une présentation courte de MPI

Xavier JUVIGNY

24 septembre 2023

Table des matières

1 Une introduction à MPI

1.1 Une spécification d'interface

Le standard MPI (Message Passing Interface) est une **spécification** pour les développeurs et les utilisateurs d'une librairie d'échange de messages. En soi, ce n'est pas un bibliothèque, mais plutôt les spécifications de ce qu'une telle bibliothèque devrait être.

MPI propose fondamentalement un *modèle de programmation parallèle par échange de messsage* : les données sont déplacées d'un espace d'adressage d'un processus à l'espace d'adressage d'un autre processus à l'aide d'opérations coopératives sur chacun des processus.

Autrement dit, le but de cet interface est de proposer un standard largement utilisé pour écrire des programmes par échanges de messages. L'interface tente d'être :

- Pratique
- Portable
- Optimisée
- Flexible

Les spécifications de l'interface ont été définies pour les langages C et Fortran 90. La troisième version des spécifications de MPI propose également un support pour les langages Fortran 2003 et Fortran 2008.

En fait, les diverses bibliothèques MPI existantes diffèrent dans les caractéristiques et la version de spécification qu'elles supportent. Les utilisateurs de ces bibliothèques doivent en avoir conscience.

1.2 Modèle de programmation

À l'origine, le standard MPI a été conçu pour des architectures à mémoires distribuées qui avaient commencé à émerger à cette époque (fin des années 1980 – début des années 1990).

Au fur et à mesure du changement des architectures des ordinateurs, les mises en œuvres de MPI ont adapté leurs bibliothèques aux architectures multi-cœurs avec mémoire partagée et aux différents protocoles réseaux.

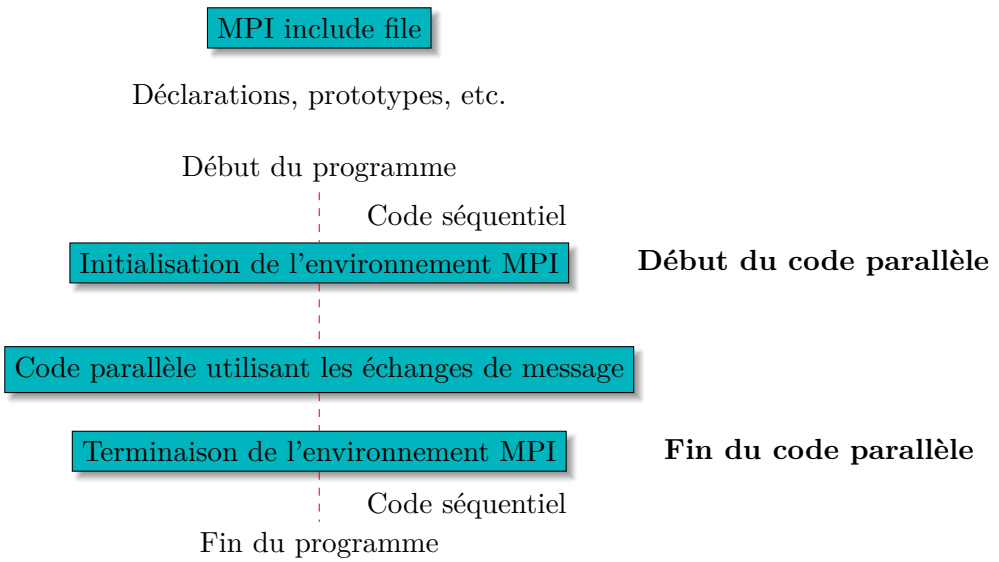
Aujourd'hui, MPI s'exécute sur virtuellement n'importe quelle plateforme :

- en mémoire distribuée ;
- en mémoire partagée ;
- en mémoire hybride (distribuée et partagée).

Néanmoins, le modèle de programmation reste clairement un modèle de **programmation sur mémoire distribuée** sans considération sur l'architecture sous-jacente de la machine.

Tout parallélisme est explicite : le programmeur a la responsabilité d'identifier correctement le parallélisme et doit mettre en œuvre ses algorithmes parallèles en utilisant l'interface MPI.

1.3 Structure générale d'un programme MPI



En Python, il est nécessaire d'importer le module MPI issu de mpi4py :

```
from mpi4py import MPI
```

Les fonctions et méthodes de l'interface MPI de python prennent deux formes différentes :

- Première forme : La première lettre de la fonction ou de la méthode est en majuscule. Dans cette forme, on accède à la forme python optimisée proposant une couche fine entre la fonction MPI en C et python n'ayant aucun surcoût par rapport à la fonction en C. Sous cette forme, on n'enverra que des tableaux numpy. Par exemple : `com.Bsend([buffer,MPI.DOUBLE], dest=1, tag=101)`
- Seconde forme : la fonction ou la méthode est en minuscule. Dans cette forme, on peut gérer tout objet python sérialisable, c'est à dire qui peuvent être compressés avec pickle. Par exemple : `com.bsend(obj, dest=1, tag=101)`. Ces fonctions sont plus faciles à manipuler que les fonctions de la première forme, mais ont un coup supplémentaire du fait de la compression et de la compression des objets via Pickle.

1.4 Communicateurs et groupe

MPI utilise des objets appelés communicateurs et des groupes qui définissent un ensemble de processus qui devront communiquer (échanger des messages) entre eux. La majorité des fonctions MPI demandent un communicateur en argument.

À l'initialisation de MPI, un communicateur est prédéfini : MPI_COMM_WORLD. Il inclut tous les processus créés par MPI.

Il est possible de définir un nouveau communicateur à partir d'un communicateur existant en prenant un sous–ensemble de processus parmi les processus du communicateur existant, mais cela dépasse le cadre de ce cours.

2 Fonctions de gestion de l'environnement

Ces fonctions sont utilisées pour interroger et définir l'environnement d'exécution MPI, et couvrent divers buts comme l'initialisation et la terminaison de l'environnement MPI, l'interrogation de l'identifiant du processus ou la version de MPI utilisée, etc.

2.1 Les fonctions de gestion de l'environnement

Les routines et valeurs les plus communes sont :

`com.size` Retourne le nombre total de processus MPI dans le communicateur spécifié. Si le communicateur `com` est `MPI.COMM_WORLD`, il représente alors le nombre des tâches MPI disponibles pour votre application.

`com.rank` Retourne le rang du processus MPI appelant dans le communicateur spécifié. Initialement, chaque processus se fera assigner un entier unique nommé *rang* entre 0 et le nombre de tâches - 1 dans le communicateur `MPI.COMM_WORLD`. Ce rang est souvent référé comme l'identité de la tâche. Si un processus devient associé avec d'autres communicateurs, il aura un rang unique (différent de celui attribué dans `MPI.COMM_WORLD`!) pour chacun de ces communicateurs.

