Formation en Calcul Scientifique - LEM2I

Introduction à MPI

Loïc Gouarin

Laboratoire de Mathématique d'Orsay

9-13 avril 2012

Plan

1. Introduction
2. Environnement
3. Message et communication point à point
4. Communication collective
5. Types dérivés
6. Communicateurs
7. Topologies

1 Introduction

2 Environnement

3 Message et communication point à point

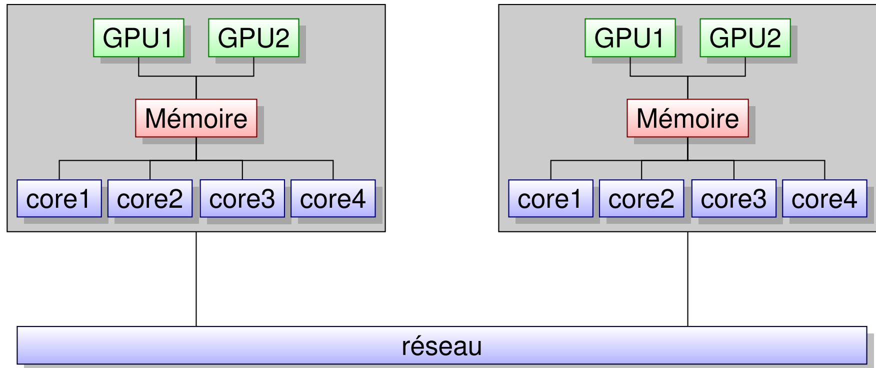
4 Communication collective

5 Types dérivés

6 Communicateurs

7 Topologies

Comment paralléliser son code ?



Modèle de programmation par échange de messages

Chaque processus exécute un sous-programme

- qui est écrit dans un langage classique (C, Fortran, ...),
- qui est généralement le même,
- qui peut avoir des parties dédiées à un processus.

Les variables de chaque sous-programme ont

- le même nom,
- mais appartiennent à des espaces mémoires différents et correspondent à des données différentes,
- elles sont donc privées.

Un processus *i* peut communiquer ses données au processus *j* via des fonctions spéciales d'envoi et de réception.

Modèles d'exécution

Modèle d'exécution SPMD

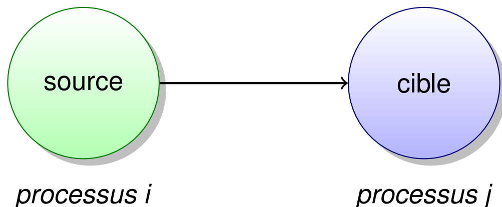
- Single Program Multiple Data.
- Le même programme est exécuté sur des données différentes.

Modèle d'exécution MPMD

- Multiple Program Multiple Data.
- Différents programmes sont exécutés sur des données différentes.

Concepts d'échange de messages

- Un processus *i* envoie des données *source* au processus *j*.
- Le processus *j* reçoit des données du processus *i* et les met dans *cible*.



Concepts d’échange de messages

Afin de transmettre correctement un message, certaines informations sont nécessaires

- l’expéditeur,
- le destinataire,
- le type du message,
- sa taille,
- le message.

MPI

- Message Passing Interface : bibliothèque portable, efficace et flexible d'échange de messages,
- conçue en 1993,
- norme définissant une bibliothèque de fonctions, utilisable avec les langages C et Fortran,
- permet d'exploiter des ordinateurs distants ou multiprocesseurs par passage de messages.

Les fonctionnalités de MPI

Fonctionnalités de MPI-1

- environnement,
- communications point à point,
- communications collectives,
- communicateurs,
- types dérivées,
- topologies.

Les fonctionnalités de MPI

Fonctionnalités de MPI-2

- gestion dynamique des processus,
- I/O parallèle,
- interfaçage avec Fortran95 et C++,
- extension des communications collectives aux intercommunicateurs,
- communications de mémoire à mémoire,
- ...

Historique et évolutions

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

1 Introduction

2 Environnement

3 Message et communication point à point

4 Communication collective

5 Types dérivés

6 Communicateurs

7 Topologies

Fichier d'en-tête

Pour pouvoir utiliser les fonctionnalités de la librairie MPI, il est nécessaire d'ajouter une en-tête dans nos programmes

En C

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

En Fortran

- MPI-1 : `include 'mpif.h'`
- MPI-2 : `use mpi`

Format des fonctions

En C

```
error = MPI_Xxxxxx(parameter, ...);
MPI_Xxxxxx(parameter, ...);
```

En Fortran

```
CALL MPI_XXXXXX(parameter, ..., IERROR)
```

Initialisation de MPI

On utilisera la fonction suivante

```
MPI_INIT(IERROR)
INTEGER    IERROR
```

Exemple :

```
program testMPI
  use mpi
  implicit none

  integer :: ierr

  call MPI_Init(ierr)
  ...
```

Finalisation de MPI

On utilisera la fonction suivante

```
MPI_FINALIZE(IERROR)
  INTEGER    IERROR
```

Exemple :

```
program testMPI
  use mpi
  implicit none

  integer :: ierr

  call MPI_Init(ierr)
  ...
  call MPI_Finalize(ierr)

end program testMPI
```

