Introduction à la programmation par passage de messages: **M**essage **P**assing **I**nterface

Julien ZORY

Centre de Recherche en Informatique

École des mines de Paris

Plan de l'exposé

❶ Le passage de messages

❷ MPI: Un besoin de standard

❸ Premiers pas en MPI

❹ Communications point à point

- ➢ bloquantes
- ➢ non bloquantes

❺ Types de données

❻ Communicateurs

❼ Communications collectives

❽ Topologies

Contexte

- DMPC
- MIMD (SPMD)
- Echange de données *coopératif* ou *unilatéral* entre les processeurs

Intérêt

- Une méthode de parallélisation **performante**
- Certains codes ne peuvent pas être parallélisés autrement
- Une base pour les compilateurs ”paralléliseurs”

MPI : Un besoin de Standard

Un besoin de standard: Les motivations

- Portabilité
- Simplicité d'utilisation
- Spécifier un contenu *minimal* pour les implémentations
- Les bibliothèques des *constructeurs* ne sont pas portables
- Les bibliothèques du *domaine publique* sont incomplètes, moins performantes

Les moments forts

- Démarrage : Workshop à Williamsburg en Avril 92
- Organisation du Forum : Supercomputing'92
- Rencontres tous les 6 mois
- Béta version présentée à Supercomputing'93
- Période d'évaluation publique
- Première version : Mai 94
- MPI-2 : en cours de réalisation

☞ Regroupe des spécialistes du domaine venant de plusieurs horizons

- Des industriels :
  - ➠ Cray, IBM, Intel, TMC, ...
- Des universités :
  - ➠ U of Tenessy, Rice University, U. of Maryland ...
- Des Laboratoires :
  - ➠ Oak Ridge National Laboratory, GMD, NASA, SC ...

- Routines de communications *point à point*
- Routines de communications *collectives*
- Support pour les groupes de *processus* statiques
- Support pour les *contextes* de communications
- support pour les *topologies* d'applications
- Interface pour le profiling

- Opérations type mémoire partagée
- support pour la manipulation des tâches
- support pour les méssages actifs
- support pour les processus légers (threads)
- support pour le debug
- fonctions d'Entrées/Sorties (dans MPI-2)

Premiers pas en MPI

MPI

Premiers pas en MPI: Introduction

- ✥ MPI c'est plus de **125 fonctions**
- ✥ La plupart des programmes peuvent s'écrire avec **seulement 6 fonctions**
- ✥ Notations :

  ✰ En **C** :

  #include <mpi.h>;

  error = MPI_xxxxx(parameter,...);

  ✰ En **Fortran** :

  include 'mpif.h'

  call MPI_xxxxx(parameter,...,ierror)

Initialiser MPI

☛ int MPI_Init(int *argc, char ***argv);

- ✥ Doit être la *première routine MPI appelée* (et une seule fois)
- ✥ *argc* et *argv* sont les arguments de commande (main) qui peuvent être utilisés pour l'implémentation (seulement en C)

☛ int MPI_Initialized(flag);

- **Sortir proprement**
  ☞ int MPI_Finalize(void);
  Doit être appelée *par tous les processus*

- **La méthode brutale**
  ☞ int MPI_Abort(comm, errorcode);
  Termine (sauvagement) tous les processus du communicateur *comm* (varie selon l'implémentation)

- **Combien sommes-nous ?**
  ☞ int MPI_Comm_size (comm, size);
  Indique le nombre de processus au sein du communicateur *comm*

- **Qui suis-je ?**
  ☞ int MPI_Comm_rank (comm, rank);
  Indique à un processus sa position au sein du communicateur *comm*

```
#include "mpi.h"
#include <stdio.h>
int main (argc, argv)
int argc; int **argv;
{int rank, size;
   MPI_Init( &argc,  &argv);
   MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
   MPI_Comm_size ( MPI_COMM_WORLD, &size );
  printf("Hello world ! I'm %d of %d \n"
         , rank, size);
   MPI_Finalize();
  return 0; }
```

