Modèles de communication

Modèles de communication

Modes d'envoi point à point

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

Modèles de communication

Appels bloquants

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

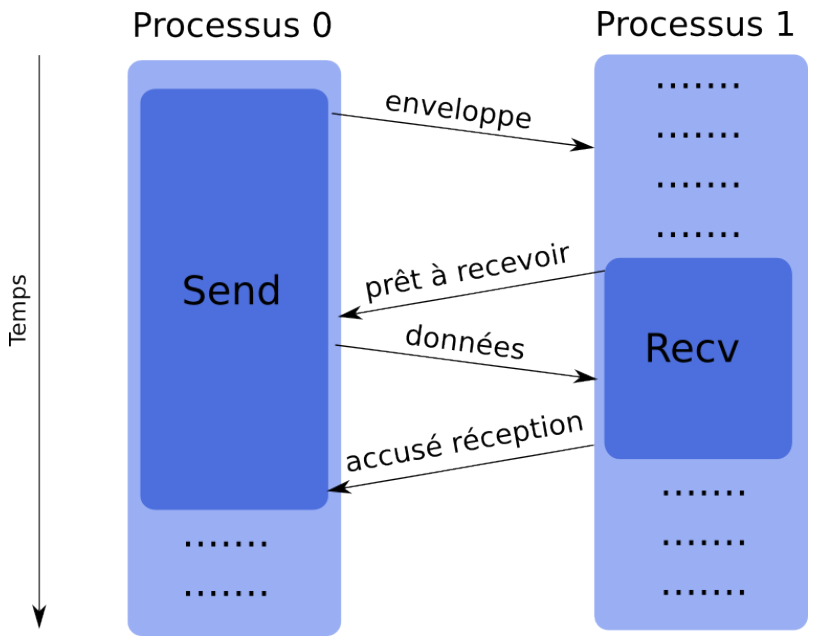
Modèles de communication

Envois synchrones

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

Protocole de *rendez-vous*

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



Modèles de communication

Interface de `MPI_Ssend()`

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

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

Avantages

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

Inconvénients

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

Exemple d'inter-blocage

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

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

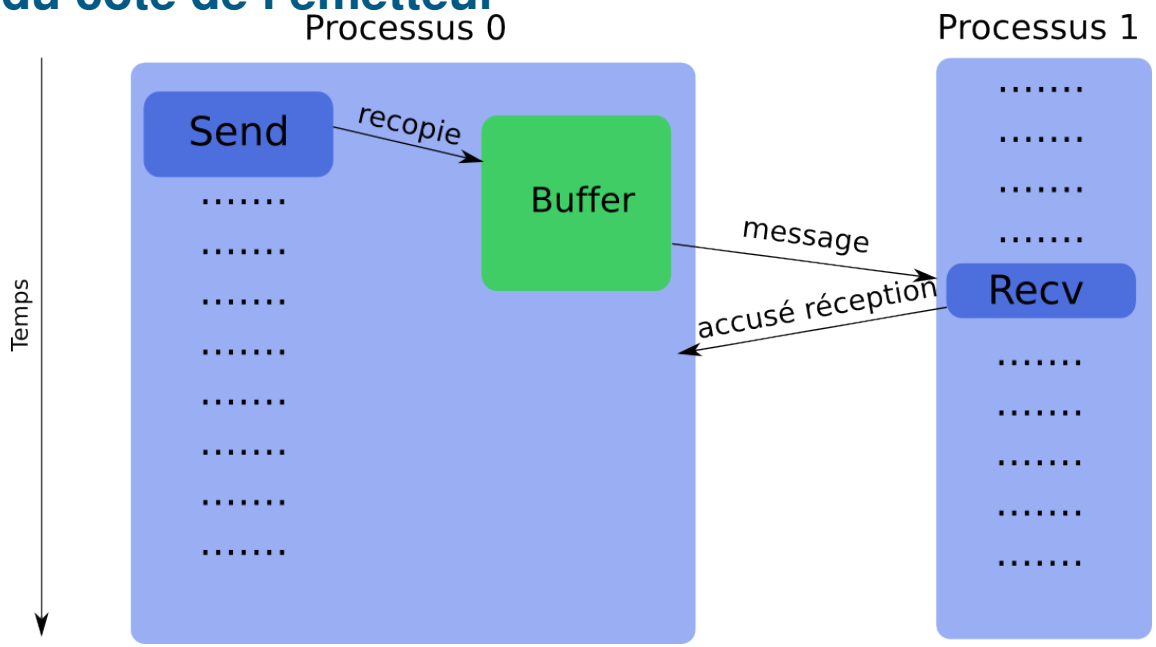
Modèles de communication

Envois *bufferisés*

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

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

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



Modèles de communication

Buffers

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

Interfaces

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

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

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