`com.Abort(errocode : int = 0)` Termine tous les processus MPI associés avec le communicateur `com`. Dans la majorité des mises en œuvre de MPI, TOUS les processus sont arrêtés sans considération pour le communicateur.

`MPI.Wtime()` Retourne le temps réel passé en seconde (double précision) sur le processeur appelant.

`MPI.Wtick()` Retourne la résolution en seconde (double précision) de `MPI.Wtime`.

2.2 Exemple : Utilisation des fonctions de gestion de l'environnement

```
from mpi4py import MPI

# Lire le nombre de tâches :
numtasks = MPI.COMM_WORLD.size
# Lire le rang de la tâche courante :
rank     = MPI.COMM_WORLD.rank

# Ici on peut commencer à faire de la programmation
# parallèle par échange de messages.
```

2.3 Exécution d'un programme MPI

Pour exécuter en parallèle sur n tâches un programme MPI, on utilise l'exécutable `mpiexec` fourni en standard avec MPI. Son utilisation est la suivante :
`mpiexec [option] <`**program**`> [ <args> ]`
où

- <**program**> est le nom de l'exécutable à lancer. Il est identifié comme le premier argument non reconnu par MPI.
- `[ <args> ]` : Passe les arguments <args> à tous les processus créés par MPI. Ils devront toujours être les derniers arguments de mpiexec.
- Les options que l'on peut passer à mpiexec sont les suivantes :

– `-np` <n> : Exécute <n> copies du programme dans autant de processus. Cette option ne peut être utilisée que dans un modèle de programmation SPMD ;
– `-hostfile` <hostfile> : Utilise le fichier <hostfile> pour connaître le nom des machines (ou des nœuds de calcul) sur lesquels on lancera des tâches MPI

Le format du fichier <hostfile> est le suivant :

```
aa slots=a1
bb slots=b1
cc slots=c1
```

où `aa`, `bb` et `cc` sont des noms de machines, et `a1`, `a2` et `a3` sont des entiers indiquant à MPI le nombre de processus qu'on lui conseille de lancer pour chaque machine.

Dans notre cas, l'exécutable sera python lui-même, et en argument on aura le script que l'on veut lancer, soit :

```
mpiexec -np <nombre procs> python3 <script python>
```

2.4 Exercices

Un hello World parallèle Écrire un programme qui, pour chaque processus, affichera à l'écran le message suivant :

```
Bonjour, je suis la tâche n° xx sur yy tâches.
```

où `xx` est le rang de la tâche et `yy` le nombre de tâches lancées par MPI.

3 Fonctions d'échanges de messages point à point

3.1 Les types d'opérations point à point

Les fonctions d'échanges de messages point à point permettent d'échanger des messages entre deux – et seulement deux – tâches. Une des tâches se charge de l'envoi de données tandis que l'autre tâche se charge de la réception correspondante.

Il y a différents types d'envoi ou de réception, utilisés pour différents buts. Par exemple :

- les envois synchrones ;
- les envois/réceptions bloquants ;
- les envois/réceptions non bloquants ;
- les envois mis en buffer ;
- les envois/réceptions combinés ;
- les envois "prêts".

À noter que tout type d'envoi peut être reçu avec tout type de réception.

MPI fournit en outre plusieurs routines associées avec les envois et les réceptions telle que celles utilisées pour tester si un message a bien été reçu, ou si un message est prêt à la réception.

Les données sont transmises via un message, lequel est constitué :

- À l'envoi :
  - du numéro de la tâche destinataire
  - d'un numéro permettant d'identifier le message (à la réception)
  - des données à envoyer
  - du type de données à envoyer dans le cas de la première forme de fonction (optionnel)
- À la réception :

– du numéro de la tâche expéditrice (peut être `MPI.ANY_SOURCE` pour recevoir le premier message arrivé provenant de n'importe quel tâche),

– du numéro identifiant le message attendu (mais on peut très bien passer `MPI.ANY_TAG` pour recevoir le premier message arrivé provenant d'une tâche spécifiée ou non),

– de la variable recevant les données (en argument dans la seconde forme, en retour de la fonction dans la première forme)

– du type de données attendues pour la première forme (optionnel)

Le type des données est passé à l'aide d'une structure MPI, `MPI_Datatype`, qui peut prendre les valeurs données dans le tableau suivant (avec le type C/numpy correspondant) :

Type MPI	Type C
MPI.INT	int
MPI.SHORT	short
MPI.LONG	long
MPI.LONG_LONG_INT	long long
MPI.UNSIGNED	unsigned
MPI.UNSIGNED_LONG	unsigned long
MPI.UNSIGNED_SHORT	unsigned short
MPI.FLOAT	float
MPI.DOUBLE	double
MPI.LONG_DOUBLE	long double
MPI.CHAR	char
MPI.UNSIGNED_CHAR	unsigned char
MPI.C_COMPLEX	float_Complex
MPI.C_DOUBLE_COMPLEX	double_Complex
MPI.BYTE	void*
MPI.PACKED	données hétérogènes

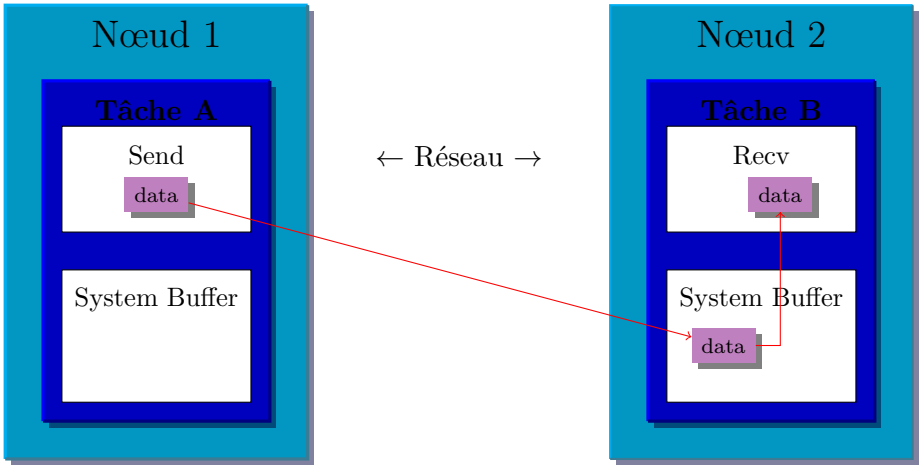
3.2 Buffering

Dans un monde parfait, les envois devraient être parfaitement synchronisés avec les réceptions correspondantes, mais en réalité c'est rarement le cas. D'une certaine manière, la bibliothèque MPI utilisée doit pouvoir gérer les données stockées lorsque les deux tâches sont non synchrones.

