Bases de la parallélisation par passage de message : MPI (Message Passing Interface)

Intervenant : Imane LASRI (imane.lasri@u-bourgogne.fr)
Centre de Calcul de l'université de Bourgogne (CCuB : ccub@u-bourgogne.fr)

12 février 2016

Sommaire

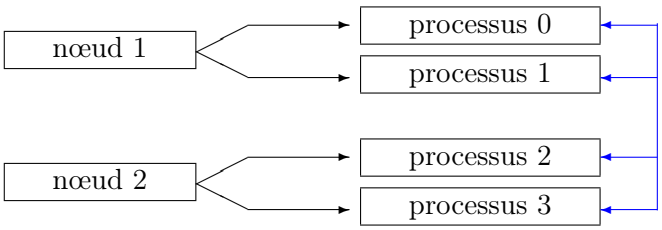
1 Introduction à la parallélisation à mémoire distribuée avec MPI

1.1 Principe

MPI (Message Passing Interface) est un ensemble de fonctions standardisées appartenant à une bibliothèque, utilisable avec les langages C/C++ et Fortran sur les architectures à mémoire distribuée. Le langage MPI permet d'exploiter plusieurs nœuds de calculs reliés par un réseau de communication.

1.2 Qu'est ce que le calcul parallèle avec MPI ?

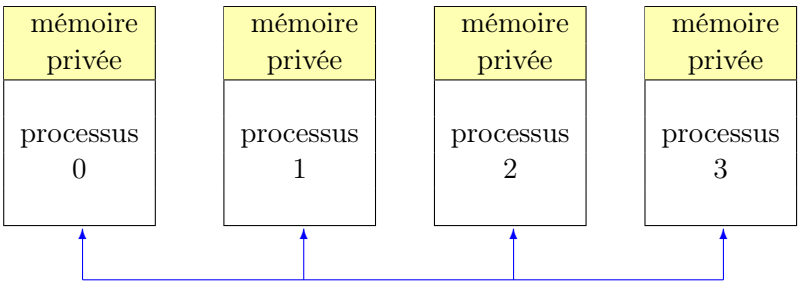
C'est la répartition des charges de calcul sur plusieurs cœurs de processeurs, appelés processus, dans le but de diminuer le temps d'exécution du programme. Ces processeurs peuvent appartenir à des machines différentes reliées par un réseau de communication qui permet les échanges de données par "passage de message"



— : Réseau de communication

1.3 Qu'est ce qu'une mémoire distribuée ?

La mémoire d'un système informatique multiprocesseur est dite distribuée lorsque la mémoire est répartie en plusieurs nœuds, chaque portion n'étant accessible qu'à certains processeurs. Un réseau de communication relie les différents nœuds, et l'échange de données doit se faire explicitement par « passage de messages ».



— : Réseau de communication

1.4 Différence avec OpenMP

MPI (**M**essage **P**assing **I**nterface) est utilisé sur des machines multiprocesseurs à mémoire distribuée (DMP : Distributed Memory Programming), c-à-d. que la mémoire est répartie sur plusieurs nœuds reliés par un réseau de communication. L'échange de données se fait par "passage de message".

OpenMP (**Open M**ulti-**P**rocessing) est utilisé sur des machines multiprocesseurs à mémoire partagée (SMP : Shared Memory Programming). OpenMP n'est utilisable que sur un seul nœuds de calcul.

1.5 Avantages de MPI

— Utilisable sur les architectures à mémoire distribuées : nombre de cœurs illimités.
— Machines peu coûteuses comparé aux machines SMP.

1.6 Compilation et exécution

Pour faire tourner un programme parallélisé en MPI il faut faire appel à une bibliothèque MPI qu'il faut installer. Il existe deux grandes bibliothèques open source :

- **MPICH** : `https://www.mpich.org/`
- **Open MPI** : `http://www.open-mpi.org/`

Sur les machines du centre de calcul, c'est la biblipthèqe OpenMPI qui est installée, pour l'utiliser, il faut charger le module en tapant :

```
$ module load openmpi
```

1.6.1 compilation

`programme en Fortran :`

```
$ mpif90 programme.f90
$ mpif77 programme.f
```

`programme en C :`

```
$ mpicc programme.c
```

`programme en C++ :`

```
$ mpicxx programme.C
```

1.6.2 Exécution

Pour l'exécution, on peut utiliser l'une des 3 commandes suivantes : `mpirun`, `mpiexec` ou `orterun` (tous des synonymes).

