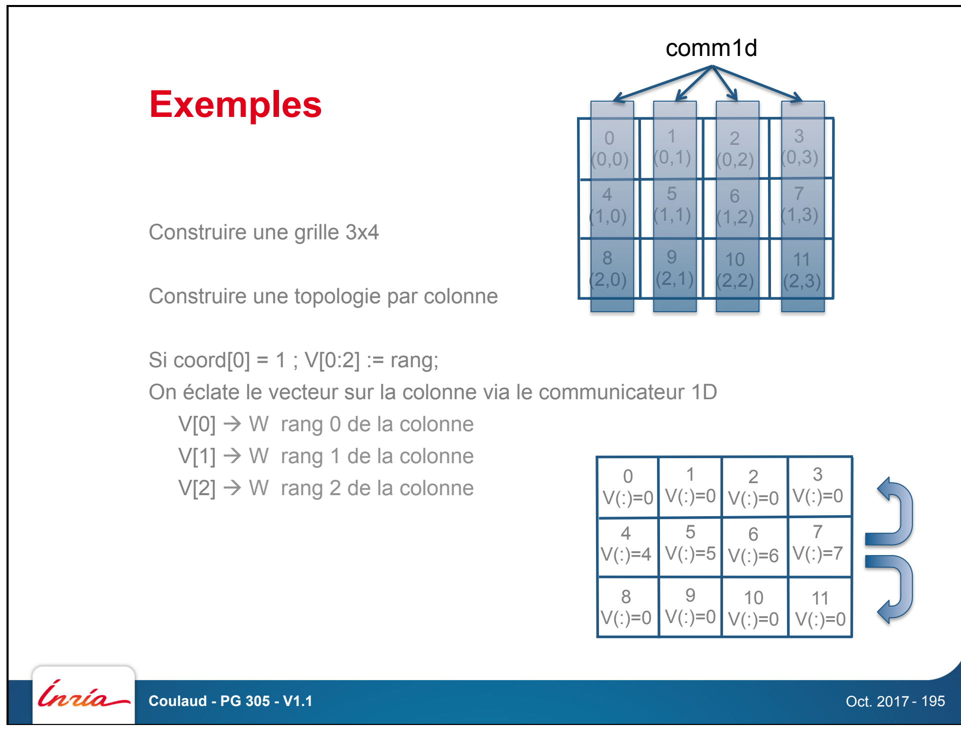
Partitionnement cartésien

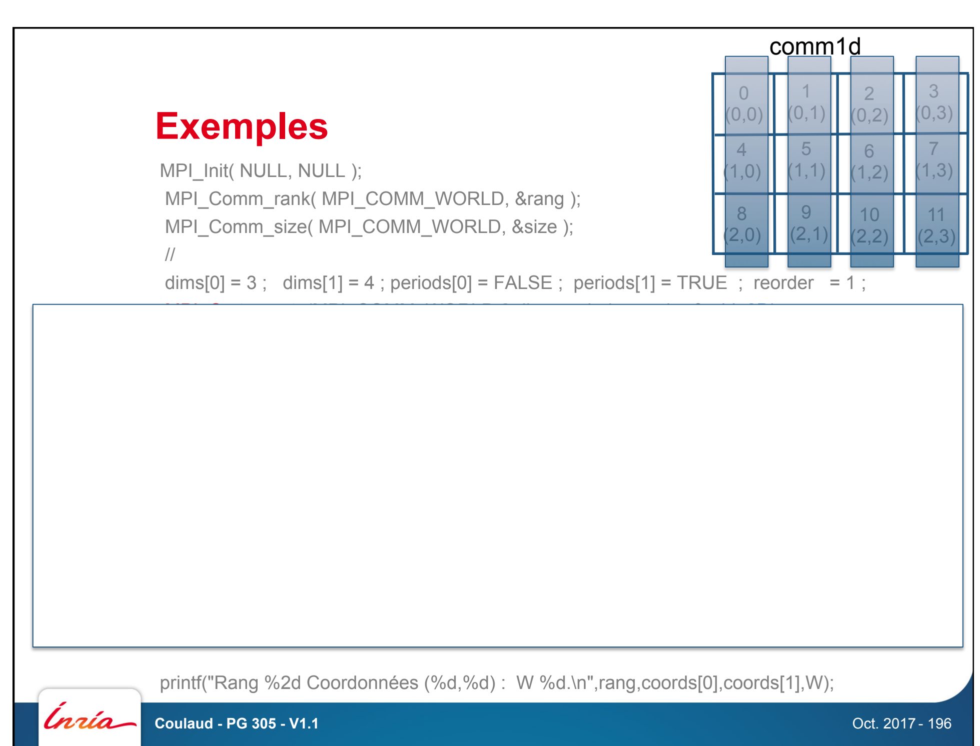
Souvent on souhaite exécuter une opération sur une partie de la topologie :
- les lignes (si grille 2D), les plans (si grille 3D)

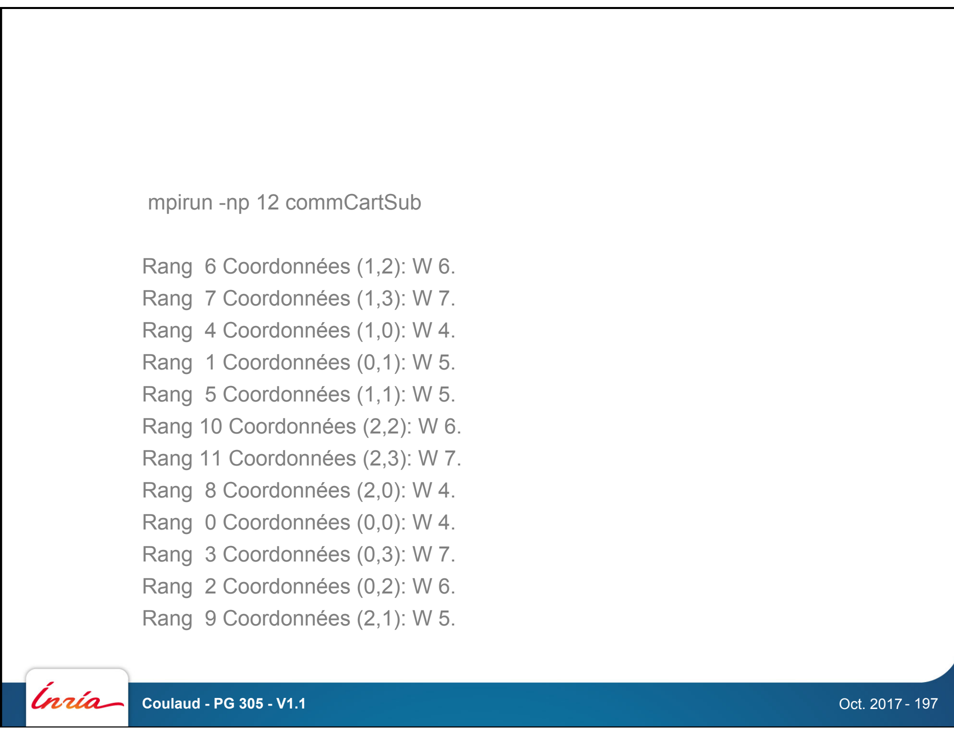
Découper une grille 3D en plan
- Un communicateur par plan (autant de communicateurs que de plans)
- Des opérations collectives seulement sur une tranche

MPI_Cart_sub(comm, remain_dims, subcom)

IN	comm	communicateur cartésien
IN	remain_dims	tableaux précisant les dimensions de la grille que l'on garde
OUT	subcomm	communicateur avec la topologie cartésienne

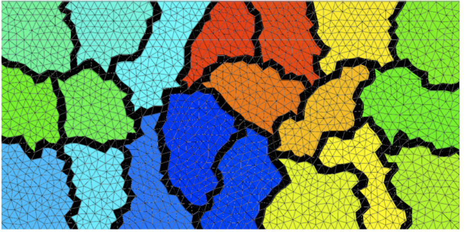

Exemples
comm1d
0 (0,0)
1 (0,1)
2 (0,2)
3 (0,3)
4 (1,0)
5 (1,1)
6 (1,2)
7 (1,3)
8 (2,0)
9 (2,1)
10 (2,2)
11 (2,3)
Construire une grille 3x4
Construire une topologie par colonne
Si coord[0] = 1 ; V[0:2] := rang;
On éclate le vecteur sur la colonne via le communicateur 1D
V[0] → W rang 0 de la colonne
V[1] → W rang 1 de la colonne
V[2] → W rang 2 de la colonne
0 V(:)=0
1 V(:)=0
2 V(:)=0
3 V(:)=0
4 V(:)=4
5 V(:)=5
6 V(:)=6
7 V(:)=7
8 V(:)=0
9 V(:)=0
10 V(:)=0
11 V(:)=0
Coulaud - PG 305 - V1.1
Oct. 2017 - 195


comm1d
Exemples
0 (0,0)
1 (0,1)
2 (0,2)
3 (0,3)
4 (1,0)
5 (1,1)
6 (1,2)
7 (1,3)
8 (2,0)
9 (2,1)
10 (2,2)
11 (2,3)
MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &rang);
MPI_Comm_size(MPI_COMM_WORLD, &size);
//
dims[0] = 3 ; dims[1] = 4 ; periods[0] = FALSE ; periods[1] = TRUE ; reorder = 1 ;
printf("Rang %2d Coordonnées (%d,%d) : W %d.\n",rang,coords[0],coords[1],W);
Coulaud - PG 305 - V1.1
Oct. 2017 - 196



Graphe de processus

Dans beaucoup d'applications, la décomposition de domaine n'est pas cartésienne mais on a un graphe.