Par exemple :

- Une opération d'envoi a lieu cinq secondes avant que la réception soit prête – où se trouve le message avant que la réception soit faite ?
- Plusieurs envois arrivent simultanément sur la même tâche qui ne peut accepter qu'une réception à la fois – que deviennent les messages en attente de réception ?

La mise en œuvre de la bibliothèque MPI utilisée décide de la gestion de ces données dans tous les cas qui se présentent (cette gestion ne fait pas partie du standard !). Typiquement, une zone **buffer système** est réservée pour contenir les données en transit.



Cette zone privée est :

- opaque au programmeur et entièrement gérée par la bibliothèque MPI employée ;
- une ressource finie qui peut être facilement épuisée ;
- souvent mystérieuse et mal documentée ;
- capable d'exister du côté de l'envoi, de la réception ou des deux à la fois ;
- quelque chose qui devrait améliorer les performances du programme car permettant l'envoi et la réception en asynchrone.

L'espace d'adresse des variables du programme est appelée le **buffer d'application**. MPI permet à l'utilisateur de gérer son buffer d'envoi dans cet espace.

3.3 Bloquant contre non bloquant

La plupart des routines point à point de MPI peuvent être utilisées en communication bloquante ou non bloquante.

Bloquant
- Un envoi bloquant rendra la main seulement après s'être assuré que modifier le buffer d'application est "sûr". "Sûr" signifie ici que ces modifications n'altéreront pas les données qui devront être reçues par la tâche destinataire – elles peuvent par contre être stockées dans un buffer système.
- Un envoi bloquant peut être synchrone, ce qui signifie qu'il y a un échange d'information avec la tâche destinataire pour confirmer que l'envoi est sûr ;
- Un envoi bloquant peut être asynchrone si un buffer système est utilisé pour contenir les données à envoyer au destinataire ;
- Une réception bloquante rend la main seulement après que les données soient arrivées et prêtes à être utilisées par l'utilisateur.

Non bloquant
- Les envois et réceptions non bloquants se comportent de manière similaire – ils rendent la main immédiatement. Ils n'attendent pas qu'une étape de communication soit finie, telle que la copie du buffer d'application vers le buffer système ou l'arrivée du message attendu ;
- Les opérations non bloquantes interrogent simplement la bibliothèque MPI pour exécuter les opérations quand elles sont valides. L'utilisateur ne peut pas deviner le moment où cela va se passer ;
- Il est dangereux de modifier le buffer d'application tant que vous n'êtes pas certain que l'opération non bloquante demandée se soit exécutée. Il existe des routines de type "attente" pour s'en assurer.
- Les communications non bloquantes sont utilisées surtout pour recouvrir les communications avec des calculs et pouvoir ainsi gagner en rapidité d'exécution.

Envoi bloquant

```
myvar = 0
for i in range(1,numtasks):
  task = i
  MPI.COMM_WORLD.bsend(myvar, dest=
    task)
  myvar = myvar + 2
  # Do some works
  ...
  ...
```

Sûr. Pourquoi ?

Envoi non bloquant

```
myvar = 0
for i in range(1,numtasks):
  task = i
  req = MPI.isend(myvar,dest=task)
  myvar = myvar + 2
  # Do some works
  ...
  req.wait ()
```

Dangereux. Pourquoi ?

3.4 Routines d'échange de message MPI

Les communications points à points MPI prennent en général une liste d'argument similaire.

Pour les fonctions de la première forme :

Envoi bloquant	`com.Send( buffer, dest, t`
Envoi non bloquant	`reques = com.Isend( buffer, de`
Réception bloquante	`com.Recv( buffer, source, tag, status`
Réception non bloquante	`request = com.IRecv(buffer, sou`

où

com Le communicateur pour lequel les numéros des tâches utilisées sont valides. Sauf si le programmeur a créé des communicateurs, on utilise par défaut `MPI.COMM_WORLD`.

buffer est soit un tableau numpy sans autre spécifications, soit une liste `[tableau,type]` donnant le tableau numpy et le type de données contenues

dest Un argument pour les routines d'envoi spécifiant le numéro de la tâche destinée à recevoir le message ;

source Spécifie le numéro de la tâche dont on attend le message. Ce numéro peut être un "joker" et on peut attendre un message provenant de n'importe quelle tâche grâce à la variable `MPI.ANY_SOURCE`.

tag Un entier arbitraire donné par le programmeur pour identifier son message. Ce nombre peut être compris entre 0 et 32767. En réception, la routine attend un message portant cet identifiant pour le recevoir. On peut là encore utiliser un "joker" à l'aide de la variable `MPI.ANY_TAG` ;

status Pour une opération de réception, indique la source et l'identifiant du message. On doit donc définir un objet de type Status : `status = MPI.Status()`

request Pour les opérations non bloquantes, c'est une structure opaque servant à analyser l'état de l'envoi ou de la réception. Cette variable sera utilisée ensuite pour synchroniser la réception ou l'envoi (à l'aide d'une méthode de type `request.Wait`).

Fonctions d'échange de message bloquantes

- `com.send` :

  C'est l'opération basique d'envoi bloquant avec sérialisation et compression de l'objet python envoyé. La fonction ne rend la main que lorsque le buffer d'application dans la tâche est de nouveau libre pour être reutilisé.

  ```
  com.Send (obj,dest,tag)
  ```

  Exemples :

  ```
  obj = ["Valeurs", 3, {'pub' : 313_201, 'priv' : 234_321}, 3.1416]
  MPI.COMM_WORLD.send(obj, dest=rank+1)
  ```

- `com.Send` :

  C'est l'opération basique d'envoi bloquant optimisé d'un tableau numpy. La fonction ne rend la main que lorsque le buffer d'application dans la tâche est de nouveau libre pour être reutilisé.

  ```
  com.Send (buffer,dest,tag)
  ```

  Exemples :

  ```
  buffer = np.array([1,3,5,7], dtype=np.int)
  MPI.COMM_WORLD.Send(buffer, dest=rank+1)
  ```

- `com.recv`

  Réception basique d'un message recevant un objet python sérialisé. Ne rend pas la main jusqu'à ce que le buffer d'application soit utilisable par l'utilisateur.

  ```
  obj = com.recv(source, tag, status)
  ```

  Exemples :

```
status = MPI.Status()
obj = com.recv(source=rank-1, status=status)
print(f"Titre : {obj[0]}, clef publique : {obj[2]['pub']}")
```

- com.Recv

  Réception d'un message recevant un buffer numpy. Ne rend pas la main jusqu'à ce que le buffer d'application soit utilisable par l'utilisateur.

  ```
  com.Recv(buffer, source, tag, status)
  ```