L'option `-n` (ou `-np`) suivi d'un entier, permet de spécifier le nombre de processus qu'on veut utiliser.

```
$ mpirun -n 4 a.out
```

ou :

```
$ mpiexec -n 4 a.out
```

ou :

```
$ orterun -n 4 a.out
```

2 Environnement MPI

2.1 MPI_INIT et MPI_FINALIZE

Pour utiliser MPI il faut inclure le module `MPI` en Fortran et le fichier d'inclusion `mpi.h` en C/C++. Les sous programmes `MPI_INIT()` et `MPI_FINALIZE()` permettent respectivement l'initialisation et la désactivation de l'environnement MPI. Ces derniers ne peuvent être appelés qu'une seule fois dans le programme.

Exemple 1 :

```
1  program init
2
3    use MPI
4    implicit none
5    integer :: code
6
7    call MPI_INIT(code)
8      write (*,*) "hello world !"
9    call MPI_FINALIZE(code)
10
11 end program init
```

```
$ mpif90 01_init.f90
$ mpirun -n 4 a.out
 hello world !
 hello world !
 hello world !
 hello world !
```

2.2 MPI_COMM_WORLD, _RANK et _SIZE

Le communicateur par défaut est `MPI_COMM_WORLD` qui regroupe tous les processus actifs (processus 0 à N) et est spécifié dans la plus part des fonctions MPI. Il est possible de définir d'autres communicateurs pour regrouper un sous ensemble de processus.

`MPI_COMM_RANK (comm , rang , code)` retourne le rang du processus.

`MPI_COMM_SIZE(comm, nbr_proc, code)` retourne le nombre de processus gérés par le communicateur.

Exemple 2 :

```
1  program communicateur
2    use MPI
3    implicit none
4    integer :: code, nbr_processus, rang
5    call MPI_INIT(code)
6    call MPI_COMM_SIZE(MPI_COMM_WORLD, nbr_processus, code)
7    call MPI_COMM_RANK(MPI_COMM_WORLD, rang, code)
8      write (*,*) "hello world ! depuis le processus : ", rang
9      if (rang==0) then
10     write(*,*) "Nombre total de processus :",nbr_processus
11     endif
12   call MPI_FINALIZE(code)
13 end program communicateur
```

```
$ mpif90 02_communicateur.f90
$ mpirun -n 4 a.out
 hello world ! depuis le processus :           2
 hello world ! depuis le processus :           0
 Nombre total de processus :           4
 hello world ! depuis le processus :           1
 hello world ! depuis le processus :           3
```

3 Communications point à point : MPI_SEND et MPI_RECV

Une communication point à point a lieu entre deux processus : un processus émetteur et un processus récepteur.

Pour que la communication ait lieu, il est nécessaire de connaître l'enveloppe du message, le type et la taille des donnés échangées. l'enveloppe est constituée de :

1) Le rang du processus émetteur (processus source)

2) Le rang du processus récepteur

3) Le tag (ou l'étiquette) du message

4) Le communicateur

L'envoi et la réception sont deux concepts fondamentaux de MPI. Presque toutes les fonctions de MPI sont basées sur les appelles d'envoi et de réception de messages.

`MPI_SEND` et `MPI_RECV` sont des opérations bloquantes, cela veut dire que le programme ne continuera que si le message est reçu.

```
MPI_SEND (message, taille, type, rang recepteur, tag, comm, code)
```

`message`	:	le message à envoyer, ça peut être un scalaire, un vecteur ou une matrice
`taille`	:	la taille du message à envoyer
`type`	:	le type du message à envoyer (voir tableau)
`rang`	:	le rang du processus qui reçoit le message
`tag`	:	entier positif quelconque qui sert à identifier le message envoyé
`comm`	:	le communicateur, par défaut : `MPI_COMM_WORLD`
`code`	:	code de retour, 0 si tout s'est bien passé

```
MPI_RECV (message, taille, type, rang emetteur, tag, comm, status, code)
```

`message`	:	le message à recevoir
`taille`	:	la taille du message à recevoir
`type`	:	le type du message à recevoir (voir tableau)
`rang`	:	le rang du processus qui envoie le message, pour recevoir de n'importe quel processus, `rang` peut prendre la valeur : `MPI_ANY_SOURCE`
`tag`	:	le même entier qu'à l'envoi, peut aussi prendre la valeur : `MPI_ANY_TAG`
`comm`	:	le communicateur, par défaut : `MPI_COMM_WORLD`
`status`	:	reçoit des informations sur la communication, c'est un tableau d'entiers positifs de taille `MPI_STATUS_SIZE`
`code`	:	code de retour, 0 si tout s'est bien passé

Type Fortran	**Type MPI**
integer	MPI_INTEGER
real	MPI_REAL
real(kind=8)	MPI_REAL8
double precision	MPI_DOUBLE_PRECISION
complex	MPI_COMPLEX
logical	MPI_LOGICAL
character	MPI_CHARACTER

Équivalent MPI des principaux types en Fortran

Exemple 3 :