Compilation

La compilation se fait via les commandes

- mpicc
- mpic++
- mpif77
- mpif90

```
terminal$ mpif90 -O3 testMPI.f90 -o test
```

Exécution

L'exécution se fait via les commandes

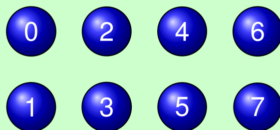
- MPI-1 : mpirun
- MPI-2 : mpiexec

```
terminal$ mpirun -np 2 ./test
terminal$ mpiexec -np 2 ./test
```

Exécution

Lors de l'initialisation de MPI, le communicateur MPI_COMM_WORLD rassemblant l'ensemble des processus est créé.

```
terminal$ mpirun -np 8 ./test
```



MPI_COMM_WORLD

Nombre de processus et rang

Le nombre de processus d'un communicateur est obtenu par la commande

```
MPI_COMM_SIZE(COMM, SIZE, IERROR)
   INTEGER   COMM, SIZE, IERROR
```

Le rang permet de distinguer les processus à l'intérieur d'un communicateur. Il est compris entre 0 et la valeur retournée par MPI_COMM_SIZE-1.

```
MPI_COMM_RANK(COMM, RANK, IERROR)
   INTEGER   COMM, RANK, IERROR
```

Exercice 2

Ecrire un programme MPI où chaque processus affiche à l'écran **Hello word ! !**.

Exercice 3

Ecrire un programme MPI où chaque processus affiche à l'écran

```
Je suis le processus x sur y. Je suis sur la machine z.
```

On utilisera la fonction

```
MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR)

    CHARACTER*(*)   NAME
    INTEGER         RESULTLEN, IERROR
```

1 Introduction

2 Environnement

3 Message et communication point à point

4 Communication collective

5 Types dérivés

6 Communicateurs

7 Topologies

Message

Un message contient un certain nombre d'éléments d'un certain type.

MPI fournit

- des types de base
- des types dérivés

Les types dérivés sont construits à partir des types de base et/ou d'autres types dérivés.

Les types C sont différents des types Fortran.

Les types de base en C

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

Les types de base en Fortran

types de données MPI	types de données Fortran
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE_PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER
MPI_BYTE	
MPI_PACKED	

Communication point à point

- Communication entre 2 pocessus : un expéditeur et un destinataire.
- La communication a lieu à l'intérieur d'un communicateur.
- Les processus liés à cette communication sont identifiés par leur rang dans ce communicateur.

Envoi d'un message

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM,
         IERROR)

    <type>     BUF(*)
    INTEGER    COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

- BUF, COUNT et DATATYPE définissent le message.
- DEST est le processus destinataire.
- TAG est un entier permettant d'identifier le message.
- COMM est le communicateur à l'intérieur duquel a lieu l'envoi.

Réception d'un message

```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,
         STATUS, IERROR)

<type>     BUF(*)
INTEGER    COUNT, DATATYPE, SOURCE, TAG, COMM
INTEGER    STATUS(MPI_STATUS_SIZE), IERROR
```

- BUF, COUNT et DATATYPE définissent le buffer de réception.
- SOURCE est le processus expéditeur.
- TAG est un entier permettant d'identifier le message.
- COMM est le communicateur à l'intérieur duquel a lieu l'envoi.
- STATUS contient des informations sur la réception.

Validation de la communication

Pour que l'échange réussisse,

- l'expéditeur doit donner un rang de destinataire valide ;
- le destinataire doit donner un rang de destinataire valide ;
- le communicateur doit être le même ;
- les tags doivent être les mêmes ;
- les types de données doivent correspondre ;
- le buffer de réception doit être suffisamment grand pour recevoir le message.

Enveloppe du message

L'enveloppe d'un message est définie par

- le rang de l'expéditeur,
- le rang du destinataire,
- l'étiquette (tag) du message,
- le communicateur.

On retrouve certaines de ces informations dans le tableau status

- status(MPI_SOURCE),
- status(MPI_TAG),
- nombre de valeurs reçues

```
MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER   STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
```

Autres possibilités

- On peut recevoir un message de n'importe quelle source

  `MPI_ANY_SOURCE`
- On peut recevoir un message muni de n'importe quelle étiquette

  `MPI_ANY_TAG`
- On peut envoyer un message à un processus n'existant pas

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

  `MPI_Sendrecv`
- On peut échanger des données en une seule communication

  `MPI_Sendrecv_replace`

Pourquoi, comment optimiser les communications point à point ?

- Important d'optimiser les communications : gain en performance.
  $\Rightarrow$ Minimiser le temps passé à faire autre chose que des calculs (i.e. l'overhead)
- Différentes possibilités
  - recouvrir les communications par des calculs,
  - éviter la recopie du message dans une zone mémoire temporaire,
  - minimiser les surcoûts dûs aux appels répétés aux routines de communications.