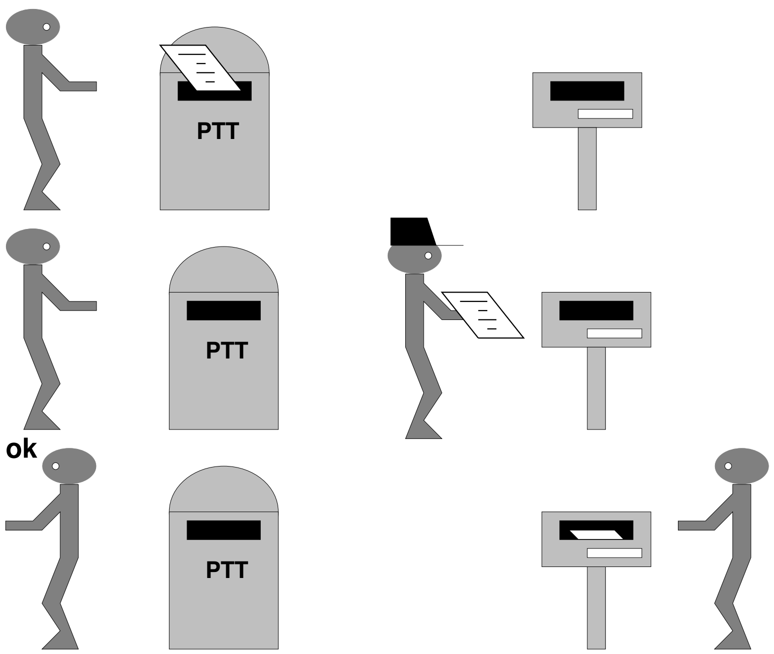

Communications point à point

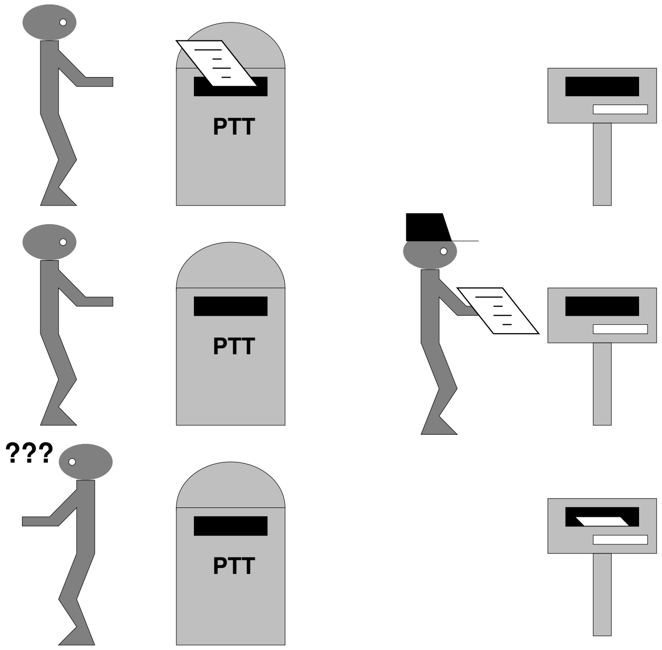
MPI

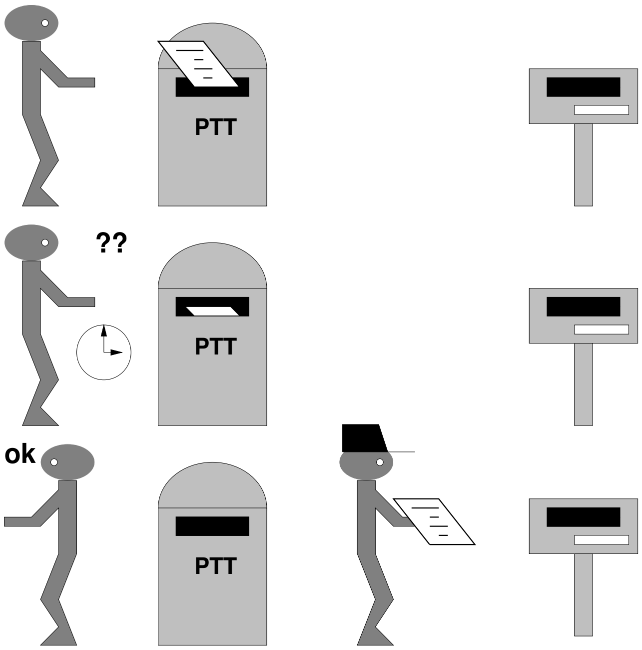
Les Modes de communications point à point 16

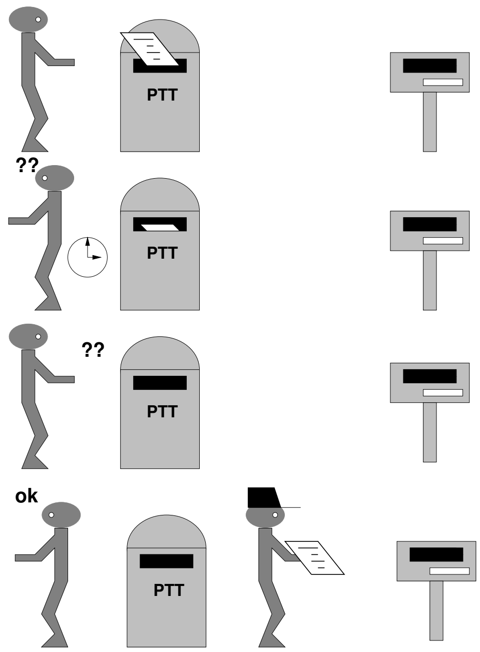
- **Objectif** :
  Permettre la communication entre deux processus
- **Contraintes** :
  Efficacité, portabilité, déterminisme, robustesse ...
- **Plusieurs réponses** :
  - ✗ synchrones ou asynchrones
  - ✗ bloquantes ou non-bloquantes
  - ✗ bufferisées ou non

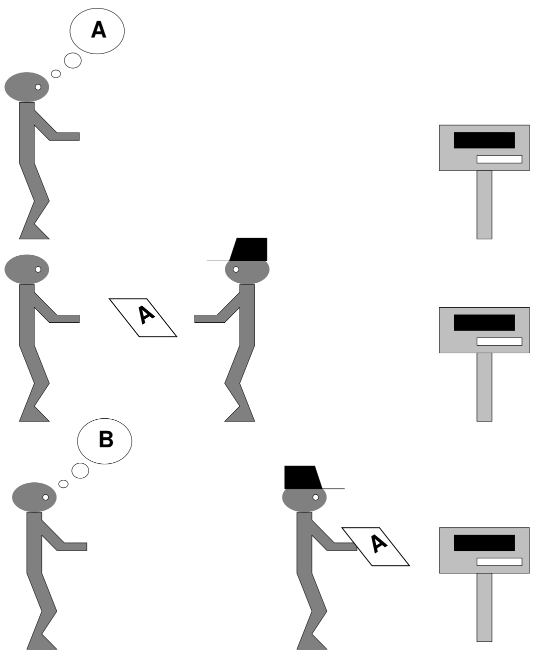
Communications Synchrones 17



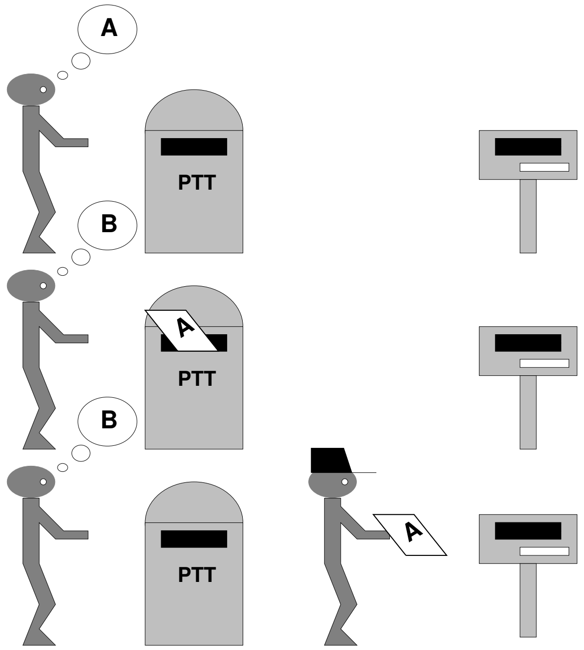

PTT
PTT
???
PTT


PTT
??
PTT
ok
PTT


PTT
??
PTT
??
PTT
ok
PTT


A
A
B
A

Communications Bufferisées



Modes de communications pour MPI

- Envoi bloquant

  ✗mode **Standard** ☚ **MPI_Send**

  Démarrage : sans contrainte

  Terminaison : MPI choisit !

  Si bufferisation → message bufferisé

  sinon → message arrivé

  ✗mode **Bufferisé** ☚ **MPI_Bsend**

  Démarrage : sans contrainte

  Terminaison : → message bufferisé

  (peut terminer avant emission du receive - si plus de place ⇒ erreur)

  Gestion du buffer laissée au programmeur (!! overflow !!)

  (MPI_Buffer_attach – MPI_Buffer_detach)

✗mode **Synchrone** ☞ **MPI_Ssend**

Démarrage : sans contrainte

Terminaison : → opération de réception commencée

✗mode **Prêt** (ready) ☞ **MPI_Rsend**

Démarrage : receive correspondant déjà posté

Terminaison : idem standard

(notion de rendez-vous)

✤ Réception Bloquante

☞ **MPI_Recv**

✤ Envoi non-bloquant (Immediatly)

✗mode **Standard** ☞ **MPI_Isend**

✗mode **Bufferisé** ☞ **MPI_Ibsend**

✗mode **Synchrone** ☞ **MPI_Issend**

✗mode **Prêt** ☞ **MPI_Irsend**

✤ Réception non-bloquante

☞ **MPI_IRecv**

- De qui vient le message ? ➔ **source**
- A qui est-il destiné ? ➔ **destination**
- Quel est le message à envoyer ? ➔ **adresse du buffer**
- Comment identifier le message ? ➔**étiquette**
- Dans quel monde sommes nous ? ➔ **communicateur**
- Que contient le message ? ➔ **quantité, type**