  Exemples :

  ```
  status = MPI.Status()
  buffer = np.empty(4, dtype=np.int)
  com.Recv(buffer, source=rank-1, status=status)
  print(f"buffer : {buffer}")
  ```

- com.ssend

  Envoi bloquant synchrone avec sérialisation objet python. Envoie un message et attend que le buffer d'application dans la tâche émettrice puisse de nouveau être reutilisé et que la tâche destinataire ait commencé à recevoir le message.

  ```
  com.ssend(obj, dest, tag)
  ```

  Exemples :

  ```
  obj = ["Valeurs", 3, {'pub' : 313_201, 'priv' : 234_321}, 3.1416]
  com.ssend(obj, dest=rank+1, tag=101)
  ```

- com.Ssend

  Envoi bloquant synchrone optimisé pour tableau numpy. Envoie un message et attend que le buffer d'application dans la tâche émettrice puisse de nouveau être reutilisé et que la tâche destinataire ait commencé à recevoir le message.

  ```
  com.Ssend(buffer, dest, tag)
  ```

  Exemples :

  ```
  buffer = np.array([1,3,5,7], dtype=np.int)
  com.Ssend(buffer, dest=rank+1, tag=101)
  ```

- com.sendrecv

  Envoie un message sérialisant des objets pythons puis attend une réception avant de bloquer. Bloque jusqu'à ce que le buffer d'application d'envoi soit de nouveau disponible pour utilisation et que le buffer d'application de réception contienne le message reçu.

  ```
  recvObj = com.sendrecv( sendObj, dest, sendtag, source, recvtag, status );
  ```

  Exemples :

  ```
  obj = ["Valeurs", 3, {'pub' : 313_201, 'priv' : 234_321}, 3.1416]
  status = MPI.Status()
  obj2 = com.sendrecv( obj, dest=rank+1, sendtag=101, source = rank-1, recvtag
      =101,status=status )
  ```

- com.Sendrecv

  Envoie un message optimisé pour les buffers numpy puis attend une réception avant de bloquer. Bloque jusqu'à ce que le buffer d'application d'envoi soit de nouveau disponible pour utilisation et que le buffer d'application de réception contienne le message reçu.

  ```
  com.Sendrecv( sendBuffer, dest, sendtag, recvBuffer, source, recvtag, status );
  ```

  Exemples :

```
buffer = np.array([1,2,3,4], dtype=np.int)
buffer2 = np.empty(4, dtype=np.int)
status = MPI.Status()
com.Sendrecv( buffer, dest=rank+1, sendtag=101, buffer2, source=rank-1, recvtag
    =101, status=status )
```

- `com.Probe`

  Effectue un test bloquant sur un message. Les "jokers" `MPI.ANY_SOURCE` et `MPI.ANY_TAG` peuvent être utilisés pour tester un message provenant de n'importe quelle tâche ou identifié par n'importe quel nombre. À la sortie de la fonction, le numéro de l'expéditeur et le numéro identifiant le message seront retournés dans le paramètre `status` (à l'aide de `status.Get_source()` et `status.Get_tag()`).

  ```
  com.Probe( source, tagn status )
  ```

- `status.Get_count()`

  Retourne le nombre d'éléments devant ou ayant été reçu d'un message MPI en réception.

  ```
  status.Get_count()
  ```

Exemple d'utilisation des fonctions bloquantes d'échange de messages

```
from mpi4py import MPI

# On recopie le communicateur global dans un nouveau communicateur
globCom  = MPI.COMM_WORLD.Dup()
numtasks = globCom.size
rank     = globCom.rank

inMsg = None
status = MPI.Status()
# La tâche 0 envoie une valeur à la tâche 1 et attend de recevoir un message en
    retour
if 0 == rank :
  outMsg = 'x'
  globCom.send(outMsg, dest=1)
  inMsg = globCom.recv(source=1, status=status)
else:
  outMsg = 'y'
  inMsg = globCom.recv(source=0, status=status)
  globCom.send(outMsg, dest=0)

print(f"Tâche {rank} : Reçu {status.Get_count()} caractères de la tâche {status.
    Get_source()} avec tag  {status.Get_tag()}")
print(f"Tâche {rank} : La lettre reçue est {inMsg}")
```

Fonctions non bloquantes d'échange de message Les fonctions non bloquantes les plus utilisées pour échanger des messages sont données ci–dessous. On y donne également les fonctions bloquantes qui leur sont associées.

- `com.isend`

  Envoie non bloquant avec sérialisation d'objet python.

  Utilise une aire en mémoire pour servir comme buffer d'envoi. La tâche continue immédiatement après l'appel sans attendre que le message soit copié du buffer d'application. Une requête est renvoyée permettant au programmeur de connaître l'état de l'envoi. Le programmeur doit s'assurer de ne pas modifier le buffer d'application jusqu'à ce qu'il ait testé si le buffer d'application est de nouveau prêt à l'emploi grâce aux fonctions `request.Wait` ou `request.Test`.

  ```
  request = com.isend( obj, dest, tag )
  ```

- com.Isend

  Envoie non bloquant optimisé pour tableaux numpy.

  Utilise une aire en mémoire pour servir comme buffer d'envoi. La tâche continue immédiatement après l'appel sans attendre que le message soit copié du buffer d'application. Une requête est renvoyée permettant au programmeur de connaître l'état de l'envoi. Le programmeur doit s'assurer de ne pas modifier le buffer d'application jusqu'à ce qu'il ait testé si le buffer d'application est de nouveau prêt à l'emploi grâce aux fonctions `request.Wait` ou `request.Test`.

  ```
  request = com.Isend( buffer, dest, tag )
  ```

- com.irecv

  Réception non bloquante avec sérialisation d'objet python.

  Utilise une aire en mémoire pour servir de buffer de réception. La tâche continue immédiatement après l'appel à la fonction sans attendre que le message soit copié dans le buffer d'application. Une requête est renvoyée permettant au programmeur de connaître l'état de la réception. Le programmeur doit utiliser une des fonctions `request.Wait` ou `request.Test` pour s'assurer que le message reçu est valable dans le buffer d'application.

  La réception de l'objet Python se fera au moment du wait ou du test :

  ```
  request = com.irecv( source, tag )
  ```

- com.Irecv

  Réception non bloquante optimisé pour tableaux numpy.

  Utilise une aire en mémoire pour servir de buffer de réception. La tâche continue immédiatement après l'appel à la fonction sans attendre que le message soit copié dans le buffer d'application. Une requête est renvoyée permettant au programmeur de connaître l'état de la réception. Le programmeur doit utiliser une des fonctions `request.Wait` ou `request.Test` pour s'assurer que le message reçu est valable dans le buffer d'application.

  ```
  request = com.Irecv( buffer, source, tag )
  ```

- com.issend et com.Issend

  Envoi synchrone non bloquant. Cette fonction est similaire à `com.isend` sauf que `request.Wait` ou `request.Test` indiquent si la tâche destinataire a reçu ou non le message.