Les différents modes de communication fournis par MPI

- Standards
  MPI effectue selon la taille du message une copie temporaire ou non.
  - Si copie : l'envoi se termine lorsque la copie est achevée.
  - Sinon : l'envoi se termine lorsque la réception a commencé.
- Synchrones
  L'envoi se termine lorsque la réception a commencé.
- Bufferisées
  Le programmeur effectue une copie temporaire du message. L'envoi se termine lorsque la copie temporaire est achevée.
- Ready

Les différents modes de communication fournis par MPI

modes	bloquant	non bloquant
envoi standard	MPI_Send	MPI_Isend
envoi synchrone	MPI_Ssend	MPI_Issend
envoi bufferisé	MPI_Bsend	MPI_Ibsend
réception	MPI_Recv	MPI_Irecv

Un peu plus sur les buffers

- Lors de l'utilisation de MPI_Bsend ou de MPI_Ibsend, le message est recopié dans un buffer alloué par le programmeur avant d'être envoyé.
- Les allocations des buffers mémoires MPI sont gérées avec

```
MPI_BUFFER_ATTACH(BUF, SIZE, IERROR)
MPI_BUFFER_DETACH(BUF, SIZE, IERROR)
```

```
    <type>     BUF(*)
    INTEGER    SIZE, IERROR
```

- Le buffer est alloué dans la mémoire locale du processus expéditeur.
- Le buffer est uniquement utilisable pour les messages bufferisés.
- Un seul buffer alloué à la fois par processus.

Les communications non-bloquantes

Les communications bloquantes empêchent le programme de faire autre chose.

Les communications non bloquantes permettent de séparer l'initialisation de l'envoi ou de la réception et la fin (lorsque tout est prêt).

Elles se déroulent donc en deux étapes :

- une étape d'initialisation : MPI_Ixsend et MPI_Irecv
- une étape de finalisation : MPI_Wait, MPI_Test et MPI_Probe

Etape d'initialisation

```
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM,
          REQUEST, IERROR)
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,
          REQUEST, IERROR)

   <type>     BUF(*)
   INTEGER    COUNT, DATATYPE, SOURCE, DEST, TAG, COMM,
              REQUEST, IERROR
```

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

Etape de finalisation

- Attendre qu'une requête soit terminée

```
MPI_WAIT(REQUEST, STATUS, IERROR)
    INTEGER   REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

- Tester si une requête est terminée

```
MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
    LOGICAL   FLAG
    INTEGER   REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

- Contrôler sans réceptionner si une requête est arrivée

```
MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
    INTEGER   SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

  Version non bloquante : MPI_Iprobe

Etape de finalisation

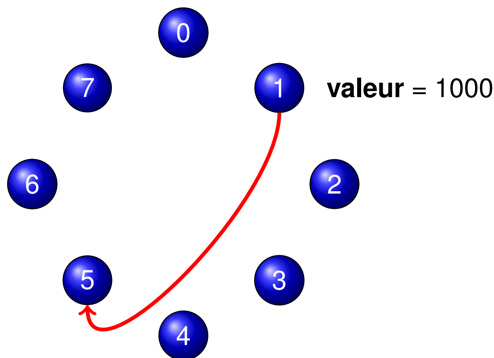
Il existe des variantes pour tester des groupes de communication

	Attendre	Tester
Au moins une renvoie exactement une	MPI_Waitany	MPI_Testany
toutes	MPI_Waitall	MPI_Testall
Au moins une renvoie toutes celles finies	MPI_Waitsome	MPI_Testsome

L'étape de finalisation désalloue la requête d'une communication non bloquante qui est achevée.

Exercice 4

Ecrire le programme MPI correspondant à la figure suivante



Exercice 5

On dispose de 2 processus. Chaque processus dispose d'un tableau de *double*.

Ecrire un programme où chaque processus envoie son tableau à l'autre en utilisant

- MPI_Send et MPI_Recv,
- MPI_Ssend et MPI_Recv,
- MPI_Bsend et MPI_Recv,
- MPI_Sendrecv.

Exercice 6

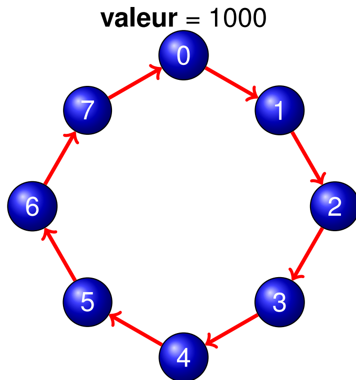
On aimerait calculer les taux de transfert pour chacune des méthodes implémentées précédemment. Pour ce faire, nous allons utiliser la fonction