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

Modèles de communication

Avantages du mode bufferisé

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

Inconvénients du mode bufferisé

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

Modèles de communication

Absence d'inter-blocage

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

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

Modèles de communication

Envois standards

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

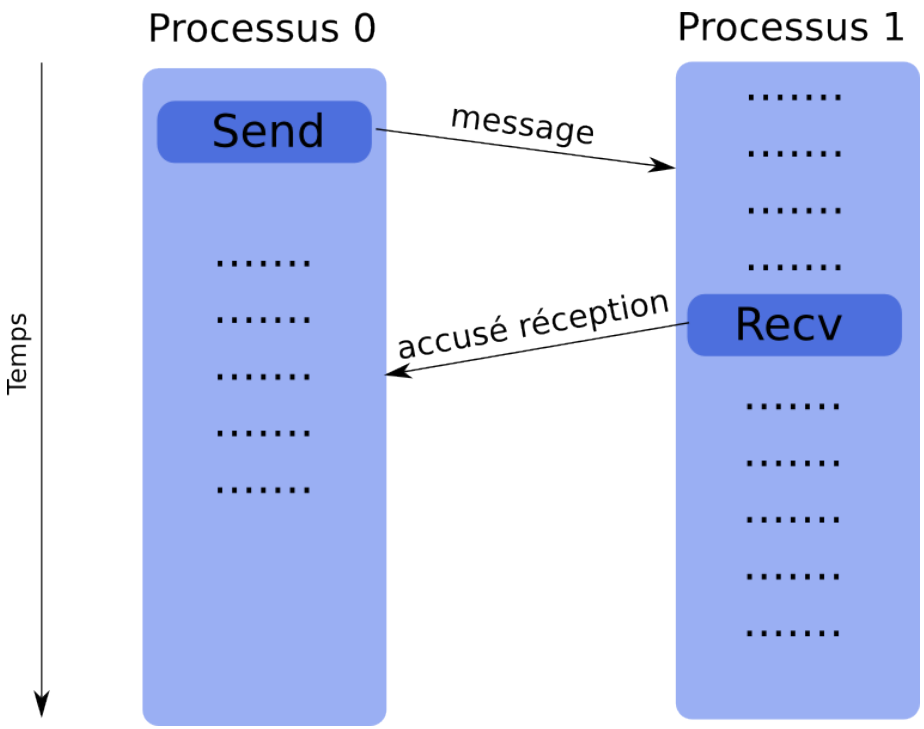
Interfaces

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

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

Modèles de communication

Protocole *eager*

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



Modèles de communication

Avantages du mode standard

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

Inconvénients du mode standard

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

Modèles de communication

Nombre d'éléments reçus

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

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

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

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

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

Modèles de communication

Nombre d'éléments reçus

MPI_Probe permet de vérifier les messages entrants sans les recevoir.