Graphe de processus

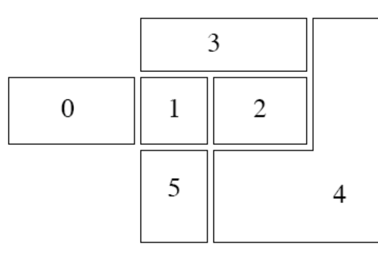
MPI_Graph_create permet de définir une topologie de type graphe

MPI_Graph_create(comm, Nbnodes,index, edges, reorder, comm_graph)

IN	comm	communicateur
IN	Nbnodes	nombre de nœuds dans le graphe
IN	index	décrit le degré des nœuds
IN	edges	décrit les arêtes associées au nœud
IN	reorder	true ou réordonne les processus ; false on garde l'ordre de comm
OUT	comm_graph	communicateur avec la topologie graphe

Nbnodes = nombre de processus dans le communicateur *comm*.

Graphe de processus - exemple



Numéro de processus	liste_voisins
0	1
1	0,5,2,3
2	1,3,4
3	1,2,4
4	3,2,5
5	1,4

index = (/ 1, 5, 8, 11, 14, 16 /)
en C : index [0] degré du noeud 0
index[i] – index[i-1] degré du noeud i

edges = (/ 1, **0,5,2,3**, 1,3,4, **1,2,4**, 3,2,5, **1,4** /)
la liste des voisins
- du noeud 0 est stockée dans edge[j] ; 0≤ j ≤ index[1]
- du noeud i est stockée dans edge[j] ; index[i-1] ≤ j ≤ index[i]

Graphe de processus

Deux autres fonctions sont utiles pour connaître

- Le nombre de voisins, *nneighbors,* pour un processus donné

int MPI_Graph_neighbors_count(MPI_Comm *comm*, **int** *rank*, **int** **nneighbors* **)**

- La liste des voisins pour un processus donné

int MPI_Graph_neighbors(MPI_Comm *comm*, **int** *rank*, **int** *maxneighbors*, **int** **neighbors* **)**

maxneighbors : taille du tableau neighbors
comm : intracommunicateur avec une topologie de graphe.

Gestion dynamique de processus

Gestion dynamique de processus

Ajout de processus à un code qui s'exécute

- Dans le cadre d'un algorithme i.e. branch and bound ;
- Quand des ressources supplémentaires sont disponibles ;
- Les codes master-slave codes quand le maitre est lancé en 1er et doit demander à l'environnement le nombre de processus à créer.

Rejoindre une application qui s'exécute

- Client-server or peer-to-peer

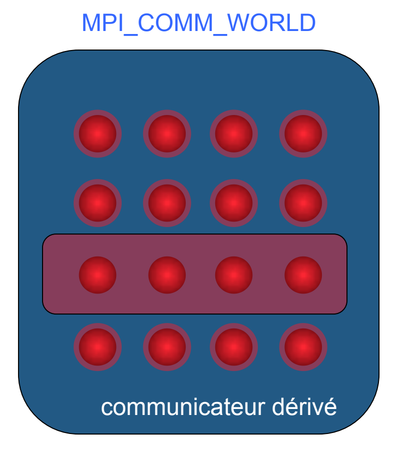
Supporter des défauts / défaillances

Approche statique de MPI-1

MPI_COMM_WORLD
Contient 16 processus

On ne peut utiliser qu'un sous ensemble de processus

Pas de processus externes



Désavantage du modèle statique

Impossible d'ajouter des processus
Impossible d'enlever des processus

Si un processus échoue ou disparaît, tous les communicateurs auxquels il appartient deviennent invalides.

Conséquences :
- ➔ La tolérance aux pannes n'est pas possible
- ➔ Pas de connexion/ déconnexion d'outil de visualisation
- ➔ ….

MPI-2

Ajout du support pour les processus dynamiques

Création de nouveaux processus à la volée se connectant à des processus déjà existants

Ne standardise pas la communication entre implémentation de MPI

MPI Interopérable (IMPI) est créé pour cela

Questions

Comment ajoute-t-on des processus à un code qui s'exécute ?

Comment peut-on gérer une défaillance du processus?

Comment peut-on établir des connexions entre deux codes lancés indépendamment et simultanément par un code MPI ?

LANCEMENT DE PROCESSUS

Fonctions Spawn

MPI_COMM_SPAWN

- Démarre un ensemble de nouveaux processus avec la même ligne de commande ;
- Single Process Multiple Data

MPI_COMM_SPAWN_MULTIPLE

- Démarre un ensemble de nouveaux processus avec différentes lignes de commande ;
- Différents exécutables et / ou différents arguments ;
- Multiple Processes Multiple Data

Création des processus - API

MPI_Comm_spawn(command, argv, nbprocs, info, root, comm, intercomm, errcodes)

IN	command	nom du programme à lancer
IN	argv	argument du programme
IN	nbprocs	nombre de processus à lancer
IN	info	un ensemble de pair précisant comment, où lancer le programme
IN	root	rang du processus qui va lancer les codes
IN	comm	intracommunicateur contenant les processus qui exécutent la fonction
OUT	intercomm	intercommunicateur entre le goupe de parents et le groupe d'enfants
OUT	errcodes	tableau d'erreur (une par processus)

MPI_Info

ToDo

Sémantique du Spawn

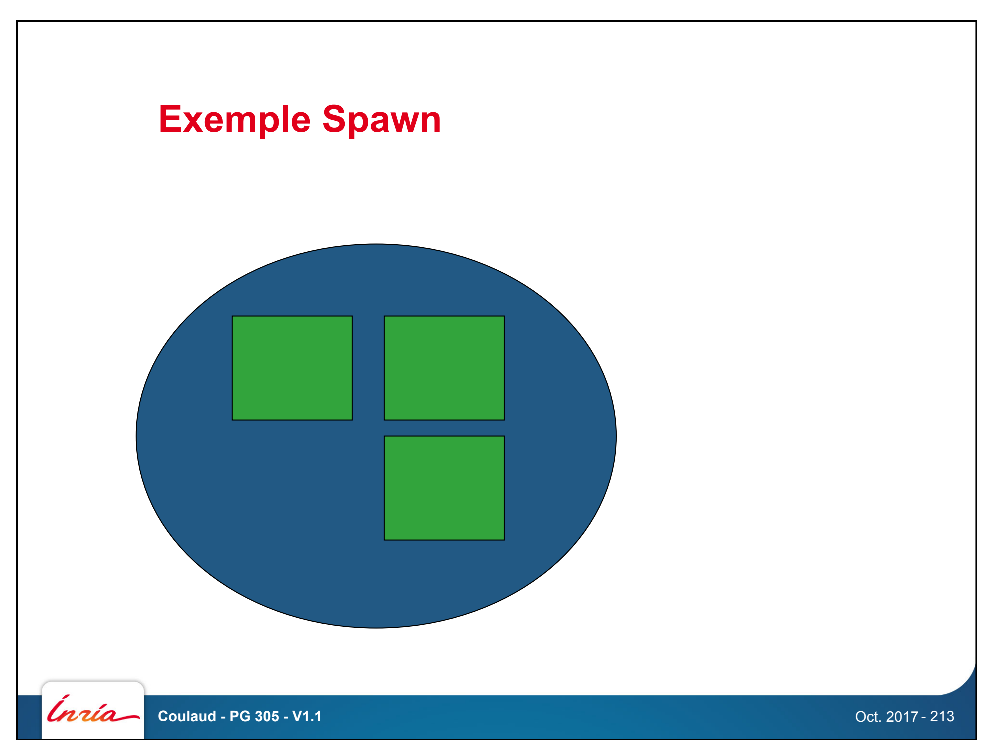
Le groupe de parents appelle collectivement la fonction spawn :
- Lance un nouvel ensemble de processus
- Les processus enfants deviennent des jobs

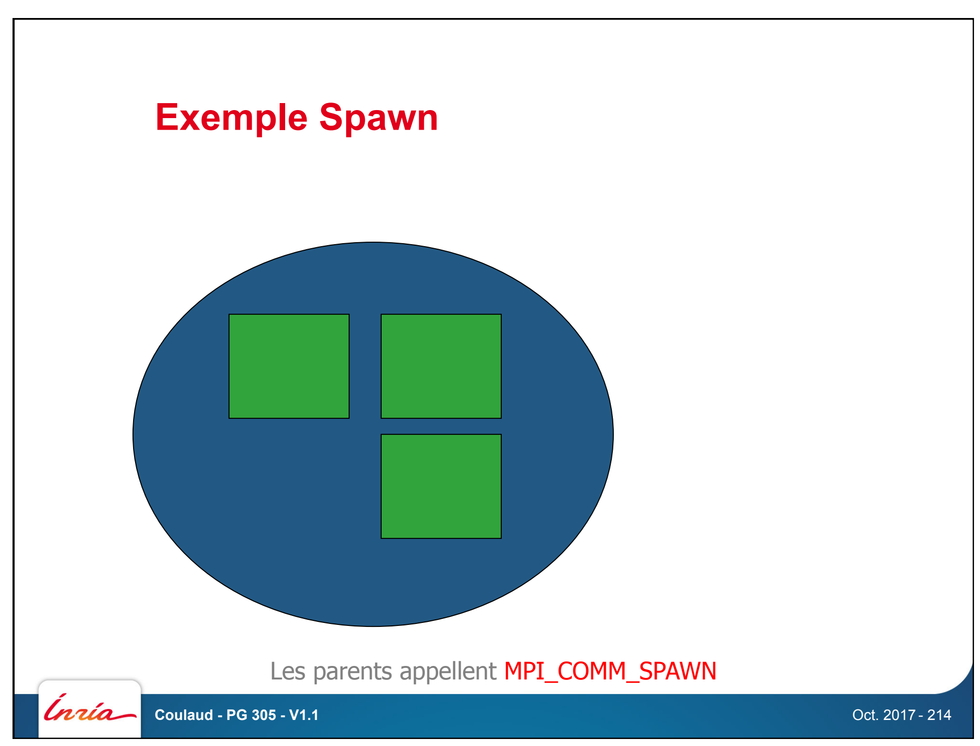
Un **intercommunicator** est créé entre les parents et les enfants