- int **MPI_Send**(buf, count, datatype, dest, tag, comm)
- int **MPI_Recv**(buf, count, datatype, source, tag, comm, status)

(en reception : count est une borne sup)

```
MPI_Comm_rank ( MPI_COMM_WORLD, &myrank );
 if (myrank == 0) {
   strcpy (msg,"Hello there");
    MPI_Send ( msg, strlen(msg)+1, MPI_CHAR, 1, tag,
                         MPI_COMM_WORLD);
 } else if (myrank ==1) {
    MPI_Recv (msg, 20, MPI_CHAR, 0, tag,
                         MPI_COMM_WORLD, &status);
 }
```

Type MPI	Type C correspondant
MPI_CHAR	signed char
MPI_INT	signed int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_BYTE	
MPI_PACKED	

❶ Message lu dans le buffer d'emission puis assemblé
❷ Message transféré de l'émetteur vers le récepteur
❸ Message désassemblé et copié dans le buffer de réception

Correspondances

✓ types des variables émetteur et récepteur
✓ types C/Fortran et les types MPI

Conversions

✓ nécéssaires dans des environnements non homogènes
(ex: réel codé sur 4 ou 8 octets)

Les jokers

▲ Uniquement en RECEPTION ▼

- MPI_ANY_SOURCE
- MPI_ANY_TAG
- Return Status (en réception)
  - ✗ status.MPI_SOURCE
  - ✗ status.MPI_TAG
  - ✗ status.MPI_ERROR
  - ✗ MPI_GET_COUNT(status $<$ IN $>$,datatype $<$ IN $>$,count $<$ OUT $>$)

    (byte count → entry count : coûte cher)

Et aussi ...

- int MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)

  (échange de données, shift ...)

- int MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag, comm, status) (sendbuf = recvbuf)

  (idem mais on écrase la donnée)

- MPI_PROC_NULL

  (source ou destination inéxistante)

☛ Un process peut s'envoyer un message à lui-même (Bufferisation)

☛ Que faire si la taille du message à recevoir est inconnue ?
→ envoyer 2 messages (infos + données)
→ utiliser pack et unpack (self-typed message)


```
MPI_Comm_rank ( MPI_COMM_WORLD, &myrank );
 if (myrank == 0) {
   MPI_Send (sendbuf, count, MPI_REAL, 1, tag,
                   MPI_COMM_WORLD);
   MPI_Recv (recvbuf, count, MPI_REAL, 1, tag,
                   MPI_COMM_WORLD, &status);
 } else if (myrank ==1) {
   MPI_Send (sendbuf, count, MPI_REAL, 0, tag,
                   MPI_COMM_WORLD);
   MPI_Recv (recvbuf, count, MPI_REAL, 0, tag,
                   MPI_COMM_WORLD, &status);
 }
```

✘ Dépend de l'espace de bufferisation disponible

✒ int MPI_Isend(buf,count,datatype,dest,tag,comm,request)
L'envoi du message est initié - Retour immédiat

Allocation de l'objet *request* pour "manipuler" la communication en cours

✒ int MPI_Irecv(buf,count,datatype,source,tag,comm,request)
La réception du message est initiée (pas de status)

- NE PAS ACCEDER A LA DONNÉE avant terminaison de l'envoi
- Recouvrement des communications par les calculs
- Compatibilité avec les communications bloquantes

✒ int MPI_Test (request, flag, status)
Teste si l'envoi/réception est terminé
Retour "immédiat" avec flag = vrai ou faux

✒ int MPI_Wait (request, status)
Bloque jusqu'à terminaison de l'envoi/réception

✒ Remarque :
L'ordre d'éxécution des communications est respecté
(initialisation et non terminaison)

```
MPI_Comm_rank ( MPI_COMM_WORLD, &myrank );
 if (myrank == 0) {
   MPI_Isend (sendbuf, count, MPI_REAL, 1, tag,
                       MPI_COMM_WORLD,&req);
   MPI_Recv (recvbuf, count, MPI_REAL, 1, tag,
                       MPI_COMM_WORLD, &status);
   MPI_Wait (&req,&status);
 } else if (myrank ==1) {
   MPI_Isend (sendbuf, count, MPI_REAL, 0, tag,
                       MPI_COMM_WORLD,&req);
   MPI_Recv (recvbuf, count, MPI_REAL, 0, tag,
                       MPI_COMM_WORLD, &status);
   MPI_Wait (&req,&status);
 }
```

✘ Ne dépend plus de l'espace de bufferisation disponible

☛ int MPI_Request_free (MPI_Request *request)
Libération si la terminaison du message est connue avec certitude
(pour les envois uniquement)

➠ Appel à MPI_WAIT ou MPI_TEST superflu

(attention : ce n'est pas une annulation)

☛ int MPI_Probe(source, tag, comm, status)
☛ int MPI_Iprobe(source, tag, comm, flag, status)
Examine un "maching message" sans le recevoir
Récupère les informations dans status

Annuler la communication 38

☛ int MPI_Cancel(MPI_Request *request)
Demande l'annulation de la communication (marquage)
Retour immédiat

☛ int MPI_Test_cancelled(status < IN >, flag)
Teste la réussite de l'annulation

Terminaisons multiples : Any 39

➠ Versions des routines Wait et Test travaillant sur plusieurs communications

☛ int MPI_Waitany(count,array_of_requests,index,status)
Bloque jusqu'à ce que UN Envoi/Réception soit terminé

☛ int MPI_Testany(count,array_of_requests,index,flag,status)
Teste si UN Envoi/Réception est terminé

(si plusieurs : choix arbitraire)

☛ int MPI_Waitall(count,array_of_requests,array_of_statuses)
Bloque jusqu'à ce que TOUS les Envoi/Réception soient terminés

☛ int MPI_Testall(count,array_of_requests,flag, array_of_statuses)
Teste si TOUS les Envoi/Réception sont terminés

☛ int MPI_Waitsome(incount,array_of_requests,outcount,
array_of_indices,array_of_statuses)
Bloque jusqu'à ce que AU MOINS UN Envoi/Réception soit terminé

☛ int MPI_Testsome(incount,array_of_requests,outcount,
array_of_indices,array_of_statuses)
Teste si AU MOINS UN Envoi/Réception est terminé

(Au moins un – tous ceux disponibles en même temps)

Types de données

Types de données en MPI

- Communication = Adresse du buffer + Type de la donnée
- Données *non-homogènes* (structures)
- Données *non contigues en mémoire* (sections de tableaux)
- 2 méchanismes proposés

  ❶ Types dérivés - généraux définis par l'utilisateur

  (construction hiérarchique à partir des types simples)

  ❷ primitives Pack et Unpack

  ➠ Diminution des surcoûts d'initialisation

➠ Description de la disposition en mémoire d'un ensemble de types simples

- ❖ Une séquence de type simples
- ❖ Une séquence de déplacements en octets

type map = $\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\}$
type signature = $\{type_0, ..., type_{n-1}\}$

- ❖ définit n entrées
- ❖ l'entrée i est à l'adresse $buf + disp_i$
- ❖ l'entrée i est de type $type_i$