```
double MPI_Wtime( void )
```

- Allouer un tableau de **double** occupant 1 Mo (104 857 600 octets).
- Faire une boucle qui calcule le temps pour envoyer 8, 16, 32, 64, ... octets.
- Stocker les résultats dans un fichier.
- Représenter les courbes via *gnuplot*.

Exercice 6

Ecrire un programme MPI réalisant une communication en anneau.
Le processus 0 a une valeur initialisée à 1000 et envoie à son voisin, le voisin rajoute 1 à cette valeur et envoie au suivant...



Exercice 7(suite)

- Utiliser MPI_Ssend et MPI_Recv,
- Utiliser MPI_Isend et MPI_Recv,
- Utiliser MPI_Isend et MPI_Irecv.

1 Introduction

2 Environnement

3 Message et communication point à point

4 Communication collective

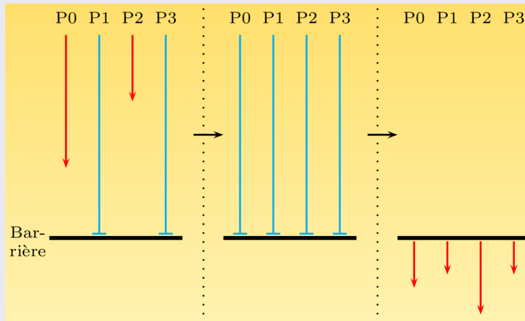
5 Types dérivés

6 Communicateurs

7 Topologies

- La communication collective est une communication qui implique un ensemble de processus, tous ceux du communicateur fourni en argument.
- En une seule opération, on effectue une série de communications point à point.
- Tous les processus du communicateur doivent appeler la fonction.
- Ces communications sont bloquantes.
- Il n'y a pas d'étiquettes.
- Les buffers de réception doivent tous avoir la même taille.

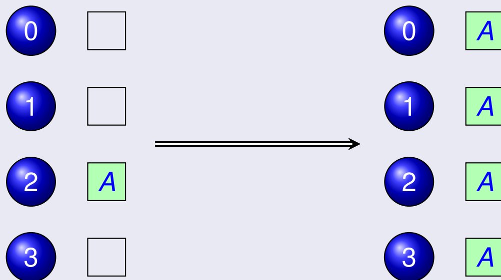
Synchronisation globale




```
MPI_BARRIER(COMM, IERROR)

    INTEGER   COMM, IERROR
```

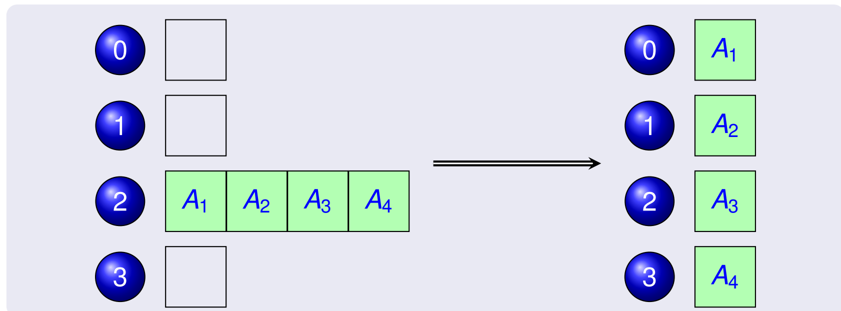
Diffusion générale

```
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)

<type>     BUFFER(*)
INTEGER    COUNT, DATATYPE, ROOT, COMM, IERROR
```



Communication dispersive

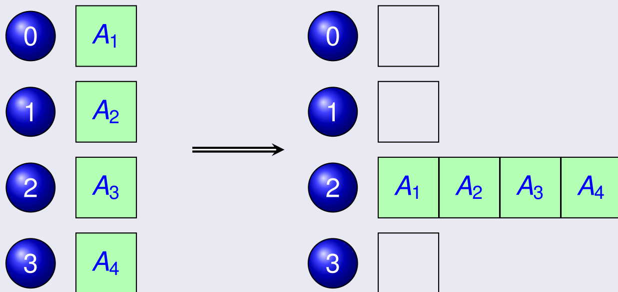
```
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
            RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)

<type>     SENDBUF(*), RECVBUF(*)
INTEGER    SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT
INTEGER    COMM, IERROR
```



Rassembler

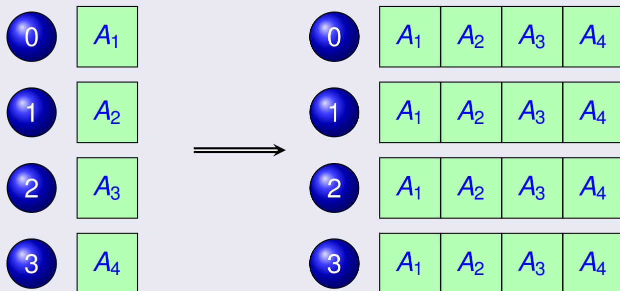
```
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
           RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
```

```
<type>     SENDBUF(*), RECVBUF(*)
INTEGER    SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT
INTEGER    COMM, IERROR
```



Tout rassembler

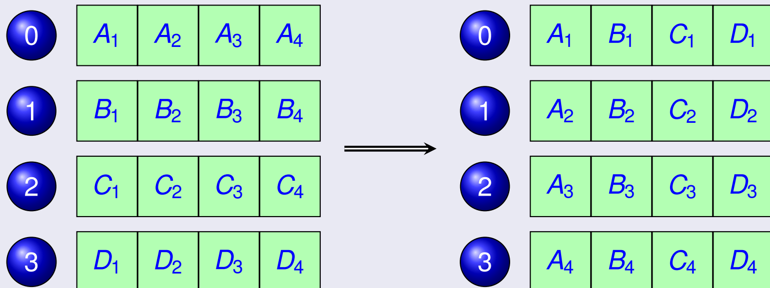
```
MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
              RECVCOUNT, RECVTYPE, COMM, IERROR)