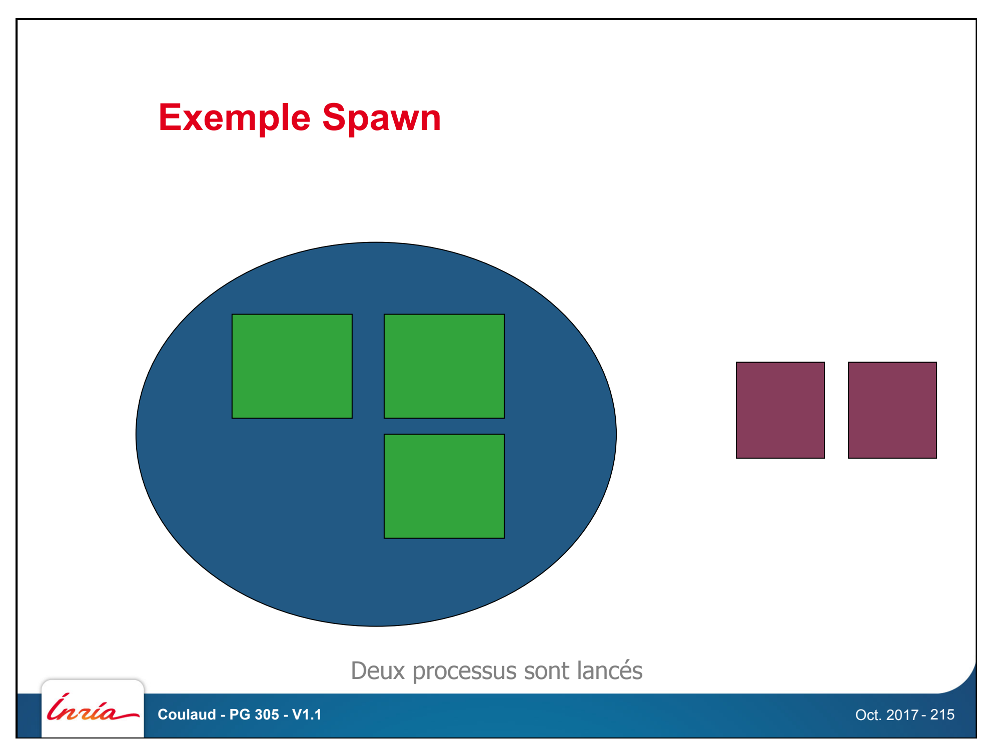
Parents et enfants peuvent alors utiliser MPI pour échanger des messages

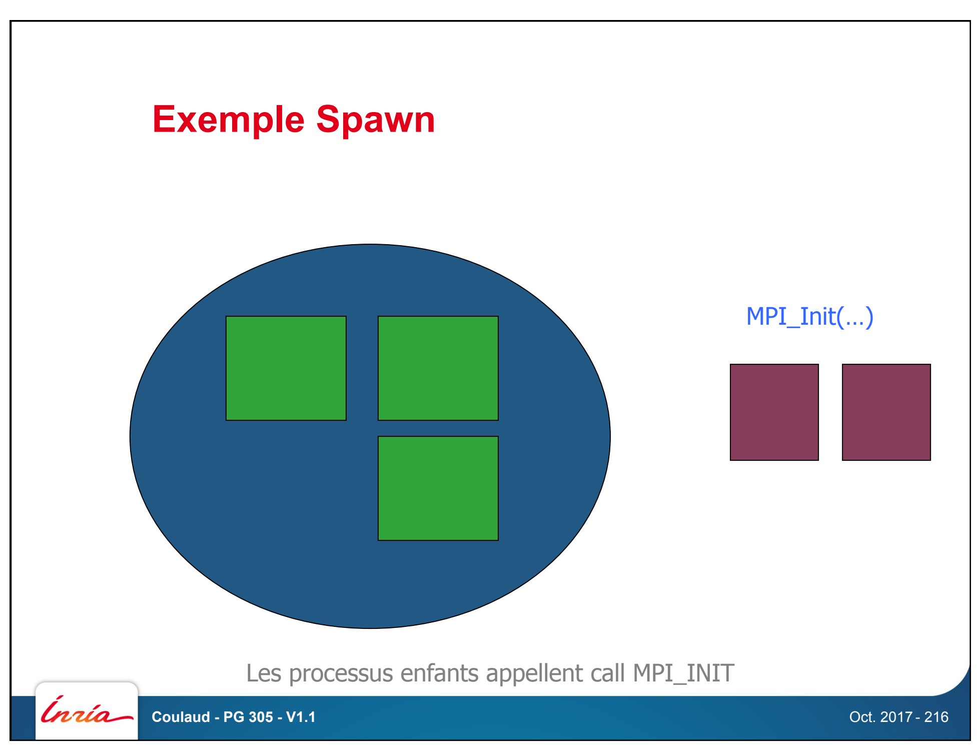
MPI_UNIVERSE_SIZE : précise le nombre de processus que l'on peut lancer en une fois.

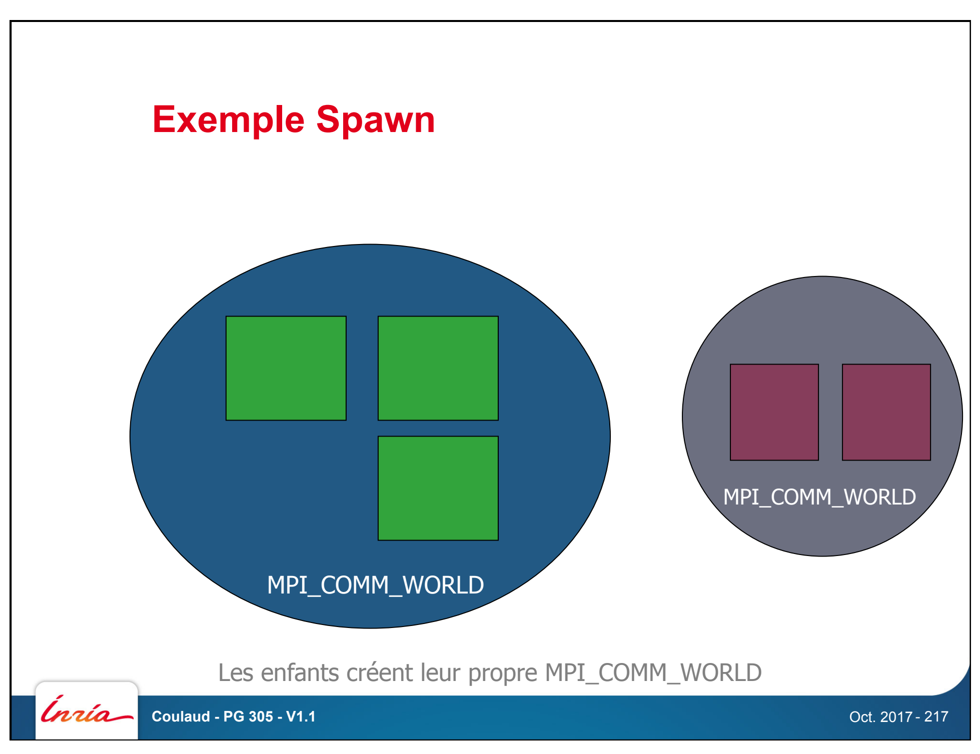
MPI_UNIVERSE_SIZE - (nombre de processus dans MPI_COMM_WORLD)