  Premère forme pour version numpy optimisée :

  ```
  request = com.Issend( buffer, dest, tag )
  ```

  Seconde forme pour objet python sérialisés :

  ```
  request = com.issend( obj, dest, tag )
  ```

- request.Test, request.Testany, request.Testall, request.Testsome

  Ces fonctions testent le status d'un envoi ou d'une réception non bloquante spécifique à l'aide de la requête associée. Le paramètre "`flag`" est retourné comme logiquement vrai (=1) si l'opération est finie et logiquement fausse (=0) sinon. Pour plusieurs opérations non bloquantes, le programmeur peut demander à tester si une des opérations, toutes les opérations ou certaines opérations sont finies.

  Dans la seconde forme, retourne en plus (dans un tuple) l'objet déssérialisé reçu par la réception non bloquante.

  Première forme optimisée pour tableaux numpy :

  ```
  # Retourne vrai si message associé à request est reçu
  flag = request.Test(status=None)
  # Retourne vrai si un des messages reçus par les requests est reçu ainsi que l'
      indice du message reçu
  flag,index = MPI.Request.Testany( [request1,request2,...,requestn], status=None
      )
  # Retourne vrai si tous les messages portés par les requests sont reçus
  ```

```
flag = MPI.Request.Testall( [request1,request2,...,requestn], [status1,status2
    ,...,statusn] );
# Test si certains messages ont été reçus dans les requests. Retourne les
    indices des requests dans la liste associée aux messages.
listIndices = MPI.Request.Testsome( [request1,request2,...,requestn], , [
    status1,status2,...,statusn] )
```

Seconde forme optimisée pour objets pythons :

```
# Retourne vrai et l'objet reçu si message reçu, sinon renvoie faux et None
flag,obj = request.test(status=None)
# Retourne vrai et l'objet reçu si un des messages reçus par les requests est
    reçu ainsi que l'indice du message reçu
flag,index,obj = MPI.Request.testany( [request1,request2,...,requestn], status=
    None )
# Retourne vrai et les objets reçus si tous les messages portés par les
    requests sont reçus
flag,listObjs = MPI.Request.testall( [request1,request2,...,requestn], [status1
    ,status2,...,statusn] );
# Test si certains messages ont été reçus dans les requests. Retourne les
    indices des requests ansi que les objets reçus dans la liste associée aux
    messages.
listIndices, listObjs = MPI.Request.testsome( [request1,request2,...,requestn],
     , [status1,status2,...,statusn] )
```

- request.Wait, request.Waitany, request.Waitall, request.Waitsome

  Ces fonctions bloquent jusqu'à ce qu'un envoi ou une réception spécifié soit fini. Pour plusieurs envois/réceptions non bloquants, le programmeur peut spécifier si une des opérations, toutes ou certaines opérations sont finies.

  Première forme optimisée pour tableaux numpy :

```
# Attend que le message associé à request soit reçu
request.Wait(status=None)
# Attend que un des messages attendus soit reçu et renvoie l'index du request
    associé dans la liste
index = MPI.Request.Waitany( [request1,request2,...,requestn], status=None )
# Attend que tous les messages portés par les request sont reçus
MPI.Request.Waitall( [request1,request2,...,requestn], [status1,status2,...,
    statusn] = None)
# Attend que certains messages portés les requests sont reçus
MPI.Request.Waitsome( [request1,request2,...,requestn], [status1,status2,...,
    statusn] = None )
```

  Seconde forme pour objets python sérialisés :

```
# Attend que le message associé à request soit reçu et renvoit l'objet reçu
obj = request.wait(status=None)
# Attend que un des messages attendus soit reçu et renvoie l'index du request
    associé dans la liste ainsi que l'objet reçu associé
index, obj = MPI.Request.waitany( [request1,request2,...,requestn], status=None
    )
# Attend que tous les messages portés par les request sont reçus et renvoie les
     objets reçus
listObjs = MPI.Request.waitall( [request1,request2,...,requestn], [status1,
    status2,...,statusn] = None)
# Attend que certains messages portés les requests sont reçus et renvoie les
    objets associés
listObjs =  MPI.Request.waitsome( [request1,request2,...,requestn], [status1,
    status2,...,statusn] = None )
```

- com.Iprobe

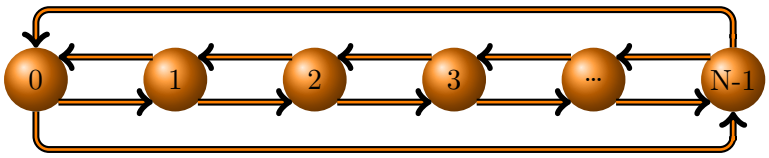
  Effectue un test non bloquant pour un message. Les "jokers" `MPI.ANY_SOURCE` et `MPI.ANY_TAG` peuvent être utilisés pour tester un message provenant de n'importe quelle autre tâche ou ayant n'importe quel identifiant. Un blléen est retourné avec une valeur logiquement vraie si le message

est arrivé et logiquement fausse sinon. Le numéro de l'expéditeur et l'identifiant du message sont retournés via `status` comme `status.Get_source()` et `status.Get_tag()`.

```
flag = com.Iprobe( source, tag, status = None )
```

Exemple de programme utilisant des messages non bloquants

Échange de données avec son plus proche voisin dans une topologie en anneau :




```
import numpy as np
from mpi4py import MPI

globCom = MPI.COMM_WORLD.Dup()
numtasks = globCom.size
rank     = globCom.rank

# Détermine les processus avec un numéro "voisin"
prev = rank-1 if rank > 0 else numtasks-1
next = (rank+1)%numtasks

buffer0 = np.array([rank,2*rank+1,3*rank+5, 5*rank+7], dtype=np.double)
buffer1 = np.array([rank,2*rank-1,3*rank-5, 5*rank-7], dtype=np.double)
buffer2 = np.empty(4, dtype=np.double)
buffer3 = np.empty(4, dtype=np.double)

tag1 = 101
tag2 = 102
# Poste des réceptions non bloquantes et envoie des données aux voisins
req0 = globCom.Irecv(buffer2, source=prev, tag=tag1)
req1 = globCom.Irecv(buffer3, source=next, tag=tag2)

req2 = globCom.Isend(buffer0, dest=prev, tag=tag2)
req3 = globCom.Isend(buffer1, dest=next, tag=tag1)

# EFfectuer plusieurs calculs indépendants des données qu'on va recevoir tandis
    ques les envoies et les réceptions se font en arrière fond.

# wait for all non-blocking operations to complete
lstats = [ MPI.Status(), MPI.Status(), MPI.Status(), MPI.Status() ]
MPI.Request.Waitall([req0,req1,req2,req3], lstats);

# Continuer à effectuer d'autres calculs dépendant de ce qu'on a reçu
```

4 Les fonctions de communication collectives

Il existe trois sortes de fonctions de communication collectives :

1. **Synchronisation** : Les tâches attendent que les autres tâches du groupe aient atteint le point de synchronisation ;
2. **Échanges de données** : Diffusion, regroupement, partition, ...
3. **Réduction** : Calcul collectif : une tâche du groupe collecte toutes les données des autres membres et effectue une opération sur ces données (min, max, opération arithmétique, etc.).