Le vecteur `V` est initialisé sur le processus 0 et est envoyé ensuite au processus 1.

```
1  program send_recv
2
3    use MPI
4    implicit none
5    integer, parameter                      :: M=5
6    integer                                 :: i, nbr_procs, rang
7    integer, dimension(MPI_STATUS_SIZE)     :: statut
8    integer                                 :: code
9    integer, dimension(M)                   :: V
10
11   call MPI_INIT(code)
12
13   call MPI_COMM_SIZE(MPI_COMM_WORLD, nbr_procs, code)
14   call MPI_COMM_RANK(MPI_COMM_WORLD, rang, code)
15
16
17   if (rang==0) then
18     ! initialiser V sur le processus 0
19     do i=1,M
20       V(i)=i
21     enddo
22
23     ! afficher le nombre de processus utilises dans ce programme
24     write (*,*) "Dans cet exemple, il y a :", nbr_procs, "processus"
25
26     ! envoyer la valeur de V au processus 1
27     call MPI_SEND(V, M, MPI_INTEGER, 1, 100, MPI_COMM_WORLD, code)
28
29     ! afficher la valeur de V
30     write (*,*) "proc num", rang, "envoi le message : "
31     write (*,*) V
32
33   elseif (rang==1) then
34     ! reception de V sur le processus 1 depuis le processus 0
35     call MPI_RECV(V, M, MPI_INTEGER, 0, 100, MPI_COMM_WORLD, statut,code)
36
37     ! afficher la valeur recue de V
38     write (*,*) "proc num", rang, "a recu le message :"
39     write (*,*)  V
40
41   endif
42
43   call MPI_FINALIZE(code)
44
45 end program send_recv
```

```
$ mpif90 03_send_recv.f90
$ mpiexec -n 4 a.out

Dans cet exemple, il y a : 4  processus
proc num 0 envoi le message :
          1        2        3        4        5
proc num 1 a recu le message :
          1        2        3        4        5
```

4 Communications collectives

Une communication collective a lieu entre plusieurs processus : il y a plusieurs processessus émetteurs et plusieurs processus récepteurs.

4.1 Opérations de réduction : `MPI_REDUCE`

Lorsqu'un calcul est fait sur plusieurs processus et que le résultat final dépend du calcul de chaque processus (sommation, produit, rechercher un maximum...) on doit effectuer une opération de réduction pour que le résultat soit correct.

```
MPI_REDUCE (message_a_reduire, resultat_de_recution, taille, type,
                operation, comm, proc recepteur, code)
```

Le tableau suivant regroupe les opérations de réduction de MPI :

Opérations Fortran	Opérations MPI
sommation	MPI_SUM
produit	MPI_PROD
et logique	MPI_LAND
ou logique	MPI_LOR
maximum	MPI_MAX
minimum	MPI_MIN

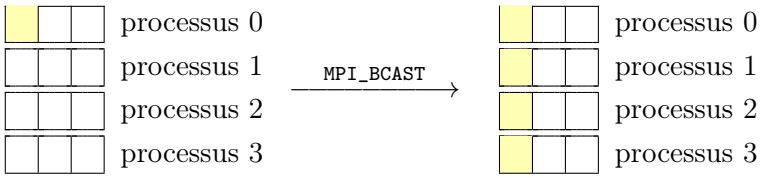
Exemple 4 :

```
1  program reduction
2
3    use MPI
4    implicit none
5    integer :: i, compteur=0, resultat_final=0
6    integer :: code, nbr_processus, rang
7
8    call MPI_INIT(code)
9    call MPI_COMM_SIZE(MPI_COMM_WORLD, nbr_processus, code)
10   call MPI_COMM_RANK(MPI_COMM_WORLD, rang, code)
11
12   do i=1,5
13      compteur=compteur+1
14   enddo
15   call MPI_REDUCE(compteur, resultat_final, 1, MPI_INTEGER, MPI_SUM, 0,
        MPI_COMM_WORLD, code)
16
17   write(*,*)"la valeur de a calculee par le proc ",rang, ":",compteur
18
19   call sleep (1)
20   write(*,*)"la valeur de a final par le proc ",rang, ":" , resultat_final
21
22   call MPI_FINALIZE(code)
23
24 end program reduction
```

```
  $ mpif90 04_reduction.f90
  $ mpirun -n 4 a.out

 la valeur de a calculee par le proc            0 :           5
 la valeur de a calculee par le proc            1 :           5
 la valeur de a calculee par le proc            2 :           5
 la valeur de a calculee par le proc            3 :           5

 la valeur de a final par le proc           2 :           0
 la valeur de a final par le proc           3 :           0
 la valeur de a final par le proc           0 :          20
 la valeur de a final par le proc           1 :           0
```

4.2 Envoie d'une donnée à tous les processus : MPI_BCAST

Cette fontion permet d'envoyer à l'ensemble des processus d'un communicateur un message défini sur un seul processus.




```
MPI_BCAST (message, taille_message, type_message, rang_source, comm, code)
```

Exemple 5 :

Dans cet exemple, l'entier `a` est initialisé à 0 sur tous les processus. La valeur de `a` sur le processus 0 est changée à 22.