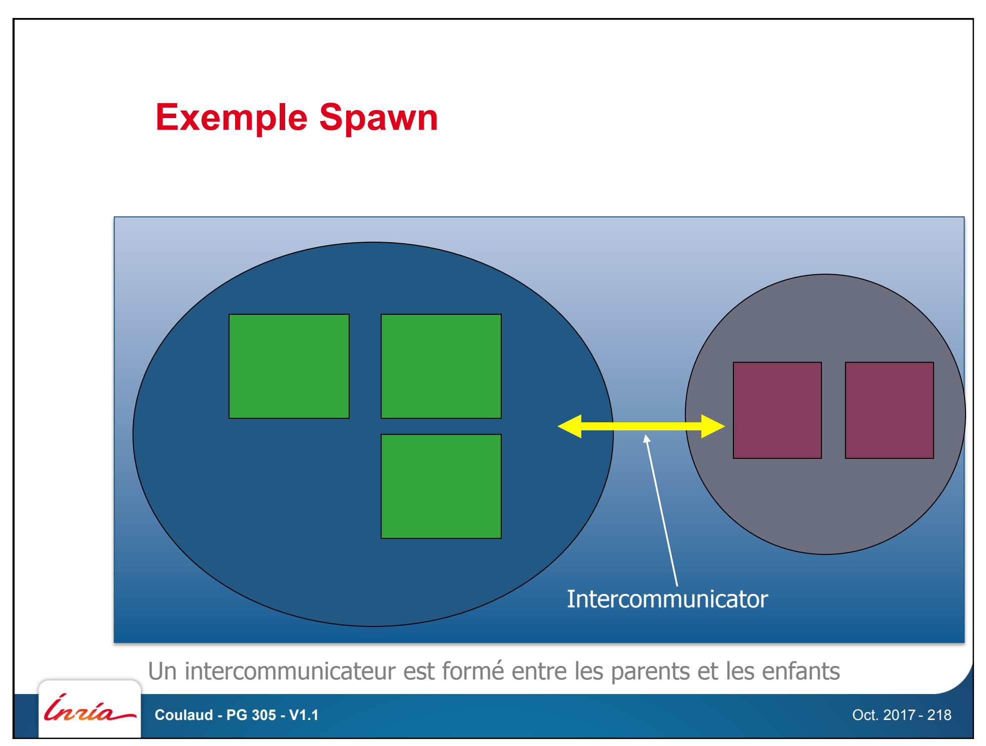

Exemple Spawn
Coulaud - PG 305 - V1.1
Oct. 2017 - 213

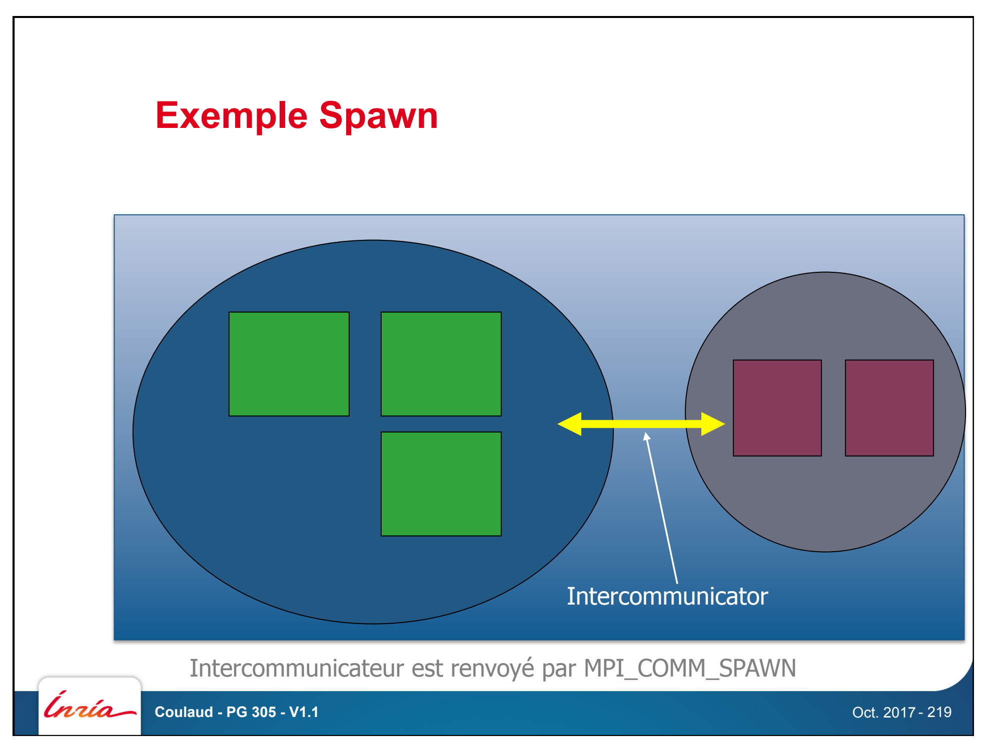

Exemple Spawn
Les parents appellent MPI_COMM_SPAWN
Coulaud - PG 305 - V1.1
Oct. 2017 - 214

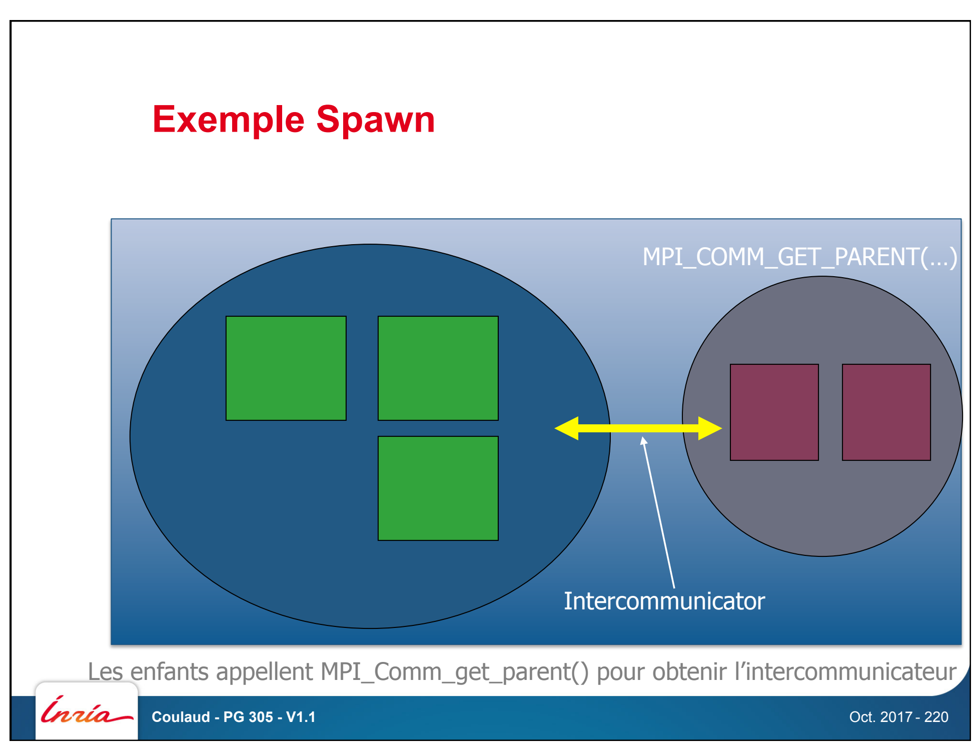

Exemple Spawn
Deux processus sont lancés
Coulaud - PG 305 - V1.1
Oct. 2017 - 215


Exemple Spawn
MPI_Init(...)
Les processus enfants appellent call MPI_INIT
Coulaud - PG 305 - V1.1
Oct. 2017 - 216


Exemple Spawn
MPI_COMM_WORLD
MPI_COMM_WORLD
Les enfants créent leur propre MPI_COMM_WORLD
Coulaud - PG 305 - V1.1
Oct. 2017 - 217


Exemple Spawn
Intercommunicator
Un intercommunicateur est formé entre les parents et les enfants
Coulaud - PG 305 - V1.1
Oct. 2017 - 218


Exemple Spawn
Intercommunicator
Intercommunicateur est renvoyé par MPI_COMM_SPAWN
Coulaud - PG 305 - V1.1
Oct. 2017 - 219


Exemple Spawn
MPI_COMM_GET_PARENT(...)
Intercommunicator
Les enfants appellent MPI_Comm_get_parent() pour obtenir l'intercommunicateur
Coulaud - PG 305 - V1.1
Oct. 2017 - 220

Exemple Maitre / Esclave

Exemple simple de programme (à la PVM)

- L'utilisateur lance un seul programme maitre ;
- Le processus maitre lance les esclaves ;
- Maitre et esclaves échangent des données ; les esclaves travaillent
- Le maitre rassemble les résultats
- Le maitre affiche les résultats
- Tous les processus terminent

Exemple : Master / Slave

Maitre

```
MPI_Init(...)
MPI_Spawn(..., slave,..., &intercomm,...);

for (i=0; i < size; ++i)
    MPI_Send(work, ...,i, ...
             &intercomm, ...);
for (i=0; i < size; ++i)
    MPI_Recv(presults,...intercomm,...)
calc_and_display_result(...)

MPI_Finalize()
```

Esclave

```
MPI_Init(...)
MPI_Comm_get_parent (&intercomm)

MPI_Recv(work,..., intercomm)
result = do_something(work)

MPI_Send(result,..., intercomm)


MPI_Finalize()
```

Notion de « connecté »

« Deux processus sont connectés s’il y a un chemin les reliant soit directement soit indirectement »

- E.g., processus appartenant au même communicateur
- Parents et enfants issus d’un SPAWN sont connectés