Les communications collectives mettent en œuvre toutes les tâches appartenant à un même groupe de communication

- Toutes les tâches lancées par l'utilisateur en utilisant `MPI.COMM_WORLD` ;

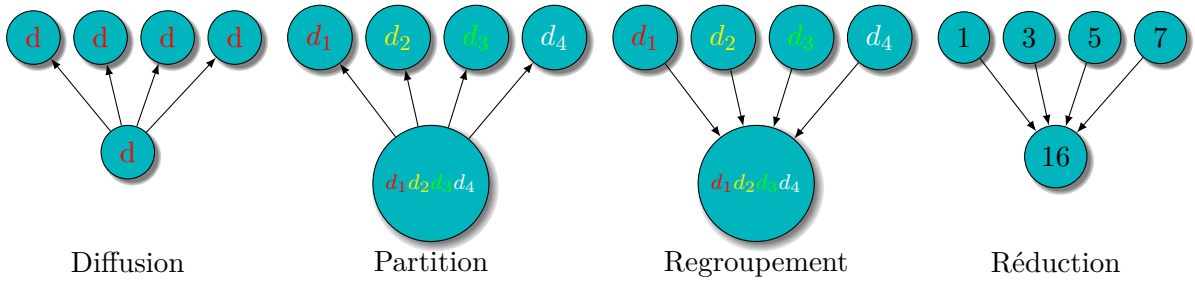


FIG. 1 : Les différents échanges de messages collectifs

- Un sous-ensemble de tâches si on a défini un nouveau communicateur ne prenant en compte que ce sous-ensemble

Des comportements aléatoires peuvent avoir lieu, dont un plantage du code, si même une seule tâche du communicateur n'appelle pas la fonction.

C'est de la responsabilité du programmeur de s'assurer que toutes les tâches du communicateur utilisé appellent la fonction de communication collective.

Les communications collectives ne prennent pas d'identificateur en arguments.

Si on veut exécuter une communication collective sur un sous-ensemble de tâches, il faut tout d'abord partitionner un groupe de tâches en sous-ensembles puis associer ces groupes à des communicateurs (voir

À partir de la norme MPI 3 on peut choisir entre des communications collectives bloquantes ou non bloquantes.

Fonctions de communications collectives

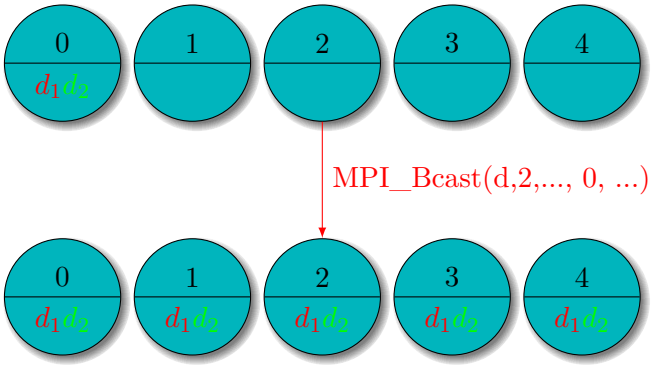
- `com.Barrier`

  Opération de synchronisation. Crée une barrière de synchronisation dans un groupe. Chaque tâche, lorsqu'elle atteint l'appel au `MPI_Barrier`, bloque jusqu'à ce que toutes les tâches du groupe atteignent le même appel à `com.Barrier`. Alors toutes les tâches peuvent continuer à travailler.

  ```
  com.Barrier()
  ```

- `com.Bcast`

  Opération de mouvement de données. Diffuse (envoie) un message de la tâche ayant le rang "root" à toutes les autres tâches du groupe.



  Sous la première forme optimisée pour les tableaux numpy :

  ```
  com.Bcast(buffer, root = 0)
  ```

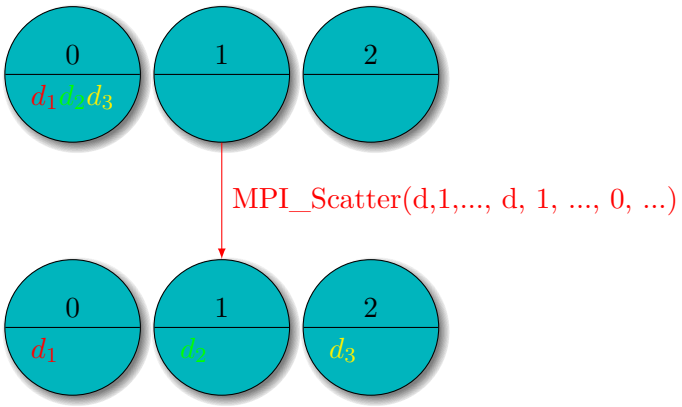
  Sous la seconde forme servant aux objets python sérialisables :

  ```
  objToReceive = com.bcast(objToBCast, root = 0)
  ```

  Note : POur la seconde forme, les processus qui ne sont pas root peuvent passer None comme objet à diffuser.

- com.Scatter

  Opération de mouvement de données. Distribue (en partitionnant) des données distinctes d'une seule tâche source "root" vers chaque tâche du groupe.



  Sous la première forme optimiséé pour les tableaux numpy :

```
com.Scatter(sendBuffer, recvBuffer, root=0)
```

  où `recvBuffer` peut être aussi égal à `MPI.IN_PLACE`

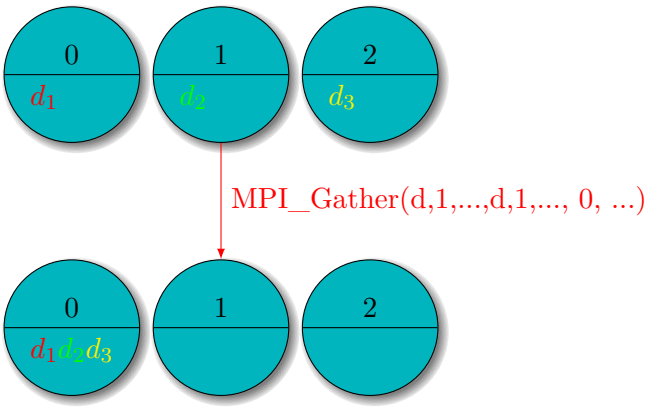
  Sous la seconde forme pour les objets pythons sérialisables :

```
scatObj =  com.scatter( sendobj : Sequence[Any], root : int = 0 )
```

  Là encore, l'objet partitionné peut être mis à None pour les processus non root.