Exemple : $\{(int, 0)\}$ définirait le type MPI_INT

Extent : Distance entre le premier et le dernier octet réservés pour une donnée de ce type

$lb(Typemap) = \min_j disp_j$
$ub(Typemap) = \max_j (disp_j + sizeof(type_j)) + \epsilon$
$extent(Typemap) = ub(Typemap) - lb(Typemap)$

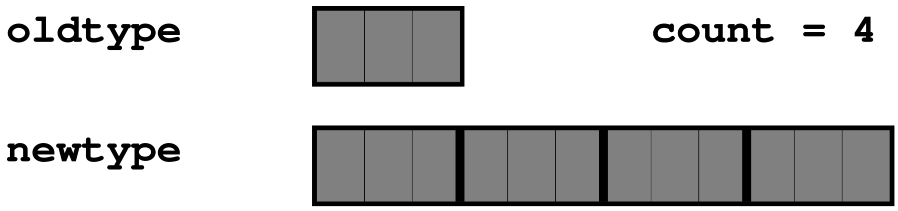
Exemple : $\{(double, 0), (char, 8)\}$
Alignement des doubles sur des adresses multiples de 8
➠ $extent = 16$

☞ int MPI_Type_extent (datatype, extent) (Ex = 16)
☞ int MPI_Type_size (datatype, size) (Ex = 9)

➠ Concaténation de *count* copies d'un type donné

(chaque copie est de taille extent)

☞ int MPI_Type_contiguous(count, old_type, newtype)



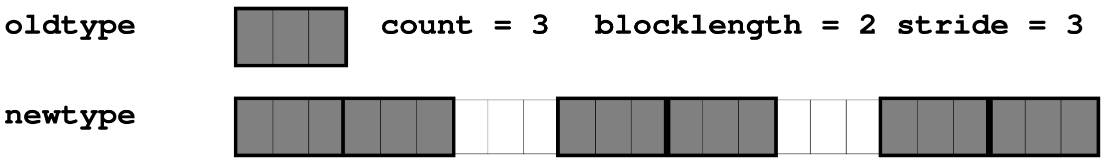
Exemple : old_type = $\{(double, 0), (char, 8)\}$ et count = 3

newtype = $\{(double, 0), (char, 8), (double, 16), (char, 24), (double, 32), (char, 40)\}$

➠ Concaténation avec un pas *stride*
de *count* blocs de tailles identiques
contenant *blocklen* copies d'un type donné

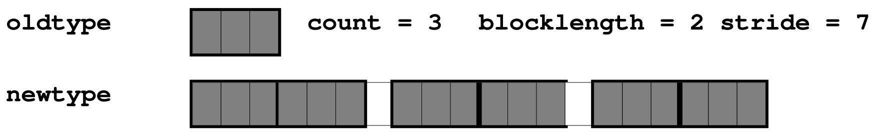
Le pas est donné en *nombre d'éléments*

☞ int MPI_Type_vector(count,blocklen,stride,old_type,newtype)



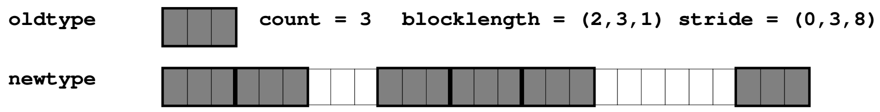
Le pas est donné en nombre d'octets (H = heterogeneous)

✎ int MPI_Type_hvector(count, blocklen, stride, old_type, newtype)

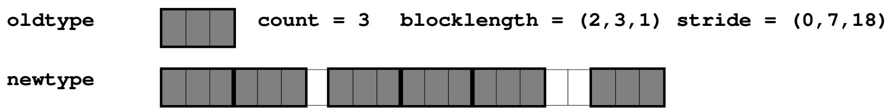


➠ le nombre de copies par blocs et le pas changent pour chaque bloc

✎ int MPI_Type_indexed(count, array_of_blocklens, array_of_strides, old_type, newtype)

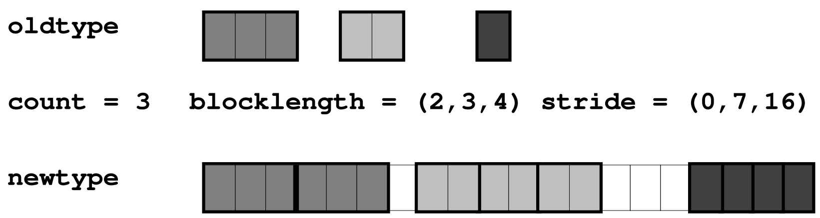


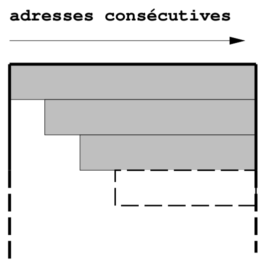
✎ int MPI_Type_hindexed(count, array_of_blocklens, array_of_strides, old_type, newtype)



➠ les types utilisés sont différents pour chaque bloc

☞ int MPI_Type_struct(count, array_of_blocklens, array_of_indices, array_of_types, newtype)






```
double a[100][100];
int disp[100],blocklen[100],i;
MPI_Datatype upper;
```

```
...
/* Calculer le début et la taille pour chaque ligne */
for (i=0; i  <  100; i++) {
     disp[i] = 100 *i + i;
     blocklen[i] = 100 - i;
}
/* Créer un type pour la partie triangulaire supérieure */
MPI_Type_indexed (100, blocklen, disp, MPI_DOUBLE, &upper);
MPI_Type_commit (&upper);
/* Effectuer l'envoi ... */
MPI_Send (a,1,upper, dest,tag,MPI_COMM_WORLD);
```

- Installer : int MPI_Type_commit (datatype)
- Libérer : int MPI_Type_free (datatype)
- int MPI_Get_count(status, datatype, count)
  Nombre d'éléments de type *datatype* reçus
- int MPI_Get_elements (status, datatype, elements)
  Nombre d'éléments de types primaires

MPI_Send(buf, **count**, datatype ...)

EST EQUIVALENT A

MPI_Type_contiguous(count, datatype, newtype)
MPI_Type_commit(newtype)
MPI_Send (buf, 1, newtype, ...)