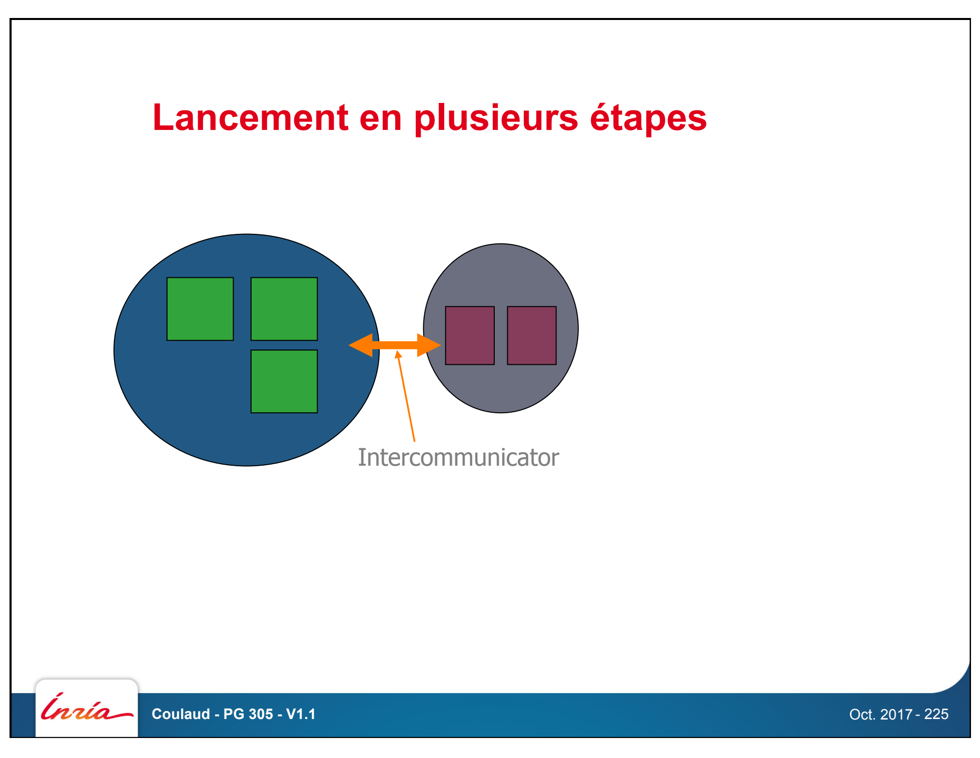
Connectivité est transitive

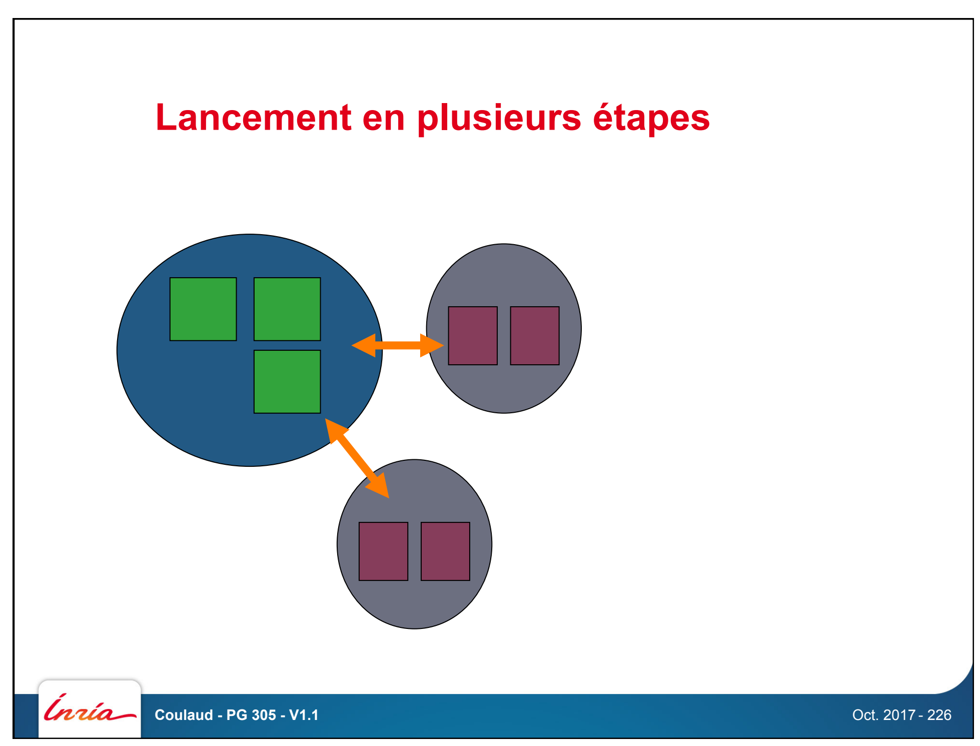
- Si A est connecté à B, et B est connecté à C
- A est connecté à C

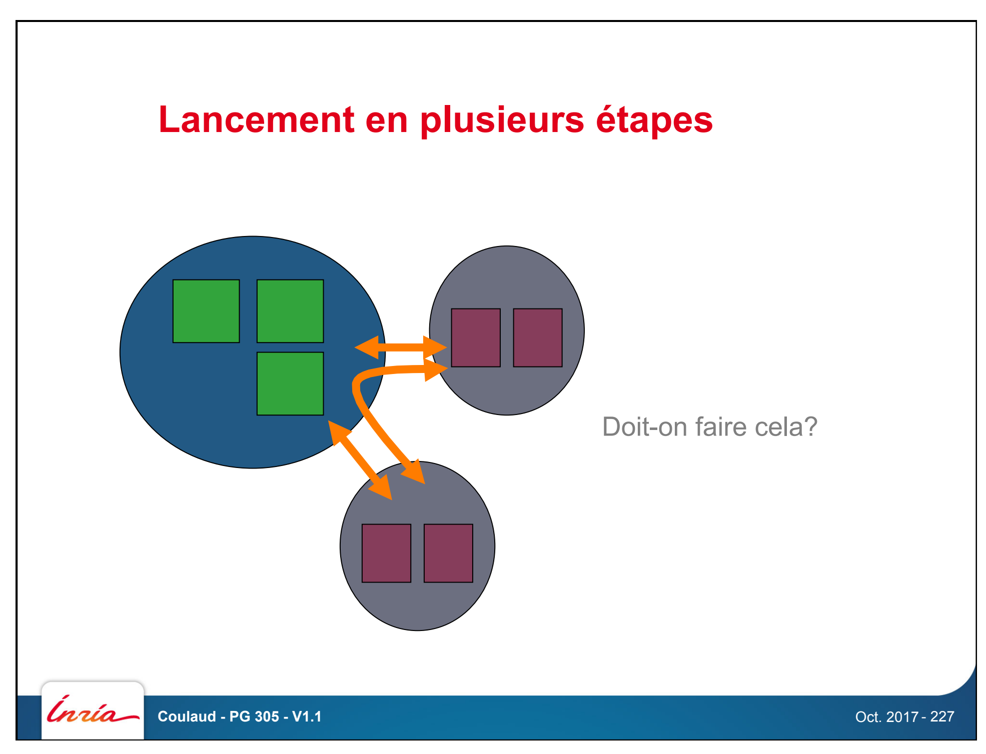
Lancement en plusieurs étapes

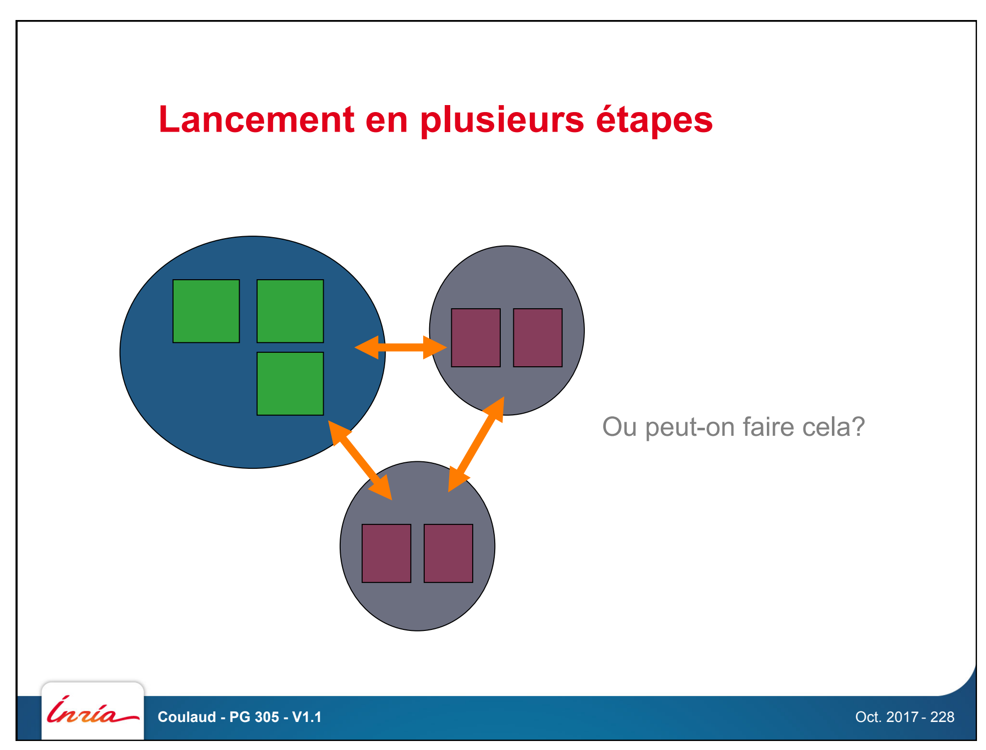
Que se passe-t-il si on lance plusieurs codes ?

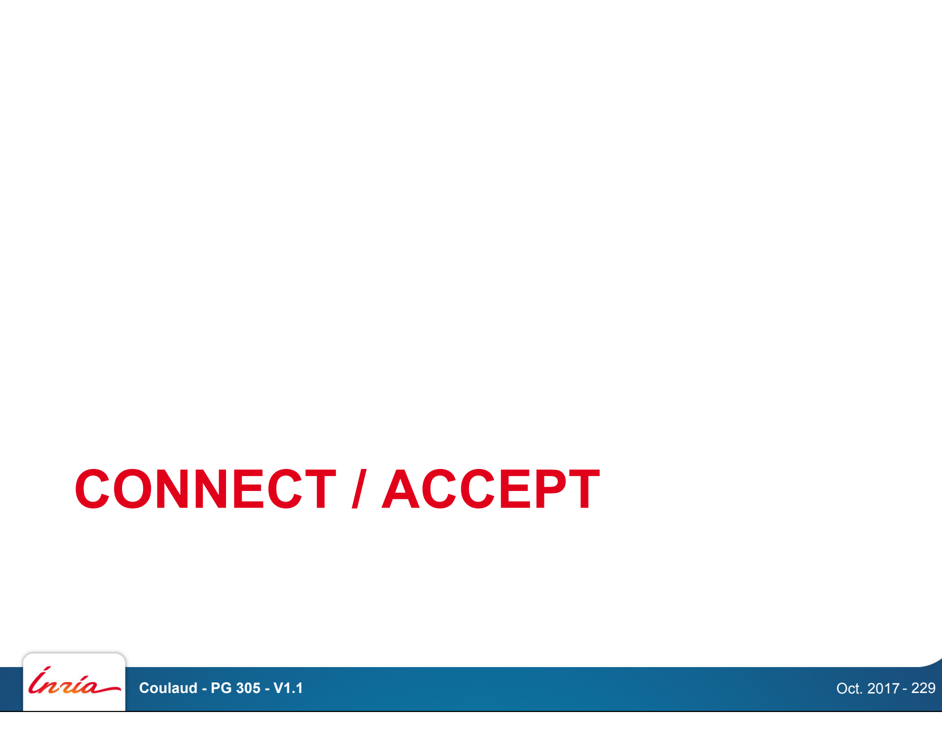
- Les enfants peuvent-il communiquer directement ?
- Ou doivent-ils communiquer en passant par leur parent commun ?