- com.Gather

  Opération de mouvement de données. Rassemble différentes données provenant de chaque tâche du groupe dans une seule tâche destinataire "root". Cette fonction est l'opération inverse de `MPI_Scatter`.



  Sous la première forme optimisée pour les tableaux numpy :

```
com.Gather(sendBuf, recvBuf, root=0)
```

  où `sendBuffer` peut être aussi égal à `MPI.IN_PLACE`. Dans ce cas, recvBuf ne doit pas être passé en argument.

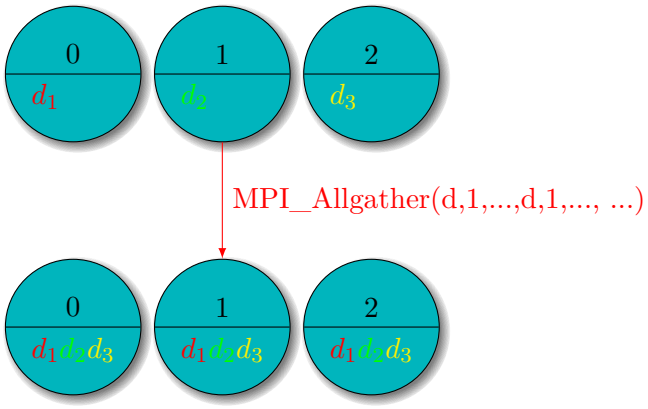
  Sous la seconde forme pour les objets pythons sérialisables :

```
listObjs = com.gather(sendObj, root=0)
```

  qui concatène chaque objet passé par un processus dans une liste (donc un peu différent du fonctionnement normal)

- com.Allgather

  Opération de mouvement de données. Concaténation des données sur toutes les tâches dans un groupe. Chaque tâche dans le groupe, en effet, effectue une diffusion une tâche vers toutes dans le groupe.



Sous la première forme optimisée pour les tableaux numpy :

```
com.Allgather(sendBuf, recvBuf)
```

où `recvBuf` peut être aussi égal à `MPI.IN_PLACE`.

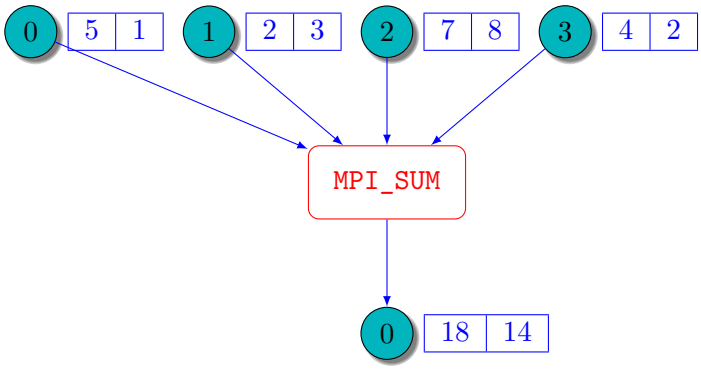
Sous la seconde forme pour les objets pythons sérialisables :

```
listObjs = com.allgather(sendObj)
```

qui concatène chaque objet passé par un processus dans une liste pour tous les processus(donc un peu différent du fonctionnement normal)

- `com.Reduce`

  Opération de calcul collectif. Applique une opération de réduction sur toutes les tâches du groupe et place le résultat dans une tâche.



Sous la première forme optimisée pour les tableaux numpy :

```
com.Reduce(sendBuf, recvBuf, op = MPI.SUM, root = 0)
```

où `sendBuf` peut être aussi égal à `MPI.IN_PLACE`. Dans ce cas, il ne faut pas passer l'argument `recvBuf`.

Sous la seconde forme pour les objets pythons sérialisables :

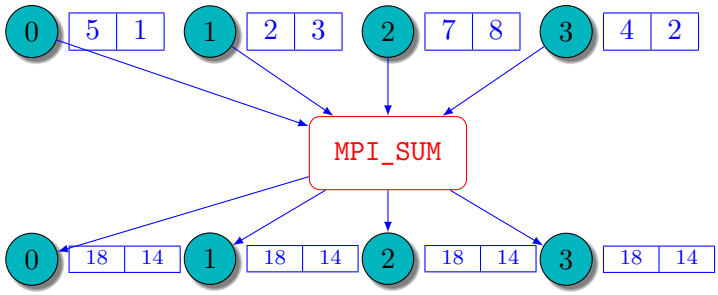
```
obj = com.reduce(sendObj, op = MPI.SUM, root = 0)
```

Dans sa seconde forme, op peut être un objet appelable (une fonction ou autre), ou comme pour la première forme, une opération prédéfinie par MPI comme donnée dans la table ci-dessous :

Opération de réduction MPI		Type de données C
`MPI.MAX`	Maximum	Entiers, Réels
`MPI.MIN`	Minimum	Entiers, Réels
`MPI.SUM`	Somme	Entiers, Réels
`MPI.PROD`	Produit	Entiers, Réels
`MPI.LAND`	ET logique	Entiers
`MPI.BAND`	ET par bit	Entiers, `MPI_BYTE`
`MPI.LOR`	OU logique	Entiers
`MPI.BOR`	OU par bit	Entiers, `MPI_BYTE`
`MPI.LXOR`	OU exclusif logique	Entiers
`MPI.BXOR`	OU exclusif bit à bit	Entiers, `MPI_BYTE`
`MPI.MAXLOC`	Valeur maximale et location	float, double et long double
`MPI.MINLOC`	Valeur minimale et location	float, double et long double

- `com.Allreduce`

  Opération de calcul collectif et de déplacement de données. Applique une opération de réduction et diffuse le résultat dans toutes les tâches du groupe. Cette fonction est équivalente à appeler une fonction de réduction suivi d'un appel à `MPI_Bcast`.