<type>     SENDBUF (*), RECVBUF (*)
INTEGER    SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM,
INTEGER    IERROR
```



Echanges croisés

```
MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
             RECVCOUNT, RECVTYPE, COMM, IERROR)
```

```
<type>     SENDBUF(*), RECVBUF(*)
INTEGER    SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE
INTEGER    COMM, IERROR
```



Remarque importante

On a supposé que les données à diffuser et à collecter étaient de même taille.
Si ce n'est pas le cas, il faut utiliser les fonctions MPI_Scatterv, MPI_Gatherv, MPI_AllGatherv, MPI_Alltoallv.

Pour plus d'informations sur ces fonctions, ouvrir un terminal et taper par exemple la commande suivante

```
man MPI_Scatterv
```

Réduction

```
MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP,
           ROOT, COMM, IERROR)
```

```
<type>     SENDBUF(*), RECVBUF(*)
INTEGER    COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```

OP représente l'opération que l'on veut faire sur les données réparties SENDBUF.
Le résultat sera mis dans RECVBUF du processus ROOT.

Opérations de réduction

- MPI_SUM : somme des éléments
- MPI_PROD : produit des éléments
- MPI_MAX : recherche du maximum
- MPI_MIN : recherche du minimum
- MPI_MAXLOC : recherche de l'indice du maximum
- MPI_MINLOC : recherche de l'indice du minimum
- MPI_LAND : ET logique
- MPI_LOR : OU logique
- MPI_LXOR : OU exclusif logique

Réduction globale

```
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE,
              OP, COMM, IERROR)

<type>    SENDBUF(*), RECVBUF(*)
INTEGER   COUNT, DATATYPE, OP, COMM, IERROR
```

Exercice 8

On souhaite paralléliser le produit de deux matrices. Soit le code séquentiel suivant

```
program matMult
  implicit none
  integer, parameter :: n = 10
  real(kind=8), allocatable, dimension(:,:) :: A, B, C

  allocate(A(n, n), B(n, n), C(n, n))
  call random_number(A)
  call random_number(B)
  C = matmul(A, B)
  deallocate(A, B, C)

end program matMult
```

Exercice 8

Ecrire le code parallèle qui effectue les opérations suivantes :

- le processus 0 initialise les matrices A et B,
- le processus 0 envoie la matrice A à tout le monde,
- le processus 0 envoie un bloc de colonnes à chaque processus,
- chaque processus calcule son produit matriciel,
- chaque processus envoie sa partie au processus 0.

On commencera par vérifier le résultat pour $n = 10$ et 2 processus.
On relancera le code avec 3 processus.

Que se passe-t-il ? Corriger.

Exercice 9

Reprendre l'algorithme pour calculer π et le paralléliser.

Indications :

- découper la boucle,
- utiliser une réduction pour calculer le résultat final.

1 Introduction

2 Environnement

3 Message et communication point à point

4 Communication collective

5 Types dérivés

6 Communicateurs

7 Topologies

Pour quoi faire ?

Pour le moment, on sait envoyer et recevoir des données qui ont le même type et qui sont contiguës en mémoire.

La construction de types dérivés dans MPI permet d'envoyer et recevoir une partie d'un tableau, une structure, ...

Exemple

Nous avons trois paramètres (2 réels et un entier) que seul le processus 0 connaît. Nous voudrions transmettre ces paramètres via un Bcast à tous les autres processus.

- On peut faire 3 messages : très coûteux.
- On peut créer un type dérivé.
- On peut utiliser MPI_Pack et MPI_Unpack.

Définir un nouveau type

```
MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,
                       ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES,
                       NEWTYPE, IERROR)

    INTEGER    COUNT, ARRAY_OF_BLOCKLENGTHS(*)
    INTEGER    ARRAY_OF_DISPLACEMENTS(*)
    INTEGER    ARRAY_OF_TYPES(*), NEWTYPE, IERROR
```

- COUNT : nombre de blocs,
- ARRAY_OF_BLOCKLENGTHS : nombre d'éléments dans chacun des blocs,
- ARRAY_OF_DISPLACEMENTS : nombre de bytes entre chaque bloc,
- ARRAY_OF_TYPES : type de chaque bloc.

Validation et destruction d'un nouveau type

- Validation

```
MPI_TYPE_COMMIT(DATATYPE, IERROR)
    INTEGER   DATATYPE, IERROR
```

- Destruction

```
MPI_TYPE_FREE(DATATYPE, IERROR)
    INTEGER   DATATYPE, IERROR
```

Exemple

```
program newStruct
  use mpi
  implicit none

  type MyStruct
     real(kind=8) :: a, b
     integer :: n
  end type MyStruct
...
  types = (/MPI_DOUBLE_PRECISION, &
            MPI_DOUBLE_PRECISION, &
            MPI_INTEGER/)
  longueurs_blocs = (/1, 1, 1/)
```