Lancement en plusieurs étapes
Intercommunicator
Coulaud - PG 305 - V1.1
Oct. 2017 - 225


Lancement en plusieurs étapes
Coulaud - PG 305 - V1.1
Oct. 2017 - 226


Lancement en plusieurs étapes
Doit-on faire cela?
Coulaud - PG 305 - V1.1
Oct. 2017 - 227


Lancement en plusieurs étapes
Ou peut-on faire cela?
Coulaud - PG 305 - V1.1
Oct. 2017 - 228



Etablissement des communications

MPI-2 possède une abstraction comme les « sockets TCP »

- Processus peut accepter et connecter des connexions venant d'autres processus
- Interface client-server

MPI_COMM_CONNECT

MPI_COMM_ACCEPT

Etablissement des communications

Comment le client trouve le serveur ?

- Les sockets TCP utilisent l'adresse IP et un port ;
- Qu'utilise-t-on avec MPI ?

Utilise le service de nom MPI

- Serveur ouvre un « port » MPI ;
- Serveur attribue un nom public au port ;
- Client cherche le nom public ;
- Client obtient le port à partir du nom public ;
- Client se connecte au port.

Coté Serveur

Ouverture et fermeture d'un port

- MPI_OPEN_PORT(info, port_name)
- MPI_CLOSE_PORT(port_name)

Avec char port_name[MPI_MAX_PORT_NAME];
info = MPI_INFO_NUL

Publier le nom du port

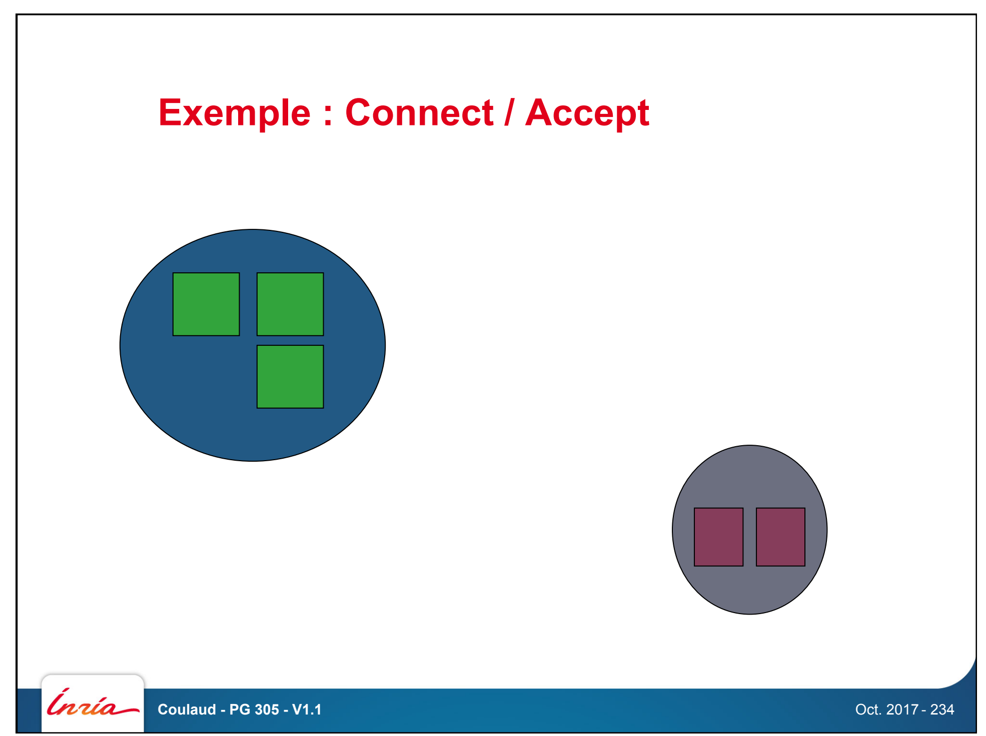
- MPI_PUBLISH_NAME(service_name, info, port_name)
- MPI_UNPUBLISH_NAME(service_name, info, port_name)

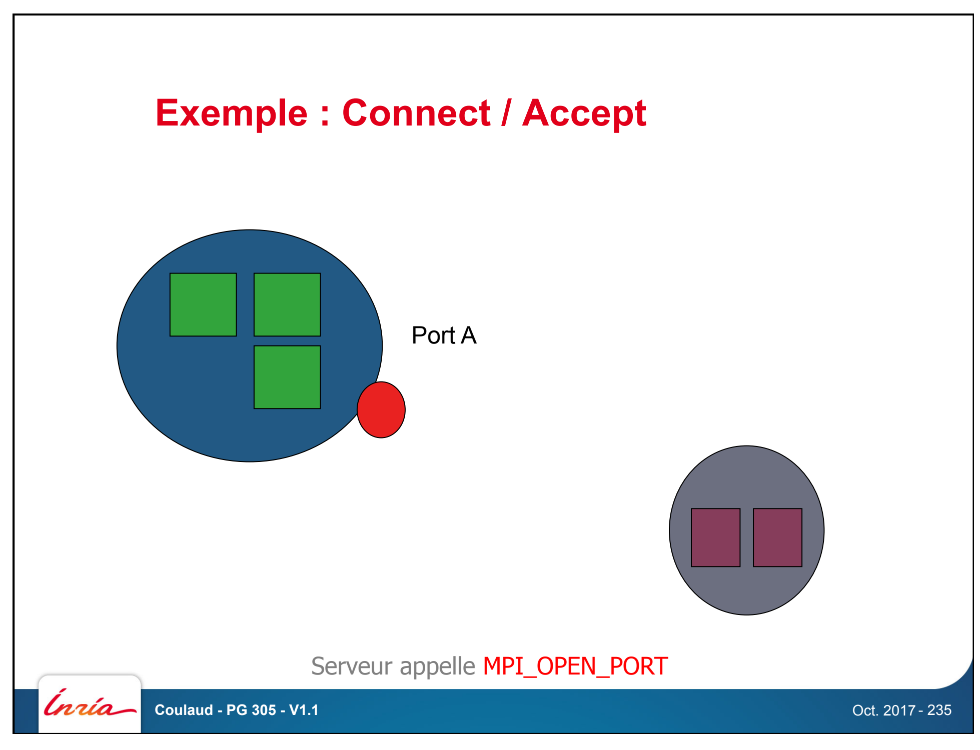
Accepter un connexion rentrante

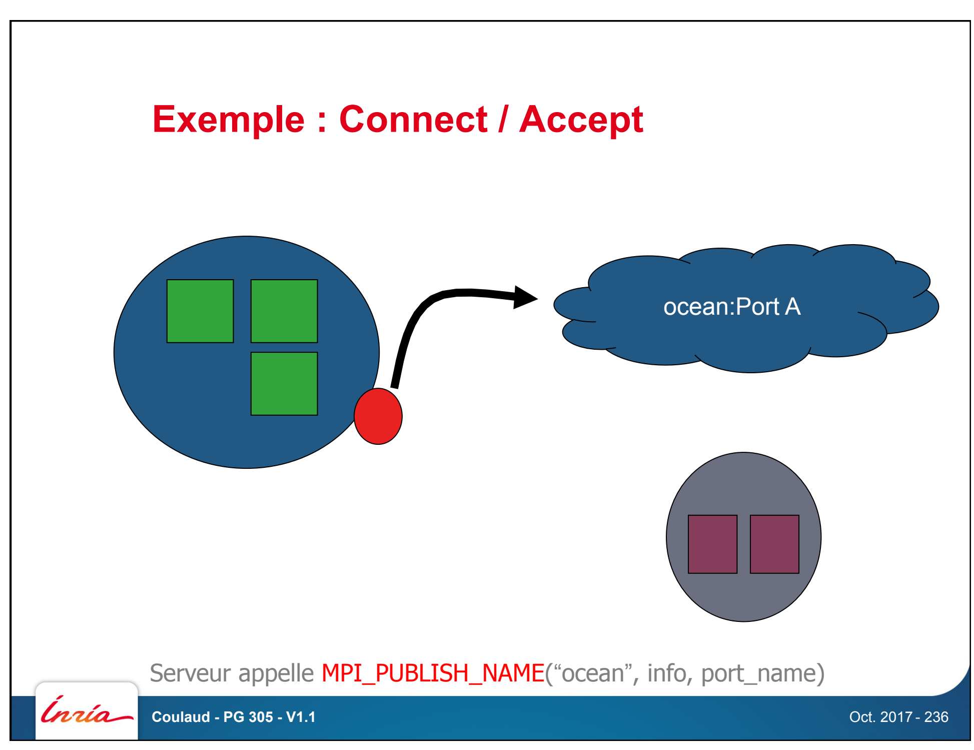
- MPI_COMM_ACCEPT(port_name, info, root, comm, newcomm)