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

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

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

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

Modèles de communication

Présentation

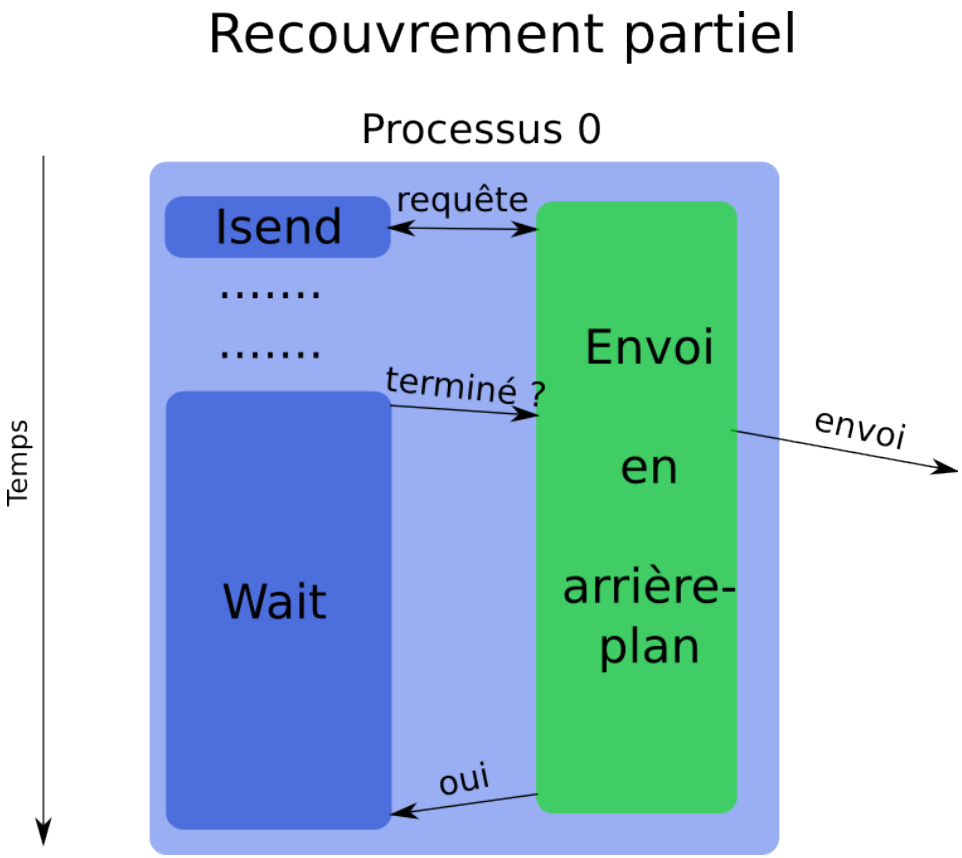
Le recouvrement des communications par des calculs est une méthode permettant de réaliser des opérations de communications en arrière-plan pendant que le programme continue de s'exécuter. Sur Jean Zay, la latence d'une communication inter-nœud est de 1μs soit 2500 cycles processeur.

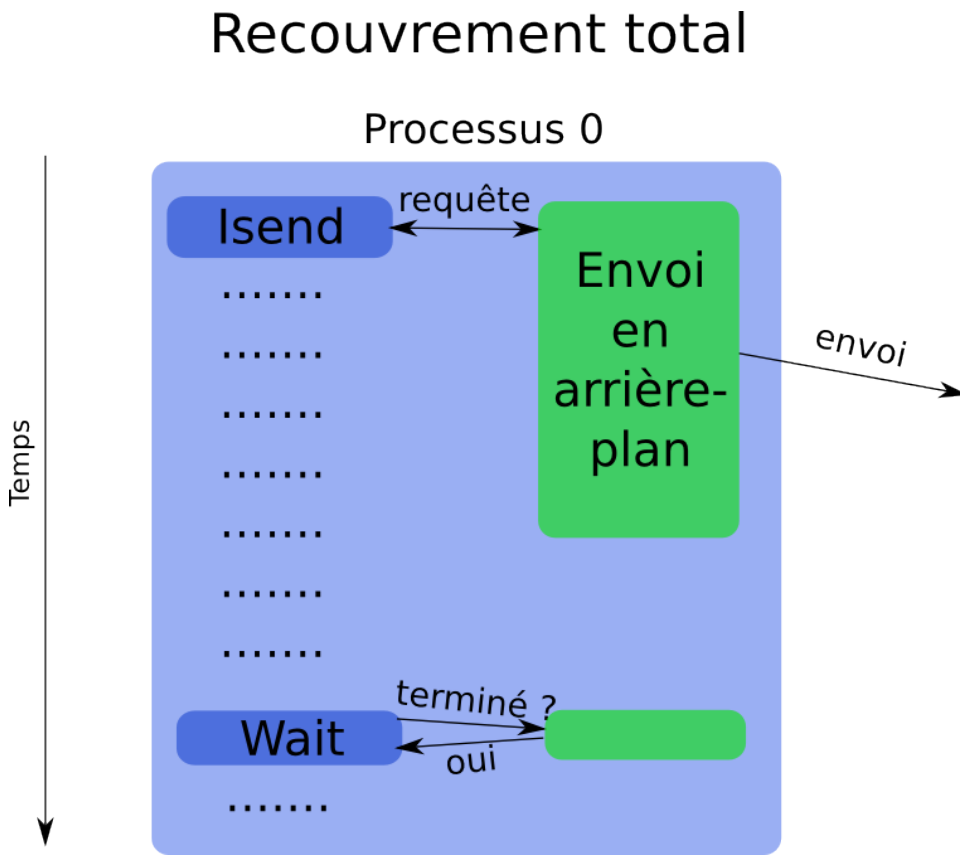
- Il est ainsi possible, si l'architecture matérielle et logicielle le permet, de masquer tout ou une partie des coûts de communications.
- Le recouvrement calculs-communications peut être vu comme un niveau supplémentaire de parallélisme.
- Cette approche s'utilise dans MPI par l'utilisation de sous-programmes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