Exemple

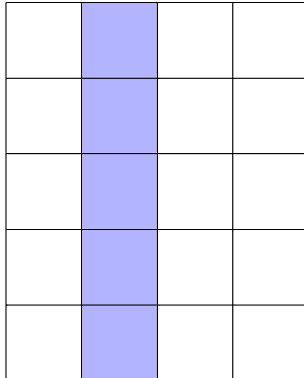
```
call MPI_Get_address(p%a, adresses(1), ierr)
call MPI_Get_address(p%b, adresses(2), ierr)
call MPI_Get_address(p%n, adresses(3), ierr)

do i=1,3
   deplacements(i) = adresses(i) - adresses(1)
end do

call MPI_TYPE_CREATE_STRUCT ( &
     3, longueurs_blocs, deplacements, &
     types, type_MyStruct, ierr)

call MPI_Type_commit(type_MyStruct, ierr)
```

Exemple

```
  if (rank == 0) then
     p%a = 1.d0
     p%b = 2.d0
     p%n = 12
  end if

  call MPI_Bcast(p, 1, type_MyStruct, 0, &
                 MPI_COMM_WORLD, ierr)

  if (rank == 1) print*, p

  call MPI_Type_free(type_MyStruct, ierr)
  call MPI_Finalize(ierr)
end program newStruct
```

Valeurs contiguës

Nous avons une matrice de double et nous voudrions envoyer juste un bloc de celle-ci à un autre processus.



Valeurs contiguës

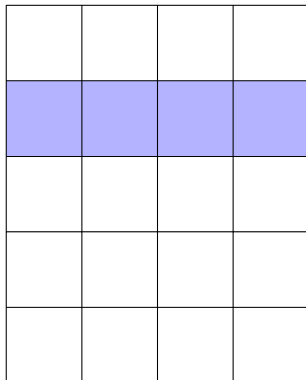
```
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)

    INTEGER   COUNT, OLDTYPE, NEWTYPE, IERROR
```

Exemple

```
call MPI_Type_contiguous(5, MPI_DOUBLE_PRECISION,&
                           colType, ierr)
call MPI_Type_commit(colType, ierr)
```

Valeurs à pas constant

Nous avons une matrice de double et nous voudrions envoyer juste un bloc de celle-ci à un autre processus.



Valeurs à pas constant

```
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE,
                OLDTYPE, NEWTYPE, IERROR)

   INTEGER   COUNT, BLOCKLENGTH, STRIDE, OLDTYPE
   INTEGER   NEWTYPE, IERROR
```

Exemple

```
call MPI_Type_vector(4, 1, 5, MPI_DOUBLE_PRECISION, &
                     lineType, ierr)
call MPI_Type_commit(lineType, ierr)
```

Valeurs distantes d'un pas constant

Nous avons une matrice de double et nous voudrions envoyer juste un bloc de celle-ci à un autre processus.

Valeurs distantes d'un pas constant

Exemple

```
call MPI_Type_vector(3, 2, 5, MPI_DOUBLE_PRECISION, &
                     blocType, ierr)
call MPI_Type_commit(blocType, ierr)
```

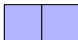
valeurs distantes d'un pas variable

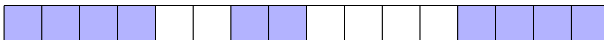
```
MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,
                 ARRAY_OF_DISPLACEMENTS, OLDTYPE,
                 NEWTYPE, IERROR)
```

```
INTEGER   COUNT, ARRAY_OF_BLOCKLENGTHS(*)
INTEGER   ARRAY_OF_DISPLACEMENTS(*), OLDTYPE
INTEGER   NEWTYPE, IERROR
```

- ARRAY_OF_BLOCKLENGTHS : taille des blocs en nombre d'éléments.
- ARRAY_OF_DISPLACEMENTS : tableau représentant le début de chaque bloc exprimé en nombre d'éléments.

valeurs distantes d'un pas variable

ancien 

nouveau 

```
blocklens = (/2, 1, 2/)
indices = (/0, 3, 6/)
...
call MPI_Type_indexed(3, blocklens, indices, &
                      oldType, newType, ierr)
call MPI_Type_commit(newType, ierr)
...
```

Autres possibilités

- MPI_Type_hvector
  Même chose que MPI_Type_vector sauf que les tailles sont données en nombre d'octets.
- MPI_Type_hindexed
  Même chose que MPI_Type_indexed sauf que les tailles sont données en nombre d'octets.
- MPI_Type_create_subarray
  Extraction d'un sous-tableau.

Un mot sur pack et unpack

Reprenons notre exemple avec nos 3 paramètres.

```
  character*100 :: buffer
  integer :: position=0
...
  if (rank == 0) then
     a = 1.d0
     b = 2.d0
     n = 12
     call MPI_Pack(a, 1, MPI_DOUBLE_PRECISION, buffer,&
                   100, position, MPI_COMM_WORLD, ierr)
     call MPI_Pack(b, 1, MPI_DOUBLE_PRECISION, buffer,&
                   100, position, MPI_COMM_WORLD, ierr)
     call MPI_Pack(n, 1, MPI_INTEGER, buffer, 100,&
                   position, MPI_COMM_WORLD, ierr)
  end if
```

Un mot sur pack et unpack

Reprenons notre exemple avec nos 3 paramètres.

```
  call MPI_Bcast(buffer, 100, MPI_PACKED,
                 0, MPI_COMM_WORLD, ierr)

  if (rank /= 0) then
    call MPI_Unpack(buffer, 100, position, a, 1,&
          MPI_DOUBLE_PRECISION, MPI_COMM_WORLD, ierr)
    call MPI_Unpack(buffer, 100, position, b, 1,&
          MPI_DOUBLE_PRECISION, MPI_COMM_WORLD, ierr)
    call MPI_Unpack(buffer, 100, position, n, 1,&
          MPI_INTEGER, MPI_COMM_WORLD, ierr)
 end if
  ...
```

Que faut-il utiliser ?