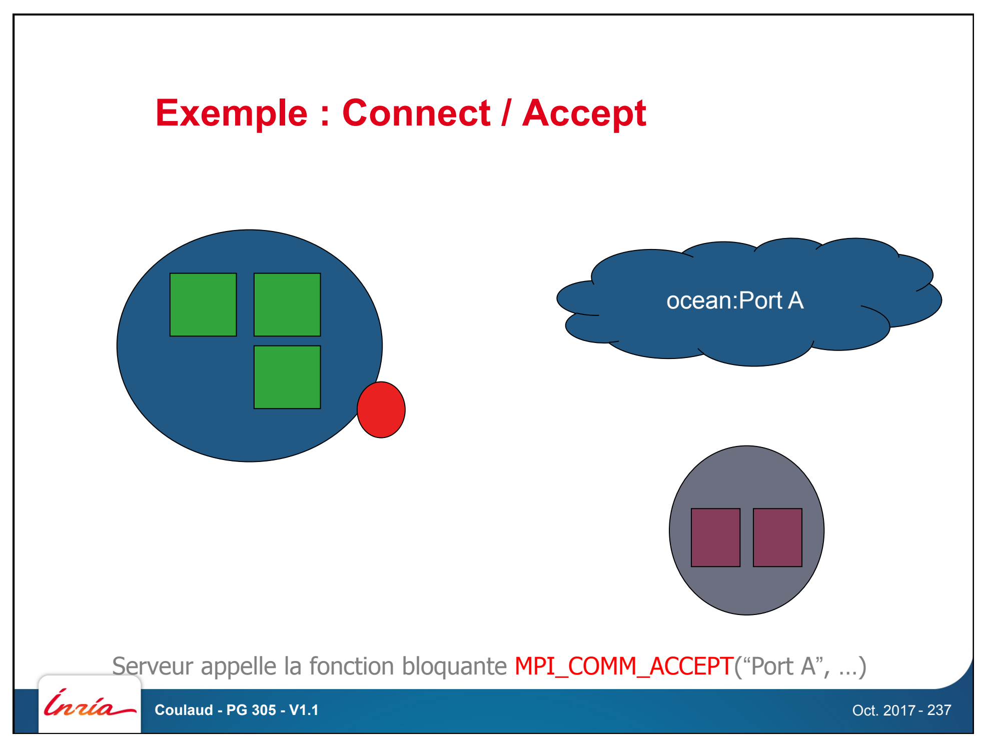
comm est un **intra**communicator; groupe local
newcomm est un **inter**communicator; les deux groupes

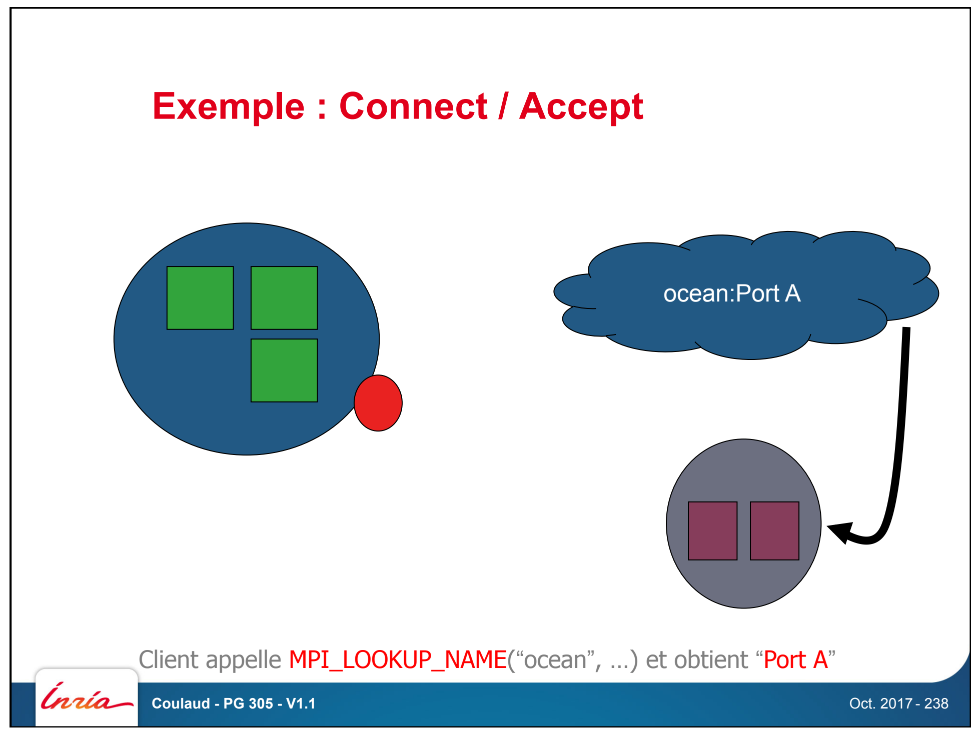
Coté client
Chercher le nom du port
MPI_LOOKUP_NAME(service_name, info, port_name)
Se connecter à un port
MPI_COMM_CONNECT(port_name, info, root, comm, newcomm)
comm est un intracommunicator; groupe local
newcomm est un intercommunicator; les deux groupes
Opération collective dans le communicateur comm
Coulaud - PG 305 - V1.1
Oct. 2017 - 233

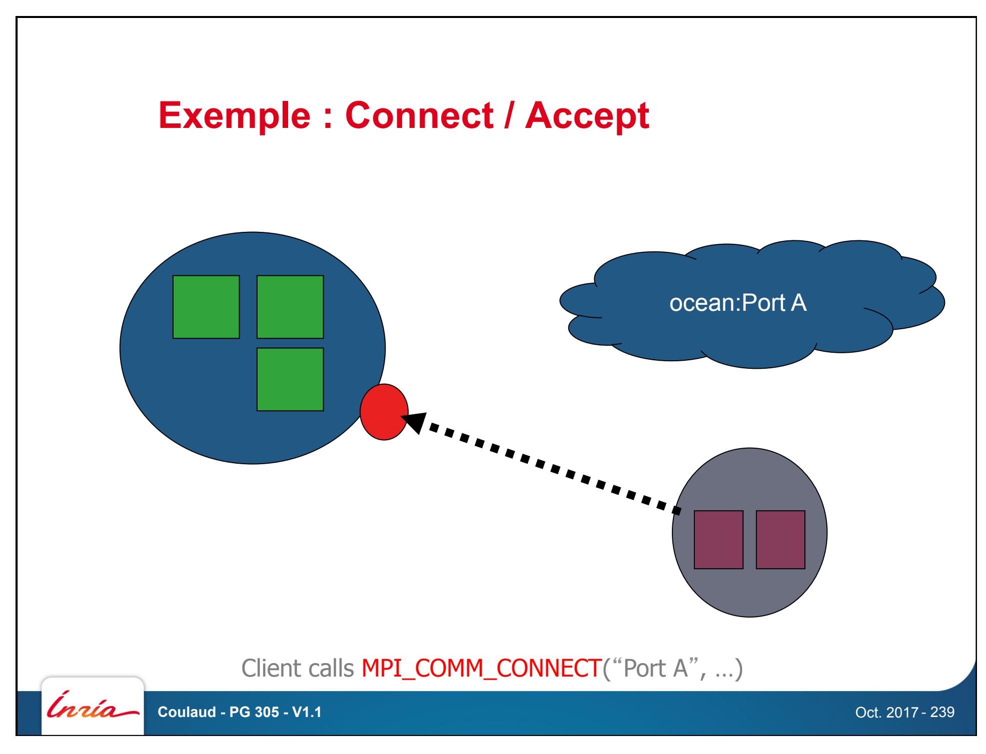

Exemple : Connect / Accept
Coulaud - PG 305 - V1.1
Oct. 2017 - 234

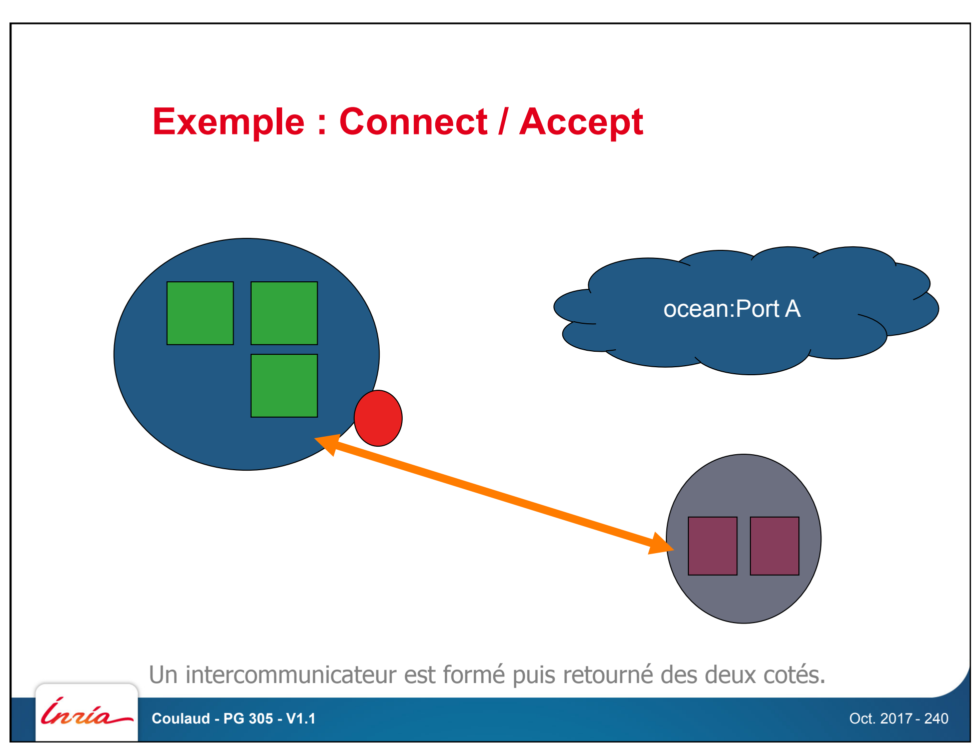

Exemple : Connect / Accept
Port A
Serveur appelle MPI_OPEN_PORT
Coulaud - PG 305 - V1.1
Oct. 2017 - 235