Appels non bloquants

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

Modèles de communication





Modèles de communication

Avantages des appels non bloquants

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

Inconvénients des appels non bloquants

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

Modèles de communication

Interfaces

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

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

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

`MPI_Irecv()` pour les réceptions non bloquantes.

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

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

Modèles de communication

Interfaces

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

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

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

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

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

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

Modèles de communication

Interfaces

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

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

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

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

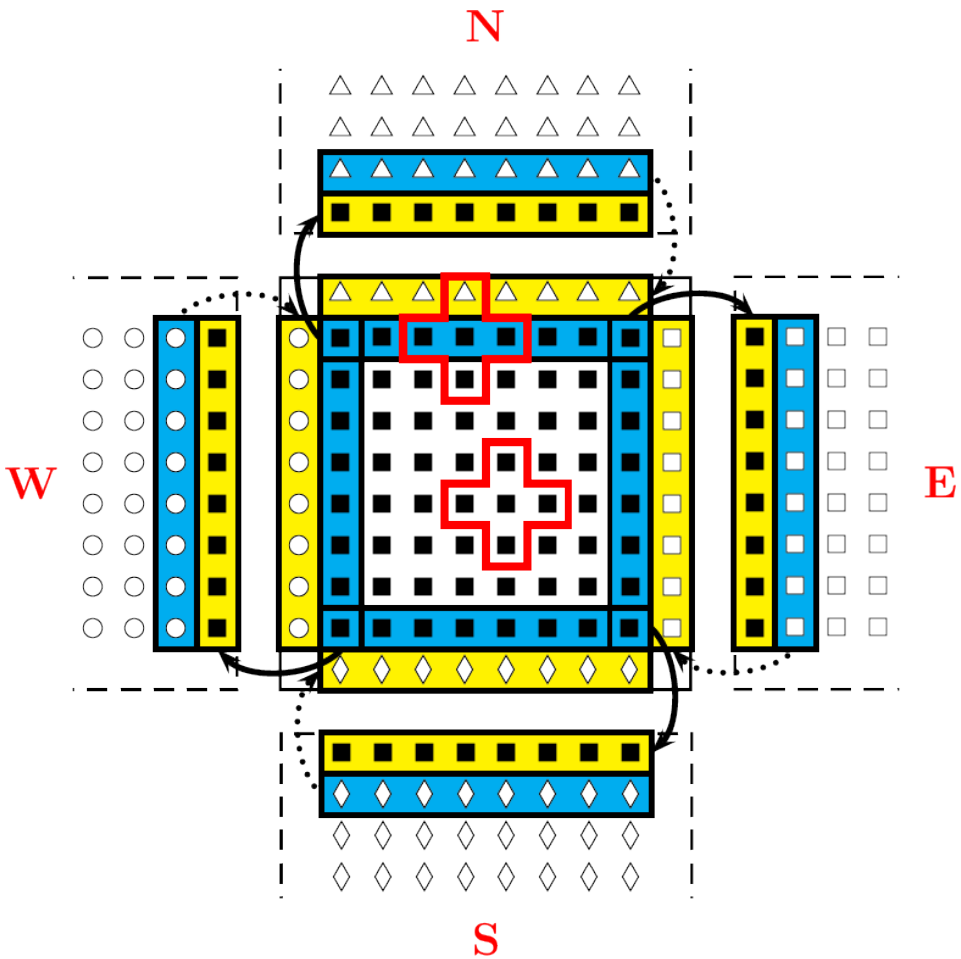
```
MPI_WAITSOME(taille, reqs, nbfin, indices, statuts, code)
MPI_TESTSOME(taille, reqs, nbfin, indices, statuts, code)

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

Modèles de communication

Gestion des requêtes

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

Modèles de communication



Modèles de communication

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

Modèles de communication

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

Modèles de communication

Niveau de recouvrement sur différentes machines

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

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

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

Modèles de communication

Communications collectives non bloquantes

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