  Sous la première forme optimisée pour les tableaux numpy :

```
com.Allreduce(sendBuf, recvBuf, op = MPI.SUM)
```

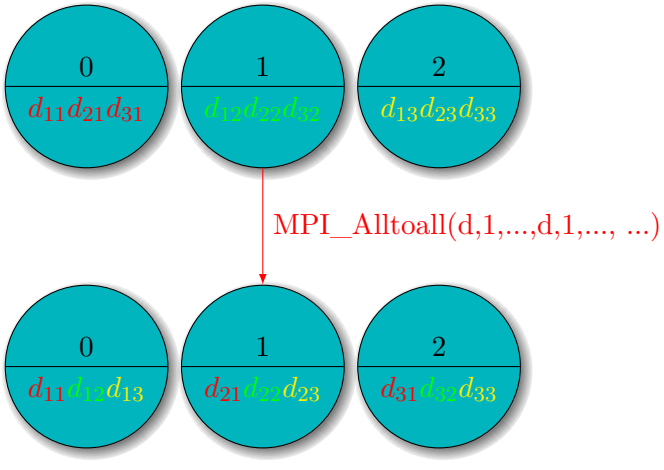
  où `sendBuf` peut être aussi égal à `MPI.IN_PLACE`. Dans ce cas, il ne faut pas passer l'argument `recvBuf`.

  Sous la seconde forme pour les objets pythons sérialisables :

```
obj = com.allreduce(sendObj, op = MPI.SUM)
```

- `com.Alltoall`

  Opération de mouvement de données collective. Chaque tâche dans le groupe effectue une opération de diffusion, envoyant un message distinct à toutes les tâches du groupe dans l'ordre par index.



  Sous la première forme optimisée pour les tableaux numpy :

```
com.Alltoall(sendBuf, recvBuf)
```

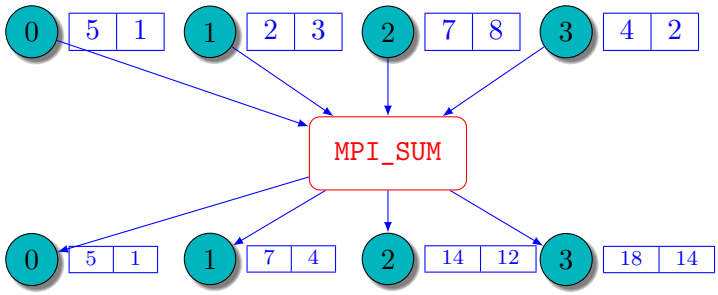
où `sendBuf` peut être aussi égal à `MPI.IN_PLACE`. Dans ce cas, il ne faut pas passer l'argument `recvBuf`.

Sous la seconde forme pour les objets pythons sérialisables :

```
Listobjs = com.all2all(sendObjs : Sequence[Any])
```

- `MPI_Scan`

  Effectue une opération de balayage par rapport à une opération de réduction au travers d'un groupe de tâches.



  Sous la première forme optimisée pour les tableaux numpy :

```
com.Scan(sendBuf, recvBuf, op = MPI.SUM)
```

  où `sendBuf` peut être aussi égal à `MPI.IN_PLACE`. Dans ce cas, il ne faut pas passer l'argument `recvBuf`.

  Sous la seconde forme pour les objets pythons sérialisables :

```
obj = com.scan(sendObj, op = MPI.SUM)
```

Exemple : Distribue les lignes d'un tableau

```
from mpi4py import MPI
import numpy as np

sendbuf = np.array([[1.,2.,3.,4.],
                    [5.,6.,7.,8.],
                    [9.,10.,11.,12.],
                    [13.,14.,15.,16.]], dtype=np.float)
recvBuf = np.empty(4, dtype=np.float)

globCom = MPI.COMM_WORLD.Dup()
rank    = globCom.rank
numtasks= globCom.size

if numtasks != SIZE :
  print("Programme ne marche qu'avec quatre processus")
  globCom.Abort()
else:
  source = 1
  globCom.Scatter(sendBuf, recvBuf, root=source)

print(f"rank {rank} : résultat {recvbuf}")
```

4.1 Fonctions de gestion de groupes et de communicateurs

Groupes et communicateurs

- Un groupe est un ensemble ordonné de tâches. On associe un rang unique à chaque tâche dans le groupe. La valeur du rang est comprise entre 0 et N-1 où N est le nombre de tâches dans le groupe. Dans MPI, un groupe est représenté par un objet en mémoire uniquement accessible par le programmeur à l'aide d'une "poignée". Un groupe est toujours associé avec un objet communicateur.

- Un communicateur englobe un groupe de tâches qui vont devoir communiquer entre elles. Tout message MPI doit spécifier un communicateur. Dans le sens le plus simple, le communicateur est un identifiant supplémentaire obligatoire pour les messages MPI. Comme les groupes, les communicateurs sont representés comme des objets uniquement accessibles par le programmeur à l'aide d'une "poignée". Par exemple, la poignée pour le communicateur contenant toutes les tâches est `MPI.COMM_WORLD`.
- Du point de vue du programmeur, un groupe et un communicateur sont identiques. Les fonctions de manipulation de groupes sont uniquement utilisées pour désigner les tâches servant à construire le communicateur.

Utilités primaires des groupes et des communicateurs

1. Vous permet d'organiser les tâches selon des fonctions en groupes de tâches ;
2. Permet d'utiliser les communications collectives sur un sous-ensemble de tâches ;
3. Permet des communications sécurisées.

Considération de programmation et restrictions

- Les groupes/communicateurs sont dynamiques : ils ne peuvent être créés ou détruits que pendant l'exécution du programme.
- Les tâches peuvent appartenir à plusieurs groupes/communicateurs. Elles auront un identifiant de rang unique pour chaque groupe/communicateur.
- Plus de quarante fonctions sont disponibles pour manipuler les groupes et les communicateurs.
- Programmation typique :
  1. Récupérer la poignée du groupe global associé à `MPI.COMM_WORLD` en faisant appel à la fonction `com.Get_group` ;
  2. Créer un nouveau groupe en tant que sous-ensemble du groupe global en appelant `group.Incl([liste de rangs])` ;
  3. Créer un nouveau communicateur pour le nouveau groupe en utilisant `com.Create(groupe)` ;
  4. Déterminer le rang de la tâche dans le nouveau communicateur en utilisant `com.rank` ;
  5. Utiliser le nouveau communicateur dans votre programme ;
  6. Quand le nouveau communicateur n'est plus utilisé, détruire le nouveau communicateur et éventuellement son groupe associé en appelant `MPI_Comm_free` et `MPI_Group_free`.
- Pour effectuer une partition des tâches au travers de plusieurs communicateurs, on utilisera plutôt `com.Split(couleur, clef)`.

Exemple : Créer deux communicateurs : un pour les tâches de rang pair dans le communicateur global et l'autre pour les tâches de rang impair.

```
from mpi4py import MPI

globCom  = MPI.COMM_WORLD.Dup()
rank     = globCom.rank
numtasks = globCom.size

new_comm = None
if rank%2 == 0:
    new_comm = globCom.Split(0, rank)
else:
    new_comm = globCom.Split(1, rank)

# Lire son rang dans le nouveau communicateur :
new_rank = new_comm.rank
print(f"Ancien rang : {rank}, nouveau rang : {new_rank}")
```