- Si on envoie et on reçoit des données contiguës de même type

  on utilisera les types de bases.

- Si on envoie et on reçoit très peu de données de types différents

  on utilisera MPI_Pack et MPI_Unpack.

- Si on envoie et on reçoit régulièrement des données de types différents ou non contiguës

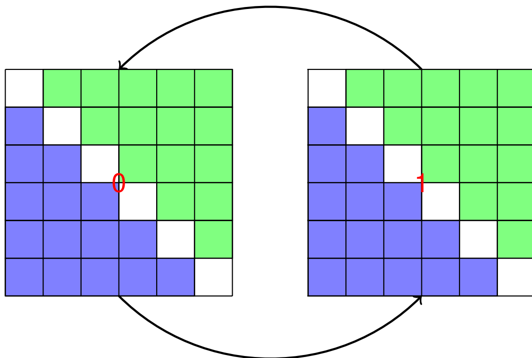
  on utilisera les types dérivés.

Exercice 10

- Reprendre l'exercice sur l'anneau.
- Envoyer un entier et un double en créant un nouveau type de données.
- Faire de même en utilisant les fonctions MPI_Pack et MPI_Unpack.

Exercice 11

Construire un type dérivé permettant de réaliser l'opération suivante



1 Introduction

2 Environnement

3 Message et communication point à point

4 Communication collective

5 Types dérivés

6 Communicateurs

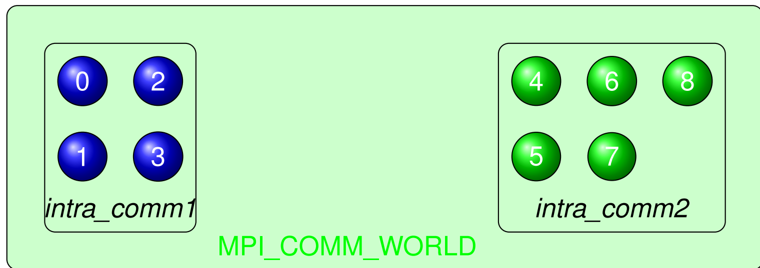
7 Topologies

Introduction

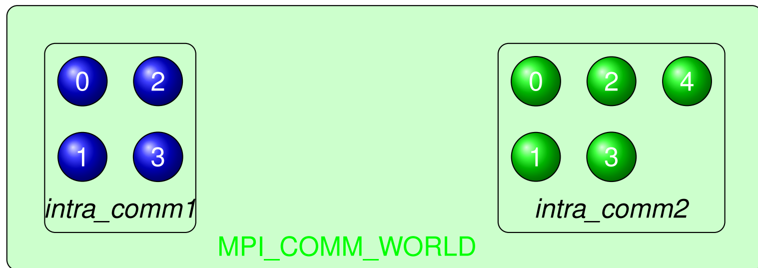
- MPI fournit un communicateur par défaut (MPI_COMM_WORLD) rassemblant l'ensemble des processus.
- Il est possible d'en créer pour regrouper juste une partie des processus.
- On distingue deux types de communicateurs
  - intra-communicateurs : pour les opérations sur un groupe de processus au sein d'un communicateur ;
  - inter-communicateurs : pour les communications entre deux groupes de processus.

Introduction

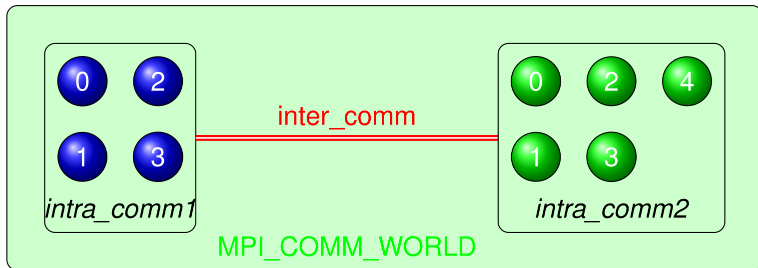
Introduction



Introduction



Introduction



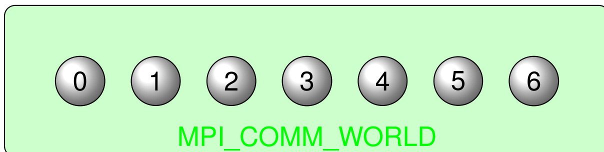
Communicateur issu d'un autre communicateur

```
MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)

    INTEGER   COMM, COLOR, KEY, NEWCOMM, IERROR
```

- COMM : le communicateur à l'intérieur duquel on va créer un nouveau communicateur.
- COLOR : entier positif permettant d'assigner un processus à un sous ensemble.
- KEY : entier permettant d'assigner un rang au processus dans ce sous ensemble.
- NEWCOMM : nouveau communicateur.