Pour envoyer cette nouvelle valeur de `a` à tous les processus, il faut utiliser `MPI_BCAST`

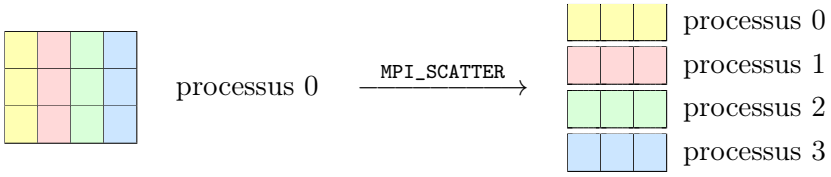
```
1  program bcast
2
3          use MPI
4          implicit none
5          integer :: code, rang
6          integer :: a=0
7
8          ! debut de la region parallele
9          call MPI_INIT (code)
10
11         ! retourner le rang de chaque processus
12         call MPI_COMM_RANK(MPI_COMM_WORLD, rang, code).
13
14         ! initialiser la variable "a" sur le processus 0
15         if (rang==0) then
16                 a=22
17         endif
18         ! afficher la valeur de "a" sur l'ensemble des processus
19         write (*,*) rang, ": avant le MPI_BCAST, a=",a
20
21         ! envoyer la valeur de "a" a tous les processus
22         call MPI_BCAST(a, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, code)
23         call sleep(1)
24
25         ! afficher la valeur de "a"
26         write (*,*) rang, ": apres le MPI_BCAST, a=",a
27
28         ! fin de la region parallele
29         call MPI_FINALIZE(code)
30
31 end program bcast
```

```
$ mpif90 05_bcast.f90
$ mpirun -n 4 a.out

           0 : avant le MPI_BCAST, a=          22
           1 : avant le MPI_BCAST, a=           0
           2 : avant le MPI_BCAST, a=           0
           3 : avant le MPI_BCAST, a=           0

           0 : apres le MPI_BCAST, a=          22
           1 : apres le MPI_BCAST, a=          22
           2 : apres le MPI_BCAST, a=          22
           3 : apres le MPI_BCAST, a=          22
```

4.3 Découper un message : MPI_SCATTER

4.3.1 MPI_SCATTER

Il est possible de diviser un message sur plusieurs processus. Pratique pour faire des calculs sur des matrices par exemple. MPI_SCATTER est utilisé pour diviser des messages de la même taille sur tous les processus.




```
MPI_SCATTER (message_a_envoye, taille_tranche_a_envoye,
type_du_message_a_envoye, tranche_recu, taille_tranche_recu,
rang_message_a_envoye, comm, code)
```

Exemple 6 :

Dans cet exemple on va initialiser une matrice A_{ij} de taille (M=4 x N=5) sur le processus 0, cette matrice est composée d'entiers positifs :

$$A_{ij} = \begin{pmatrix} A_{11} & A_{12} & A_{13} & A_{14} & A_{15} \\ A_{21} & A_{22} & A_{23} & A_{24} & A_{25} \\ A_{31} & A_{32} & A_{33} & A_{34} & A_{35} \\ A_{41} & A_{42} & A_{43} & A_{44} & A_{45} \end{pmatrix} = \begin{pmatrix} 1 & 5 & 9 & 13 & 17 \\ 2 & 6 & 10 & 14 & 18 \\ 3 & 7 & 11 & 15 & 19 \\ 4 & 8 & 12 & 16 & 20 \end{pmatrix}$$

On va ensuite envoyer chaque colonne de la matrice A_{ij} sur un processus en utilisant MPI_SCATTER, chaque colonne sera affectée à un vecteur $A_{colonne}$ de taille M=4. Il faut donc avoir N processus (N=5)

$A_{colonne} =$

P0	P1	P2	P3	P4
1	5	9	13	17
2	6	10	14	18
3	7	11	15	19
4	8	12	16	20

```
MPI_SCATTER (A, 4, MPI_INTEGER, A_colonne, 4, 0, MPI_COMM_WORLD, code)
```

4.3.2 MPI_SCATTERV

MPI_SCATTERV est utilisé pour diviser des messages de tailles différentes

```
MPI_SCATTER (message_a_envoye, tailles_tranches_a_envoye, deplacement,
    type_du_message_a_envoye, tranche_recu, taille_tranches_recues,
rang_message_a_envoye, comm, code)
```