➠ Emballage de données non contigues
dans un buffer contigu

☛ int MPI_Pack (inbuf, incount, type, outbuf, outcount, position, comm)

☛ int MPI_Unpack (inbuf, insize, position, outbuf, outcount, type, comm)

position indique la position de départ dans le buffer (en octets)

position est incrémenté de la taille du message concaténé

comm est utile pour éviter les conversions (XDR) dans un environnement homogène

☛ int MPI_Pack_size (incount, datatype, comm, size)

borne sup pour la taille du buffer *outbuf* (dépend du comm)


```
int i;
char c[100];
char buffer[110];
int position = 0;

/* pack */

MPI_Pack(&i,1,MPI_INT,buffer,110,&position,MPI_COMM_WORLD);
MPI_Pack(c,100,MPI_CHAR,buffer,110,&position,MPI_COMM_WORLD);

/* send */

MPI_Send(buffer,position,MPI_PACKED,1,0,MPI_COMM_WORLD);
```

```
int i;
char c[100];
char buffer[110];
int position = 0;

/* receive */
MPI_Recv(buffer,110,MPI_PACKED,1,0,MPI_COMM_WORLD,&status);

/* Unpack */
MPI_Unpack(buffer,110,&position,&i,1,MPI_INT,MPI_COMM_WORLD);
MPI_Unpack(buffer,110,&position,c,100,MPI_CHAR,MPI_COMM_WORLD);
```


- Types dérivés
  - ☞ Décrire des données non-contingues
  - ✗ Déclarations (structures) ou données régulières (tableaux)
  - ✗ Mémoire : description de la structure
  - ✗ Réutilisable facilement
- Pack-Unpack
  - ☞ Concaténation dans un buffer contigu
  - ✗ Données irrégulières, taille inconnue
  - ✗ Mémoire : copie de la donnée + plusieurs appels
  - ✗ Possibilité d'utiliser une partie du message pour manipuler la suite

✗ Compatibilité avec d'autres bibliothèques (PVM)

Communicateurs

MPI

➠ Créer des **univers de communication**

- Diviser les processus ➢ travaux indépendants
- Opérations collectives ”privées”
- Isoler les modules les uns des autres (bibliothèques)
- Attacher des calculs d’optimisations à un ensemble de processus
- Le contrôle total de l’utilisateur est souvent moins sûr

- Un communicateur est un

  Objet **opaque local** à un process

  Avec des **attributs**

  Spécifie un **domaine de communication**
- Le rang des processus est interprété relativement au groupe associé au communicateur

- **Intracommunicateur**
  Communication à l'intérieur d'un groupe de processus
  Attributs : groupe de processus + topologie + attribues cachés
  Point à Point ou Collectives

- **Intercommunicateur**
  Communication entre deux groupes de processus disjoints
  Attributs : 2 groupes
  Point à Point

➠ Communication à l'intérieur d'un groupe de processus

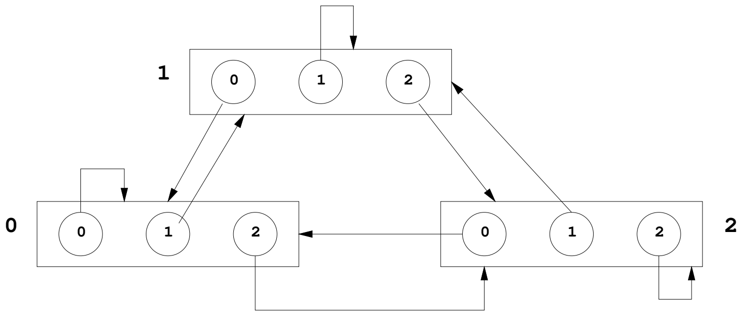
- Communications point à point ou collectives

- Toute communication est associée à un *domaine de communication*

  Représenté par un ensemble de communicateurs compatibles (un par processus)

  Chaque communicateur est une représentation locale du domaine de communication global

- Un envoi "s'accorde" avec une réception si les deux communicateurs représentent le même domaine de communication.



Manipuler un groupe

- int MPI_Group_size (group, size)
  Nombre de processus dans le groupe
- int MPI_Group_rank (group, rank)
  Rang du processus appelant au sein du groupe
- int MPI_Group_translate_ranks (group1, n, array_of_ranks1, group2, array_of_ranks2)

Donne le rang *rank2* dans *group2* des processus
de rang *rank1* dans *group1*

Exemple : group1 : {a,b,c,d,e,f} group2 : {d,e,a,c}

MPI_Group_translate_ranks (group1,4,(0,5,0,2),group2,rank2)

➪ rank2 = (2,MPI_UNDEFINED,2,3)

❖ int MPI_Group_compare (group1, group2, result)

❖ → MPI_IDENT

les membres sont les mêmes

l'ordre est identique

❖ → MPI_SIMILAR

les membres sont les mêmes

l'ordre est différent

❖ → MPI_UNEQUAL

▲ On ne peut construire un nouveau groupe que depuis un autre groupe pré-éxistant ▼

❖ Le groupe de base est MPI_COMM_WORLD

❖ int MPI_Comm_group (comm $<$ IN $>$, group $<$ OUT $>$)
Retourne le groupe associé à *comm*

Message Passing Interface —Intra-communicateurs—
Centre de Recherche en Informatique — ÉCOLE DES MINES DE PARIS
MPI

- int MPI_Group_union (group1, group2, newgroup)
- int MPI_Group_intersection (group1, group2, newgroup)
- int MPI_Group_difference (group1, group2, newgroup)

✧ Ordre : d'abord celui du group1 puis celui du group2
✧ Pas de doublons

Exemple : group1 = {a,b,c,d} group2 = {d,a,e}
union : {a,b,c,d,e} intersection : {a,d} différence : {b,c}

- int MPI_Group_incl (group, n, ranks_array, newgroup)
  ✧ Sous-ensemble réordonné
  Exemple : group = {a,b,c,d,e,f} ranks = (3,1,2)
  newgroup = {d,b,c}
- int MPI_Group_excl (group, n, ranks_array, newgroup)
  ✧ Suppression des processus indiqués
  Exemple : group = {a,b,c,d,e,f} ranks = (3,1,2)
  newgroup = {a,e,f}

- int MPI_Group_free (group)
  group est forcé à MPI_GROUP_NULL

- int MPI_Comm_size (comm, size)
  retourne la taille du groupe associé au communicateur
  (équivalent à MPI_Comm_group + MPI_Group_size + MPI_Group_free)
- int MPI_Comm_rank (comm, rank)
  (équivalent à MPI_Comm_group + MPI_Group_rank + MPI_Group_free)
- int MPI_Comm_compare (comm1, comm2, result)
  - ➔ MPI_IDENT (pointeurs sur le même objet)
  - ➔ MPI_CONGRUENT (membres + ordre)
  - ➔ MPI_SIMILAR (membres)
  - ➔ MPI_UNEQUAL

➠ Ce sont des fonctions collectives

(exécutés par tous les processus associés)

- ✥ int MPI_Comm_dup (comm, comm_out)
  Nouveau communicateur avec les mêmes attributs
  ➠ Deux domaines de communications disjoints

- ✥ int MPI_Comm_create (comm, group, comm_out)
  Nouveau communicateur basé sur le groupe *group*
  Tous les processus de *comm* exécutent cet appel
  Les processus hors *group* recoivent MPI_COMM_NULL
  ERREUR si *group* n'est pas un sous-ensemble du groupe associé à *comm*

- ✥ int MPI_Comm_split (comm, color, key, comm_out)
  Création de *k sous-ensembles* disjoints
  color définit le sous-groupe
  key définit le rang

Exemple :

rank	0	1	2	3	4	5	6	7	8	9
process	a	b	c	d	e	f	g	h	i	j
color	0	$\perp$	3	0	3	0	0	5	3	$\perp$
key	3	1	2	5	1	1	1	2	1	0

➪ 3 nouveaux groupes : {f,g,a,d} {e,i,c} {h}

➪ b et j ne participent à aucun nouveau groupe.