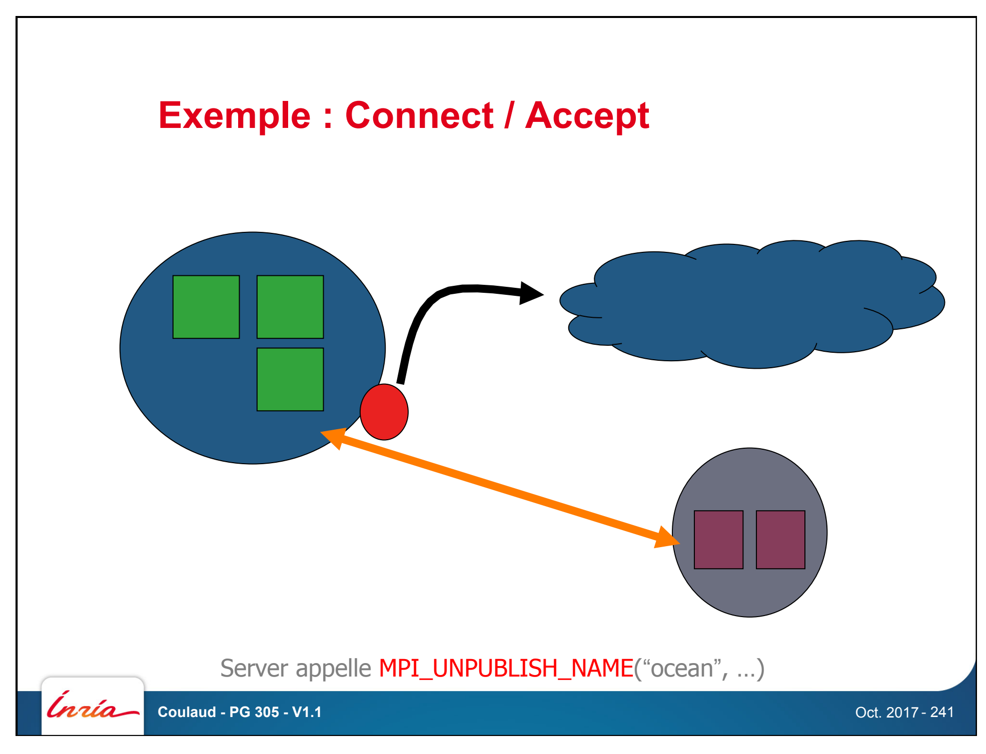

Exemple : Connect / Accept
ocean:Port A
Serveur appelle MPI_PUBLISH_NAME("ocean", info, port_name)
Coulaud - PG 305 - V1.1
Oct. 2017 - 236

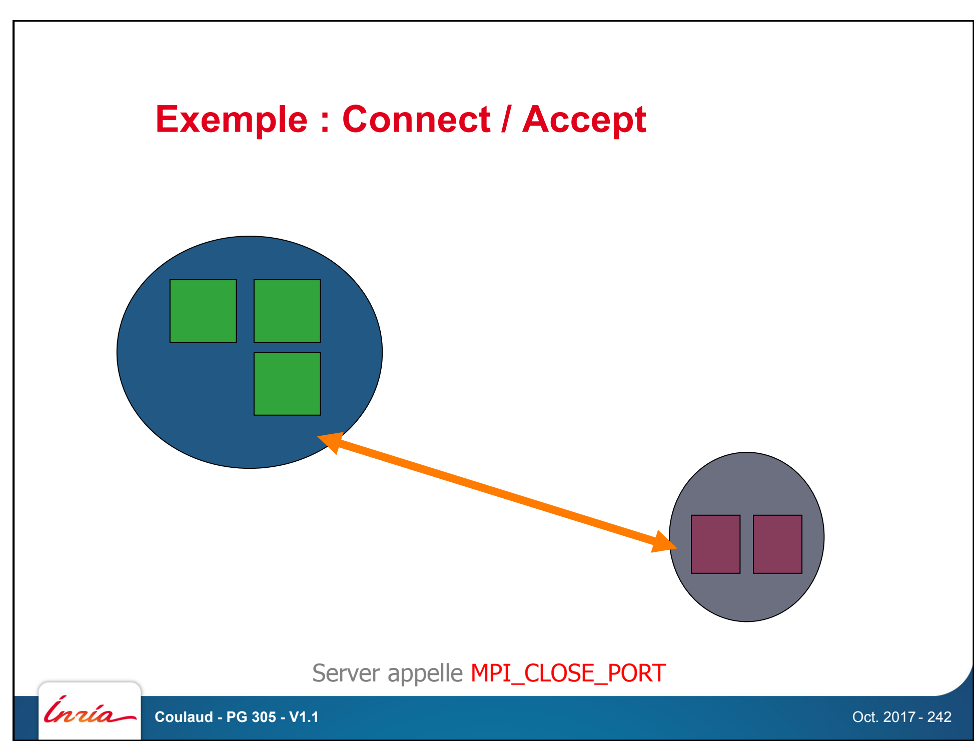

Exemple : Connect / Accept
ocean:Port A
Serveur appelle la fonction bloquante MPI_COMM_ACCEPT(“Port A”, ...)
Coulaud - PG 305 - V1.1
Oct. 2017 - 237

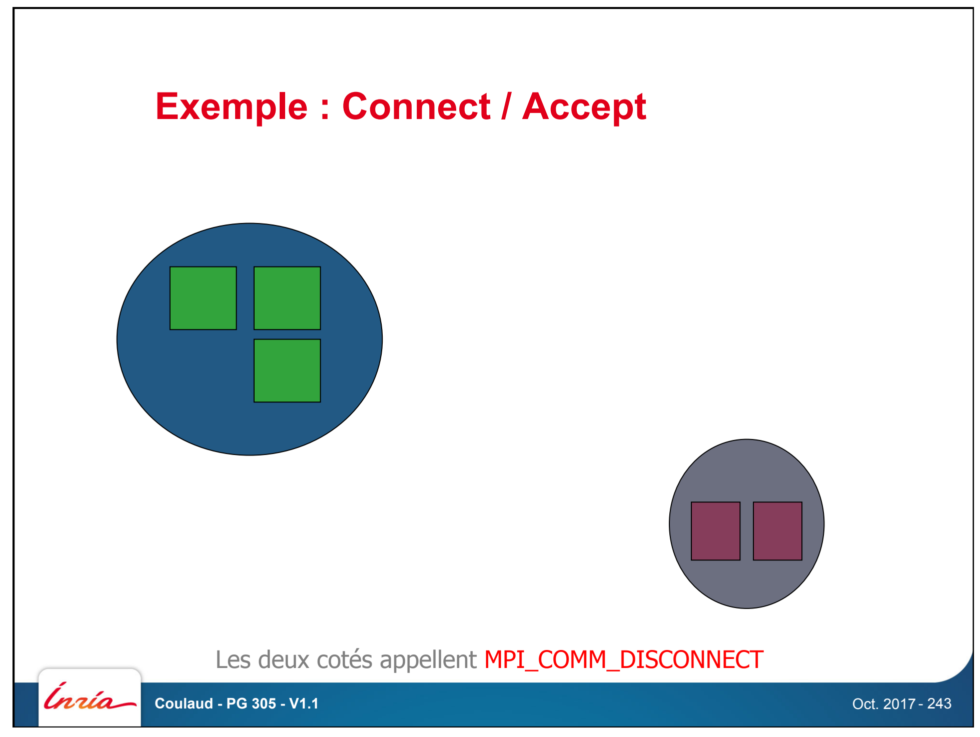

Exemple : Connect / Accept
ocean:Port A
Client appelle MPI_LOOKUP_NAME(“ocean”, ...) et obtient “Port A”
Coulaud - PG 305 - V1.1
Oct. 2017 - 238


Exemple : Connect / Accept
ocean:Port A
Client calls MPI_COMM_CONNECT("Port A", ...)
Coulaud - PG 305 - V1.1
Oct. 2017 - 239


Exemple : Connect / Accept
ocean:Port A
Un intercommunicateur est formé puis retourné des deux cotés.
Coulaud - PG 305 - V1.1
Oct. 2017 - 240


Exemple : Connect / Accept
Server appelle MPI_UNPUBLISH_NAME("ocean", ...)
Coulaud - PG 305 - V1.1
Oct. 2017 - 241


Exemple : Connect / Accept
Server appelle MPI_CLOSE_PORT
Coulaud - PG 305 - V1.1
Oct. 2017 - 242



Résumé

- Serveur ouvre un port
- Serveur publie un nom public
- Client cherche le nom public du port
- Client se connecte au port
- Serveur supprime le nom du service
- Serveur ferme le port
- Les deux cotés se déconnectent

➔ Similaire au socket TCP / DNS lookups

Déconnexion

Pour déconnecter deux processus dynamiques

MPI_Comm_disconnect(comm)

Attend que toutes les communications en cours soient terminées.

On déconnecte des groupes de processus. Ils ne sont plus « connectés ».

On ne peut pas déconnecter des processus avec le communicateur MPI_COMM_WORLD.

Exercise

Un processus master spawn deux groupes de processus b et c.

Le but est de construire un inter-communicateurs entre les groupes b et c.
Le processus b1 broadcast la valeur 200 aux processus du groupe c

Indication

- Tester vos communicateurs avec des communications
- Construire les intra-communicateurs bleus
- Construire un intra-communicateur avec tous les processus
- Construire l'inter-communicateur vert