Voir MPI_GATHERV pour plus de détails

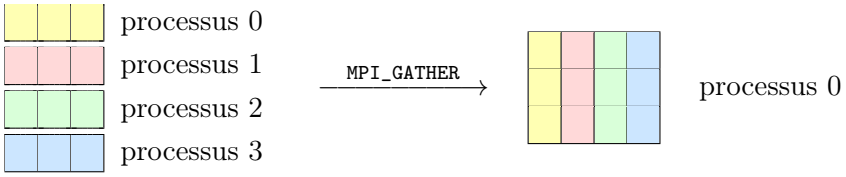
```
1  program scatter
2
3    use MPI
4    implicit none
5    integer, parameter                    :: M=4, N=5
6    integer                               :: i, j, x, nbr_procs, rang
7    integer, dimension(MPI_STATUS_SIZE)   :: statut
8    integer                               :: code
9    integer, dimension(M,N)               :: A
10   integer, dimension(M)                 :: A_colonne
11
12   call MPI_INIT(code)
13   call MPI_COMM_SIZE(MPI_COMM_WORLD, nbr_procs, code)
14   call MPI_COMM_RANK(MPI_COMM_WORLD, rang, code)
15
16   if (rang==0) then
17     ! initialiser la matrice A sur le processus 0
18     x=1
19     do j=1,N
20       do i=1,M
21         A(i,j)=x
22         x=x+1
23       enddo
24     enddo
25
26     ! afficher la valeur de la matrice A
27     write(*,*) "La matrice A"
28     do i=1,M
29       write(*,*) A(i,:)
30     enddo
31   endif
32
33   ! decouper la matrice A en A_colonne
34   call MPI_SCATTER(A, M, MPI_INTEGER,A_colonne,M, MPI_INTEGER,0,
       MPI_COMM_WORLD, code)
35   call sleep(1)
36
37   ! afficher la valeur de A_colonne sur chaque processus
38   write(*,*) rang, ":", A_colonne(:)
39
40   call MPI_FINALIZE(code)
41
42 end program scatter
```

```
$ mpif90 06_scatter.f90
$ mpiexec -n 5 a.out
La matrice A
           1           5           9          13          17
           2           6          10          14          18
           3           7          11          15          19
           4           8          12          16          20

           0 :           1           2           3           4
           1 :           5           6           7           8
           2 :           9          10          11          12
           3 :          13          14          15          16
           4 :          17          18          19          20
```

4.4 Regrouper des tranches de message

4.4.1 MPI_GATHER

`MPI_GATHER` est utilisé pour regrouper des tranches de message de la même taille se trouvant sur différents processus :




```
MPI_GATHER (tranche_a_envoyer, taille_tranche_a_envoyer,
type_tranche_a_envoyer, message_recu, taille_tranche_reçue,
    type_message_recu, rang_message_recu, comm, code)
```

Exemple 7 :

Dans cet exemple on va déclarer une variable `x` sur tous les processus, la valeur de `x` est égale au numéro du processus correspondant :
processus 0, x=0 ; processus 1, x=1 ; ... ; processus N, x=N

Les valeurs de `x` seront ensuite regroupées dans un vecteur `V` sur le processus 0. La taille du vecteur `V` correspond donc au nombre de processus utilisés.

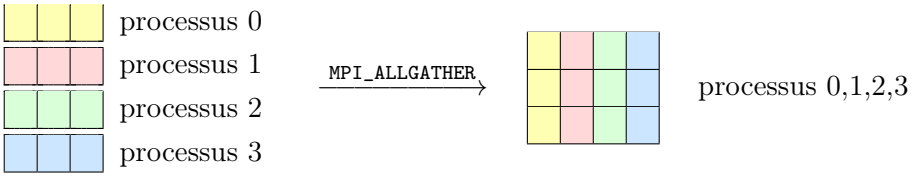
```
1  program gather
2
3    use MPI
4    implicit none
5    integer                                 :: x, nbr_procs, rang
6    integer                                 :: code
7    integer, dimension(:), allocatable      :: V
8
9    call MPI_INIT(code)
10   call MPI_COMM_SIZE(MPI_COMM_WORLD, nbr_procs, code)
11   call MPI_COMM_RANK(MPI_COMM_WORLD, rang, code)
12
13   allocate(V(nbr_procs))
14   x=rang
15
16   call MPI_GATHER(x, 1, MPI_INTEGER, V, 1, MPI_INTEGER,0, MPI_COMM_WORLD, &
         code)
17
18   write(*,*) rang, ":", V
19
20   call MPI_FINALIZE(code)
21
22 end program gather
```

```
$ mpif90 07_gather.f90
$ mpiexec -n 4 a.out
          3 :     8907280           0    16812928    50331648
          1 :    32254480           0    16812928    16777216
          2 :    35432976           0    16812928    33554432
          0 :           0           1           2           3
```