- int MPI_Comm_free (commp)
  L'objet *comm* est marqué pour être désalloué
  Mis à MPI_COMM_NULL si plus aucun communicateur ne pointe sur *comm*

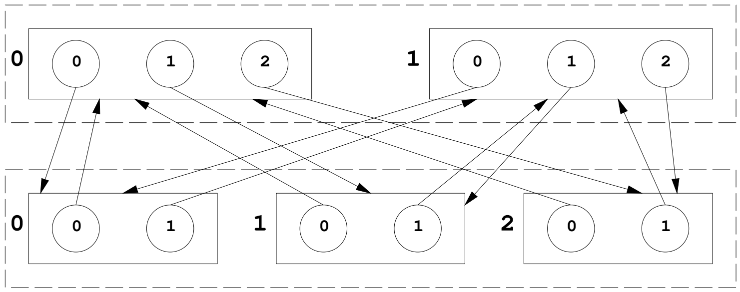
Ce sont des informations utiles attachées aux communicateurs

- Passer des informations à l'intérieur des appels
- Retrouver rapidement l'information
- Garantir que l'information est à jour
- Chaque attribut est associé à une clé
- Chaque attribut doit avoir un type prédéfini

- int MPI_Keyval_create (copy_fn, delete_fn, keyval, extra_state)
  Création d'un attribut caché+ attribution clé
- int MPI_Keyval_free (keyval)
  Destruction d'un attribut caché
- int MPI_Attr_put (comm, keyval, attr_value)
  Insertion/Modification d'une valeur dans un attribut caché
- int MPI_Attr_get (comm, keyval, attr_value, flag)
  Accès à la valeur d'un attribut caché
- int MPI_Attr_delete (comm, keyval)
  Suppression de la valeur d'un attribut caché

➠ Communication point à point entre
deux groupes de processus disjoints

- Notion de groupe *local* et groupe *distant*
- Communication point à point entre *un* processus du groupe local et *un* processus du groupe distant
- Le rang indiqué dans la communication est relatif au groupe distant
- Les groupes doivent être *disjoints*
- Un intercommunicateur ne peut pas être utilisé pour des communications collectives



- int MPI_Comm_test_inter (comm, flag)
  Indique si *comm* est un Intra ou un Inter communicateur
- int MPI_Comm_size (comm, size)
  Indique la taille du groupe *local*
- int MPI_Comm_rank (comm, rank)
  Indique le rang au sein du groupe local

Manipulation d'un Intercommunicateur 80

- int MPI_Comm_group (comm, group)
  Retourne un pointeur sur le groupe local
- int MPI_Comm_remote_size (comm, size)
  Indique la taille du groupe distant
- int MPI_Comm_remote_group (comm, group)
  Retourne un pointeur sur le groupe distant

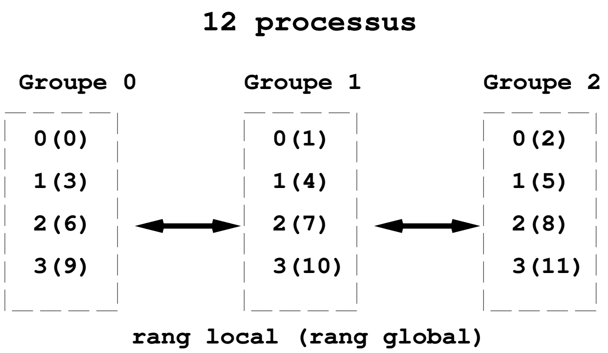
Construction d'un Intercommunicateur 81

- int MPI_Comm_dup (comm, comm_out)
  Nouveau communicateur avec les mêmes attributs
- int MPI_Intercomm_create (local_comm, local_leader, bridge_comm, remote_leader, tag, comm_out)
  - Création d'un intercommunicateur à partir de deux intra-communicateurs
  - Nécéssite une communication entre deux leaders pour chaque groupe (avec MPI_COMM_WORLD par ex)
  - Echange des données entre les deux leaders
  - Broadcast dans le groupe local

Construction d'un Intercommunicateur

- int MPI_Intercomm_merge (comm, high, comm_out)
  - ✧ Création d'un Intra-communicateur réunissant les 2 groupes de l'Intercommunicateur
  - ✧ high (T/F) permet d'imposer un ordre pour la fusion
- int MPI_Comm_free (commp)

Intercommunicateur : Exemple



```
main(int argc, char **argv)
{
  MPI_Comm   mycomm;       /* Intra-communicator of local sub-group*/
  MPI_Comm   myFirstComm;   /* Inter-communicator */
  MPI_Comm   mySecondComm; /* second inter-communicator (group 1 only)*/
  int membershipKey, rank;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  /* Generate membershipKey in the range [0,1,2] */
  membershipKey = rank % 3;

  /* Build intra-communicator for local sub-group */
  MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);
```



```
  /* Build Inter-communicators */
  if (membershipKey == 0 ) {
    /* Group 0 communicates with group 1 */
    MPI_Intercomm_create( mycomm, 0, MPI_COMM_WORLD,
                               1, 01, &myFirstComm);
  }

  else if (membershipKey == 1) {
    /* Group 1 communicates with group 0 and 2 */
    MPI_Intercomm_create( mycomm, 0, MPI_COMM_WORLD,
                               0, 01, &myFirstComm);
    MPI_Intercomm_create( mycomm, 0, MPI_COMM_WORLD,
                               2, 12, &mySecondComm);
  }

  else if (membershipKey == 2) {
    /* Group 2 communicates with group 1 */
    MPI_Intercomm_create( mycomm, 0, MPI_COMM_WORLD,
                               1, 12, &myFirstComm);
  }
```

```
/* Do work .... */

/* free communicators appropriately */

 MPI_Comm_free(&myComm);
 MPI_Comm_free(&myFirstComm);

if (membershipKey == 1)
    MPI_Comm_free(&mySecondComm);
 MPI_Finalize();
}
```


Communications collectives

MPI

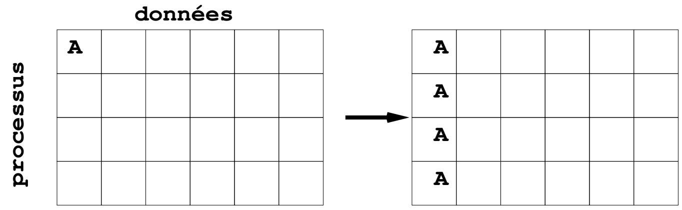
➠ Communications coordonnées à l'intérieur d'un groupe

- Blocage jusqu'à terminaison locale
- Pas d'étiquettes attachées aux messages (tag)
- La taille des données reçues doit être exactement celle attendue
- pas de conflits entre communications collectives et communications point à point
- Trois classes de communications collectives
  - Mouvements de données
  - Calculs globaux (réductions et scans)
  - Barrières de synchronisation

- int MPI_Barrier (comm)
  Bloque le processus appelant jusqu'a ce que tous les processus du groupe associé au communicateur l'aient appelé

Broadcast

- int MPI_Bcast (buffer, count, datatype, root, comm)
  Diffuse le message à tous les processus du groupe