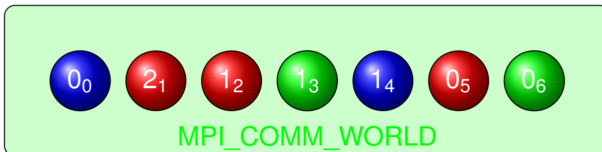
Exemple

rang dans COMM_WORLD	0	1	2	3	4	5	6
couleur	0	1	1	2	0	1	2
clé	0	10	8	1	0	5	0
nouveau rang	0	2	1	1	1	0	0



Exemple

rang dans COMM_WORLD	0	1	2	3	4	5	6
couleur	0	1	1	2	0	1	2
clé	0	10	8	1	0	5	0
nouveau rang	0	2	1	1	1	0	0



Remarques

- Si un processus a la couleur MPI_UNDEFINED alors il n'apartiendra à aucun sous communicateur.
- Pour détruire un communicateur, il suffit d'utiliser la commande

```
MPI_COMM_FREE(COMM, IERROR)
    INTEGER   COMM, IERROR
```

Exercice 12

Créer un communicateur pour les processus pairs et un communicateur pour les processus impairs.

1 Introduction

2 Environnement

3 Message et communication point à point

4 Communication collective

5 Types dérivés

6 Communicateurs

7 Topologies

Introduction

- Les domaines de calcul sont souvent des grilles.
- En décomposition de domaine, on découpe en petits domaines notre grand domaine.
- MPI fournit deux types de topologies
  - Topologies cartésiennes permettant de décrire rapidement des découpages de grilles en 2D et en 3D.
  - Topologies par graphes permettant de décrire des géométries plus complexes.

Création d'une grille cartésienne

```
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER,
                COMM_CART, IERROR)

INTEGER   COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
LOGICAL   PERIODS(*), REORDER
```

- NDIMS : nombre de dimensions de la grille cartésienne.
- DIMS : tableau d'entiers de taille dim spécifiant le nombre de processus dans chaque dimension.
- PERIODS : tableau de booléens de taille dim spécifiant si la dimension est périodique ou non.
- REORDER : booléen indiquant si les rangs doivent être réordonnés.

Exemple

```
integer :: cart_comm
integer :: ierr, ndims = 2
integer, dimension(2) :: dims, coords
logical, dimension(2) :: periods

dims = (/3, 2/)
period = (/.true., .false./)

call MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, &
                     period, .false., cart_comm, ierr)
```

Exemple

4	0	2	4	2
(-1, 0)	*(0, 0)*	*(1, 0)*	*(2, 0)*	*(3, 0)*
5	1	3	5	3
(-1, 1)	*(0, 1)*	*(1, 1)*	*(2, 1)*	*(3, 1)*

Correspondance rang et coordonnées

- Obtenir le rang du processus à partir de ses coordonnées

```
MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
    INTEGER   COMM, COORDS(*), RANK, IERROR
```

- Obtenir les coordonnées du processus à partir de son rang

```
MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
    INTEGER   COMM, RANK, MAXDIMS, COORDS(*), IERROR
```

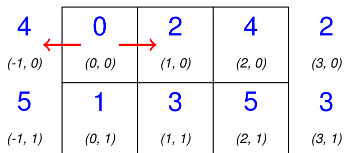
Les voisins

```
MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE,
                    RANK_DEST, IERROR)

    INTEGER   COMM, DIRECTION, DISP, RANK_SOURCE
    INTEGER   RANK_DEST, IERROR
```

- DIRECTION : axe de recherche
- DISP : déplacement

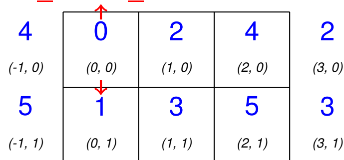
Les voisins pour le processus 0

```
integer :: direction = 0
integer :: disp = 1
```



Les voisins pour le processus 0

```
integer :: direction = 1
integer :: disp = 1
```



Choix du découpage

Dans une topologie cartésienne, la fonction MPI_Dims_create retourne les dimensions dans chaque direction en fonction du nombre de processus.

```
MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
    INTEGER   NNODES, NDIMS, DIMS(*), IERROR
```

Si les valeurs de dims sont mises à 0, cela signifie qu'on laisse à MPI le choix du nombre de processus dans chaque direction.

dims avant	appel de la fonction	dims après
(0, 0)	(8, 2, dims)	(4, 2)
(0, 0)	(5, 2, dims)	(5, 2)
(0, 2, 0)	(16, 3, dims)	(4, 2, 2)
(0, 3)	(8, 2, dims)	erreur

Exercice 13

Reprendre l'exercice sur l'anneau et ajouter une topologie pour communiquer avec les voisins.

Références

- Cours de G. Moebs lors de l'ANGD "Calcul parallèle et application aux plasmas froids",
- Cours de l'IDRIS,
- Parallel programming workshop.