4.4.2 MPI_ALLGATHER

MPI_ALLGATHER correspond à MPI_GATHER suivi d'un MPI_BCAST sans spécifier le rang du processus :

```
MPI_GATHER (tranche_a_envoyer, taille_tranche_a_envoyer,
type_tranche_a_envoyer, message_recu, taille_tranche_reçue,
type_message_recu, comm, code)
```



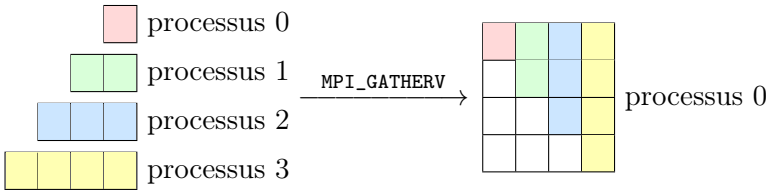
Si on remplace MPI_GATHER par MPI_ALLGATHER dans l'exemple précédent on va obtenir :

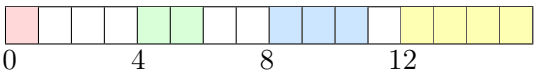
```
$ mpif90 04_gather.f90
$ mpiexec -n 4 a.out
          0 :           0           1           2           3
          1 :           0           1           2           3
          2 :           0           1           2           3
          3 :           0           1           2           3
```

4.4.3 MPI_GATHERV

MPI_GATHERV est la même fonction que MPI_GATHER sauf que les messages qui seront envoyés ont une taille différente.

```
MPI_GATHERV (tranche, taille_tranche, type_tranche, message,
nb_elements_tranche, deplacement, type_message, rang_message, comm, code)
```





tranche	:	une tranche de message à envoyer
taille tranche	:	la taille de la tranche de message à envoyer (varie selon les processus)
type tranche	:	le type de la tranche de message à envoyer (ex. MPI_INTEGER)
message	:	le message global qui sera reçu
nb elem tranche	:	un vecteur qui va regrouper le nombre d'élements de chaque tranche qui sera envoyé, ici nb_elem_tranche=(1,2,3,4)
déplacement	:	un vecteur qui regroupe la position en mémoire de chaque début de tranche de message envoyé, ici deplacement=(0,4,8,12)
type message	:	le type du message global reçu
rang message	:	le rang du processus du message global
comm	:	le communicateur, par défaut : MPI_COMM_WORLD
code	:	code de retour, 0 si tout s'est bien passé

Exemple 8 :

Dans cet exemple, on déclare un vecteur `vecteur` sur plusieurs processus, la taille de `vecteur` sera différente sur chaque processus (taille vecteur = rang processus + 1), on choisit 4 processus :

vecteur =

P0	P1	P2	P3
1	1	1	1
	2	2	2
		3	3
			4

On souhaite regrouper ces vecteurs en une seule matrice `matrice` de taille `(nbr_procs x nbr_procs) = (4x4)`

La matrice `matrice` sera allouée et initialisée à 0 sur le processus 0 uniquement, en regroupant tous les vecteurs :

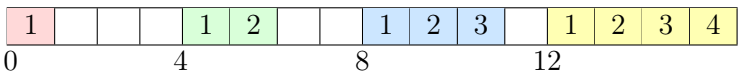
$$matrice_{(ij)} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 2 & 2 & 2 \\ 0 & 0 & 3 & 3 \\ 0 & 0 & 0 & 4 \end{pmatrix}$$

Pour regrouper les vecteurs de tailles différentes, on fera appel à `MPI_GATHERV` :

```
MPI_GATHERV (tranche, taille_tranche, type_tranche, message,
nb_elements_tranche, deplacement, type_message, rang_message, comm, code)
```

où :

`tranche` : `vecteur`
`taille tranche` : `rang+1`
`type tranche` : `MPI_INTEGER`
`message` : `matrice`
`nb elem tranche` : `(1,2,3,4)` un vecteur de taille `nbr_procs=4`
`déplacement` : `(0,4,8,12)`