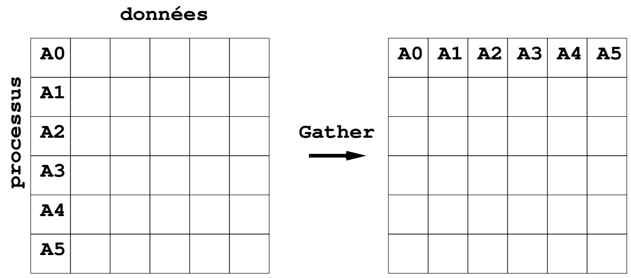
Gather

- int MPI_Gather (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
  - Chaque processus envoie le contenu du buffer au processus ”racine”.
  - Le processus racine recoit les messages et les range selon le rang
  - *recvcount* spécifie le nombre d’éléments reçus de CHAQUE processus
  - équivalent à

  MPI_Send(sendbuf, sendcount, sendtype, root ...)(par tous)

  MPI_Recv(recvbuf+i.recvcount.extent(recvtype), recvcount, recvtype,i ...) (par la racine)

Gather



Gatherv

- int MPI_Gatherv (sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm)

  - Autorise l'envoi de différentes quantités de données

  - Placement des données dans le buffer de reception spécifié par le tableau *displs*

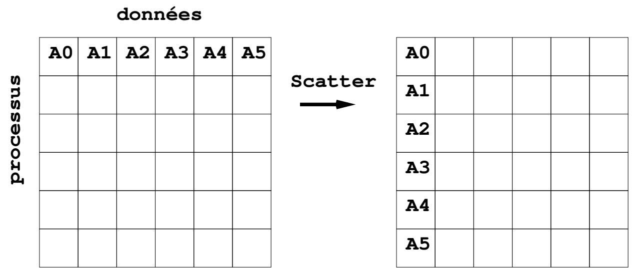
  - équivalent à

MPI_Send(sendbuf, sendcount, sendtype, root ...)

MPI_Recv(recvbuf+displs[i].extent(recvtype), recvcount[i], recvtype,i ...)

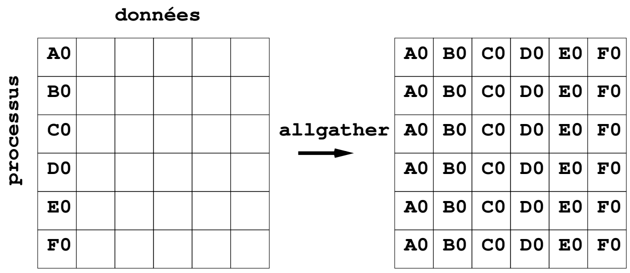
Scatter

- int MPI_Scatter (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
- int MPI_Scatterv (sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)



ALLGATHER

- MPI_Allgather
- MPI_Allgather



ALLTOALL

- MPI_Alltoall
- MPI_Alltoallv

données (colonnes) — processus (lignes)

A0	A1	A2	A3	A4	A5
B0	B1	B2	B3	B4	B5
C0	C1	C2	C3	C4	C5
D0	D1	D2	D3	D4	D5
E0	E1	E2	E3	E4	E5
F0	F1	F2	F3	F4	F5

alltoall $\longrightarrow$

A0	B0	C0	D0	E0	F0
A1	B1	C1	D1	E1	F1
A2	B2	C2	D2	E2	F2
A3	B3	C3	D3	E3	F3
A4	B4	C4	D4	E4	F4
A5	B5	C5	D5	E5	F5

Calculs globaux

➠ Effectuer un calcul à partir de données partagées

- 2 types d'opérations
  - réduction (reduce)
  - préfixe parallèle (scan)
- n est la taille du groupe
- $D_{i,j}$ est la j^{eme} donnée dans le processus i
- $\oplus$ est la fonction de calcul
- La routine de réduction évalue :

$$D_j = D_{0,j} \oplus D_{1,j} \oplus ... \oplus D_{n-1,j}$$

- $\oplus$ doit être associative

- La routine de préfixe parallèle évalue pour le processus k:

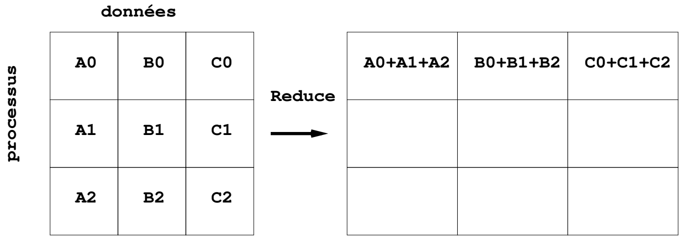
$$D_{k,j} = D_{0,j} \oplus D_{1,j} \oplus ... \oplus D_{k,j}$$

- La fonction de calcul peut-être :
  - Une fonction prédéfinie
  - Une fonction fournie par l'utilisateur

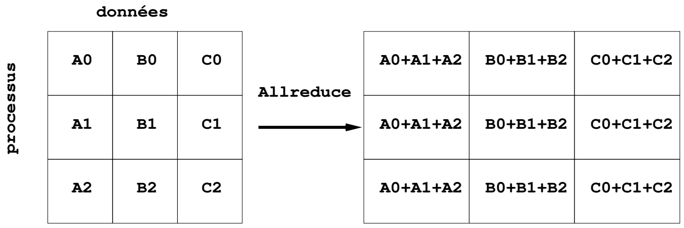
- Les résultats D_j peuvent être rangés:
  - dans un processus fixé (reduce)
  - sur tous les processeurs (allreduce)
  - distribués sur les processeurs (reduce-scatter)
- Chaque processus peut envoyer un élément ou une séquence d'éléments

  (opérations éxécutées séparemment sur chaque élément de la séquence)

❖ int MPI_Reduce (sendbuf, recvbuf, count, datatype, op, root, comm)



❖ int MPI_Allreduce (sendbuf, recvbuf, count, datatype, op, comm)



Reduce-scatter

✥int MPI_Reduce_scatter (sendbuf, recvbuf, recvcnts, datatype, op, com m)

données / processus

A0	B0	C0	Reduce-scatter →	A0+A1+A2		
A1	B1	C1		B0+B1+B2		
A2	B2	C2		C0+C1+C2		

Scan

✥ int MPI_Scan (sendbuf, recvbuf, count, datatype, op, comm)

données / processus

A0	B0	C0		A0	B0	C0
A1	B1	C1	Scan →	A0+A1	B0+B1	C0+C1
A2	B2	C2		A0+A1+A2	B0+B1+B2	C0+C1+C2

Opérateur MPI	**opération**
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bite-wise and

Opérateur MPI	**opération**
MPI_LOR	logical or
MPI_BOR	bite-wise or
MPI_LXOR	logical xor
MPI_BXOR	bite-wise xor
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

Définir un nouvel opérateur

- int MPI_Op_create(function, commute, op)
- La fonction est supposée *associative*
- *commute* indique si elle est commutative
- La fonction doit être de la forme :

  typedef void MPI_User_function(void *invec, void *inouvec, int * len , MPI_Datatype * datatype)

Exemple d'opérateur : déclaration


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

void myProd(Complex *in,Complex *inout,int *len,MPI_Datatype *dptr )
{
   int i;
   Complex c;

   for (i=0: i< *len; i++) {
     c.real = inout->real*in->real
               - inout->imag*in->imag;
     c.imag = inout->real*in->imag
               + inout->imag*in->real;
     *inout = c;
     in++; inout ++;
   }
}
```