```
MPI_IBARRIER(comm, request, code)

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

Modèles de communication

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

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

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

Modèles de communication

MPI-3 : fonctionnalités ajoutées

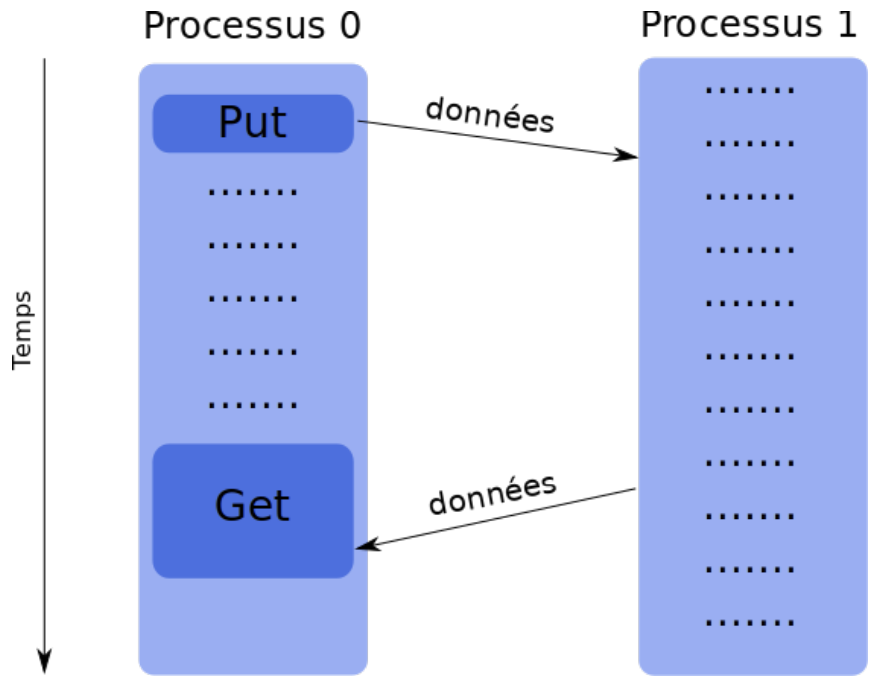
- Si MPI_SUBARRAYS_SUPPORTED est à *true*, il est possible d'utiliser des sections de tableaux pour les appels non bloquants.
- Si MPI_ASYNC_PROTECTS_NONBLOCKING est à *true*, les arguments d'envoi et/ou de réception sont *asynchronous* pour les interfaces des appels non bloquants.

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

Modèles de communication

Communications mémoire à mémoire (RMA)

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



Modèles de communication

RMA - Approche générale

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

RMA - Méthodes de synchronisation

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

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

Modèles de communication

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

Modèles de communication

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

Modèles de communication

Avantages des RMA

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

Inconvénients des RMA

- La gestion des synchronisations est délicate.
- Complexité et risques d'erreurs élevés.
- Moins performant que les communications point à point sur certaines machines.