shéma de la matrice en mémoire

`type message` : `MPI_INTEGER`
`rang message` : 0
`comm` : `MPI_COMM_WORLD`
`code` : code

```
1  program gatherv
2
3    use MPI
4    implicit none
5    integer                                  :: x, nbr_procs, rang
6    integer                                  :: code,i
7    integer, dimension(:), allocatable       :: vecteur, nb_elements_recus, &
         deplacement
8    integer, dimension(:,:), allocatable     :: matrice
9
10   call MPI_INIT(code)
11   call MPI_COMM_SIZE(MPI_COMM_WORLD, nbr_procs, code)
12   call MPI_COMM_RANK(MPI_COMM_WORLD, rang, code)
13
14   ! allouer et initialiser "vecteur" sur tous les processus
15   allocate(vecteur(rang+1))
16   x=1
17   do i=1,rang+1
18         vecteur(i)=x
19         x=x+1
20   enddo
21   ! afficher la valeur de chaque vecteur sur les differents processus
22   write(*,*) rang, ":", vecteur
23
24   if (rang==0) then
25   ! allouer et initialiser : matrice, nb_elements_recus et deplacement
26     allocate(matrice(nbr_procs,nbr_procs))
27     matrice(:,:)=0
28     allocate(nb_elements_recus(nbr_procs))
29     allocate(deplacement(nbr_procs))
30     do i=1,nbr_procs
31       nb_elements_recus(i)=i
32     enddo
33     deplacement(1)=0
34     do i=2,nbr_procs
35       deplacement(i)=deplacement(i-1)+nbr_procs
36     enddo
37   endif
38
39   ! regrouper les vecteurs dans la matrice
40   call MPI_GATHERV(vecteur, rang+1, MPI_INTEGER, matrice, &
         nb_elements_recus, deplacement,MPI_INTEGER, 0, MPI_COMM_WORLD,code)
41
42   call sleep(1)
43   ! afficher la matrice sur le processus 0
44   if (rang==0) then
45     write(*,*) "le processus", rang, "a recu : "
46     do i=1,nbr_procs
47      write(*,*) matrice(i,:)
48     enddo
49   endif
50
51   call MPI_FINALIZE(code)
52
53 end program gatherv
```

```
$ mpif90 gatherv.f90
$ mpirun -n 4 a.out
          0 :           1
          1 :           1           2
          2 :           1           2           3
          3 :           1           2           3           4
le processus  0  a recu :
          1           1           1           1
          0           2           2           2
          0           0           3           3
          0           0           0           4
```

4.4.4 MPI_ALLGATHERV

`MPI_ALLGATHERV` correspond à `MPI_GATHERV` suivi d'un `MPI_BCAST` sans spécifier le rang du processus (exemple d'application : voire exercices).

5 Type de données dérivées

Les types qu'on a vu jusqu'ici sont représentés dans le tableau § 3 , on peut créer de nouveaux types plus complexes pour regrouper un ensemble de données (ensemble de données non-contiguë en mémoire par exemple). Il faut respecter 3 étapes :

1) Création du nouveau type avec l'une des fonctions MPI qu'on va voir dans ce qui suit

2) Validation du nouveau type avec `MPI_TYPE_COMMMIT(nouveau_type,code)`

3) Libération du nouveau type avec `MPI_TYPE_FREE(nouveau_type,code)`

5.1 Type contiguë : MPI_TYPE_CONTIGUOUS

Permet de définir des types dérivés sur des données contiguës en mémoire, c'est à dire que les valeurs se suivent en mémoire.

```
MPI_TYPE_CONTIGUOUS (nb_elements, ancien_type, nouveau_type, code)
```

Exemple 9 :
Soit la matrice A sur le processus 0, tel que :

$$A_{(ij)}^{P0} = \begin{pmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{pmatrix}$$

On souhaite créer une nouvelle matrice A sur le processus 1 qui contient deux colonnes de la matrice A du processus 0 :

$$A_{(ij)}^{P1} = \begin{pmatrix} 0 & 1 & 5 & 0 \\ 0 & 2 & 6 & 0 \\ 0 & 3 & 7 & 0 \\ 0 & 4 & 8 & 0 \end{pmatrix}$$

Pour ce faire, on crée un type dérivée qui contient 8 éléments :

```
MPI_TYPE_CONTIGUOUS (8, MPI_INTEGER, colonne , code)
```