```
/* each process has an array of 100 complexes */
Complex a[100], answer[100];
MPI_Op myop;
MPI_Datatype ctype;

MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
MPI_Type_commit(&ctype);

MPI_Op_create(myProd, True, &myOp);

MPI_Reduce(a, answer, 100, ctype, myOp, root, comm);
```


Topologies

MPI

Pourquoi des topologies 110

- Dans de nombreuses applications les processus sont rangés suivant une topologie particulière
- Moyen pratique de nommage des processus
- Simplifie l'écriture des codes
- Augmente la lisibilité
- Permet des optimisations au niveau runtime
- !!! Différent de la topologie réelle de la machine
- Peut cependant aider le mapping virtuel $\rightarrow$ réel

Comment ça marche ? 111

- MPI supporte les topologie virtuelles décrites par un *graphe*

  Un processus est un noeud

  Deux processus qui communiquent sont reliés par un arc

- MPI supporte également les *grilles cartésiennes* (anneaux, grilles, tores, ...)
- Possibilité de *correspondance* entre le rang dans le groupe et les coordonnées pour la topologie
- Créer une nouvelle topologie provoque la création d'un communicateur

- int MPI_Cart_create (comm_old, ndims, dims, periods, reorder, comm_cart)
  Création d'une topologie cartésienne de dimension arbitraire

  dims: nombre de processus de chaque dimension

  periods: périodicité de chaque dimension

  reorder: reordonné ou non

- int MPI_Cartdim_get (comm, ndims)
  nombre de dimensions d'une topologie cartésienne
- int MPI_Cart_rank (comm, coords, rank)
  mapping des coordonnées vers le rang
- int MPI_Cart_coords (comm, rank, maxdims, coords)
  mapping du rang vers les coordonnées

❖ int MPI_Cart_shift (comm, direction, displ, source, dest)
Le shift est souhaité selon la dimension *direction* avec un pas *displ*
Source et *dest* sont calculés automatiquement et peuvent être réutilisés directement

Exemple

Chaque colonne i d'une matrice 2D est "shiftée" d'un pas i

La topologie est un tore de dimension 3 x 4

Un tableau a de réels est distribué sur les processeurs

Chaque processeur possède un élément du tableau

0 (0,0) 0 / 0	1 (0,1) 9 / 5	2 (0,2) 6 / 10	3 (0,3) 3 / 3
4 (1,0) 4 / 4	5 (1,1) 1 / 9	6 (1,2) 10 / 2	7 (1,3) 7 / 7
8 (2,0) 8 / 8	9 (2,1) 5 / 1	10 (2,2) 2 / 6	11 (2,3) 11 / 11

rang du processus
(ligne, colonne)
source / dest

0	9	6	3
4	1	10	7
8	5	2	11

résultat

```
INTEGER comm_2d, rank, coords(2), ierr, source, dest
INTERGER status(MPI_STATUS_SIZE), dims(2)
LOGICAL reorder, periods(2)
REAL a,b

CALL MPI_COMM_SIZE(MPI_COMM_WORLD, isize, ierr)
IF (isize.LT.12) CALL MPI_ABORT(MPI_COMM_WORLD,
                                 MPI_ERR_OTHER,ierr)

CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)

! initialize arrays
a = rank
b = -1
```



```
! create topology
! values to run on 12 processes
dims(1) = 3
dims(2) = 4
periods(1) = TRUE
periods(2) = TRUE
reorder = FALSE
CALL MPI_CART_CREATE(MPI_COMM_WORLD, 2, dims, periods,
                        reorder, comm_2d, ierr)
! rank are the same as in MPI_COMM_WORLD

! find Cartesian coordinates
CALL MPI_CART_COORDS(comm_2d, rank, 2, coords, ierr)
! compute shift source and destination
CALL MPI_CART_SHIFT(comm_2d, 0, coords(2), source, dest,
                        ierr)

! skew a into b
CALL MPI_SENDRECV(a,1, MPI_REAL, dest, 13, b, 1, MPI_REAL,
                      source, 13, comm_2d, status, ierr)
```

❖ int MPI_Cart_sub (comm, remain_dims, comm_new)
Partitionne le communicateur *comm* en topologies cartésiennes de dimensions inférieures définissant autant de nouveaux communicateurs disjoints

comm_new est le nouveau communicateur *associé au processus appelant*

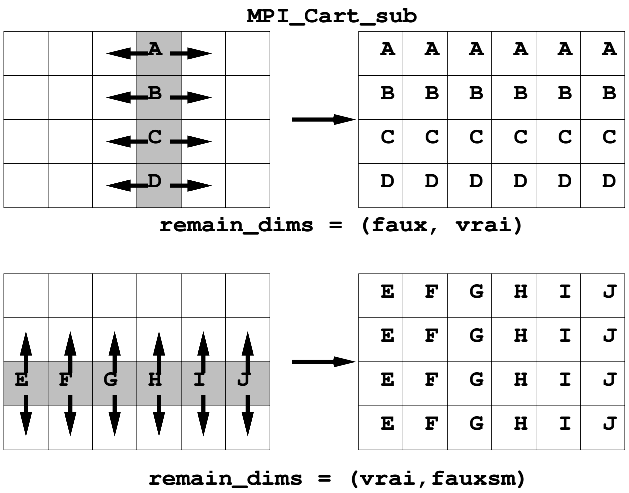
Exemple :

La topologie de *comm* est une grille 2 x 3 x 4

remain_dims = (true, false, true)

Appel à MPI_Cart_sub (comm, remain_dims, comm_new)

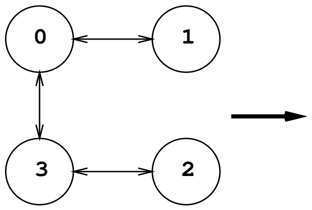
➠ 3 communicateurs avec chacun 8 processus et une topologie 2 x 4



topologies décrites par un graphe

✥ int MPI_Graph_create (comm_old, nnodes, index, edges, reorder, comm_g raph)

nnodes: nombre total de noeuds
index[i]: nombre de voisins des i premiers noeuds
edges: énumération des voisins




```
nnodes =
index = (2,3,4,6)
edges = (1,3,0,3,0,2)
```


Manipulation des graphes

✥ int MPI_Graphdims_get (comm, nnodes, nedges)
Retourne le nombre de noeuds et le nombre d'arcs du graphe

✥ int MPI_Graph_get (comm, maxindex, maxedges, index, edges)
Retourne les deux tableaux *index* et *edges* décrivant le graphe
(MPI_Graphdims_get fournit des valeurs pour les maximums)

- int MPI_Graph_neighbors_count (comm, rank, nneighbors)
  Nombre de voisins pour un noeud donné
- int MPI_Graph_neighbors (comm, rank, maxneighbors, neighbors)
  Liste des voisins d'un noeud donné
- int MPI_Topo_test (comm, top_type)
  - MPI_GRAPH
  - MPI_CART
  - MPI_UNDEFINED