Remarque :
En Fortran les éléments d'une colonne se suivent en mémoire, en C/C++ c'est les éléments d'une ligne qui se suivent en mémoire.

```
1  program contiguous
2
3    use MPI
4    implicit none
5    integer, parameter                      :: M=4, N=4
6    integer                                 :: i, j, x, rang, colonne
7    integer, dimension(MPI_STATUS_SIZE)     :: statut
8    integer                                 :: code
9    integer, dimension(M,N)                 :: A
10
11   ! initialisation de MPI
12   call MPI_INIT(code)
13   call MPI_COMM_RANK(MPI_COMM_WORLD, rang, code)
14
15   ! creation du nouveau type derive : colonne
16   call MPI_TYPE_CONTIGUOUS(M*2, MPI_INTEGER, colonne,code) ! creation
17   call MPI_TYPE_COMMIT(colonne, code)                      ! validation
18
19   if (rang==0) then
20     ! initialisation de la matrice A sur le processus 0
21     x=1
22     do j=1,N
23       do i=1,M
24         A(i,j)=x
25         x=x+1
26       enddo
27     enddo
28
29     ! affichage la matrice A
30     write(*,*) "La matrice A, proc : ", rang
31     do i=1,M
32       write(*,*) A(i,:)
33     enddo
34
35     ! envoie d'un element de type "colonne" qui commence par A(1,1)
36     call MPI_SEND(A(1,1), 1, colonne, 1, 100, MPI_COMM_WORLD, code)
37
38   elseif (rang==1) then
39     ! reception de l'element envoye par le processus 0
40     call MPI_RECV(A(1,2), 1, colonne, 0, 100, MPI_COMM_WORLD, statut,code)
41
42    ! affichage de la matrice recu
43     write(*,*) "La matrice A recu par le proc :", rang
44     do i=1,M
45       write(*,*) A(i,:)
46     enddo
47
48   endif
49   ! liberer le type derive
50   call MPI_TYPE_FREE(colonne,code)
51   ! fin de la region parallele
52   call MPI_FINALIZE(code)
53
54 end program contiguous
```

```
$ mpif90 09_type_contiguous.f90
$ mpirun -n 4 a.out
 La matrice A, proc :  0
           1           5           9          13
           2           6          10          14
           3           7          11          15
           4           8          12          16
 La matrice A recu par le proc :  1
           0           1           5           0
           0           2           6           0
           0           3           7           0
           0           4           8           0
```

5.2 Type non contiguë : MPI_TYPE_VECTOR

Très utile, car il permet de faire des opérations sur des données non-contiguës en mémoire, ainsi, une matrice en Fortran peut alors être découpée selon les lignes.

```
MPI_TYPE_VECTOR (nb_elements, nb_lignes, deplacement, ancien_type,
                 nouveau_type, code)
```

Exemple 10 :
Soit la matrice A de taille MxN sur le processus 0, tel que :

$$A_{(ij)}^{P0} = \begin{pmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{pmatrix}$$

On souhaite créer une nouvelle matrice A sur le processus 1 qui contient deux lignes de la matrice A du processus 0 :

$$A_{(ij)}^{P1} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Pour ce faire, on crée un type dérivé avec `MPI_TYPE_VECTOR`, où :

`nb_elements`	:	nombre d'éléments dans une ligne : `N`
`nb_lignes`	:	2
`deplacement`	:	nombre d'élément dans une colonne : `M`
`ancien_type`	:	`MPI_INTEGER`
`nouveau_type`	:	`ligne`
`code`	:	`code`

```
MPI_TYPE_VECTOR (N, 2, M, MPI_INTEGER, ligne, code)
```

Remarque :
Même remarque que pour `MPI_TYPE_CONTIGUOUS`, En C/C++ `MPI_TYPE_VECTOR` sert à manipuler les colonnes des matrices car les lignes sont contiguës en mémoire.

```fortran
program vector

  use MPI
  implicit none
  integer, parameter                      :: M=4, N=4
  integer                                 :: i, j, x, rang, ligne
  integer, dimension(MPI_STATUS_SIZE)     :: statut
  integer                                 :: code
  integer, dimension(M,N)                 :: A

  ! debut de la region parallele
  call MPI_INIT(code)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rang, code)

  ! creation du type derive : ligne
  call MPI_TYPE_VECTOR(N,2,M, MPI_INTEGER, ligne,code) ! creation
  call MPI_TYPE_COMMIT(ligne, code)                    ! validation

  if (rang==0) then
    ! initialisation de la matrice A sur le processus 0
    x=1
    do j=1,N
      do i=1,M
        A(i,j)=x
        x=x+1
      enddo
    enddo

    ! affichage la matrice A
    write(*,*) "La matrice A, proc : ", rang
    do i=1,M
      write(*,*) A(i,:)
    enddo

    ! envoie d'un element de type "ligne" qui commence par A(1,1)
    call MPI_SEND(A(1,1), 1, ligne, 1, 100, MPI_COMM_WORLD, code)

  elseif (rang==1) then
    ! reception de l'element envoye par le processus 0
    call MPI_RECV(A(2,1), 1, ligne, 0, 100, MPI_COMM_WORLD, statut,code)

    ! affichage de la matrice recu
    write(*,*) "La matrice A recu par le proc :", rang
    do i=1,M
      write(*,*) A(i,:)
    enddo

  endif
  ! liberer le type derive
  call MPI_TYPE_FREE(ligne,code)
  ! fin de la region parallele
  call MPI_FINALIZE(code)

end program vector
```

```
$ mpif90 10_type_vector.f90
$ mpirun -n 4 a.out
 La matrice A, proc :            0
           1           5           9          13
           2           6          10          14
           3           7          11          15
           4           8          12          16
 La matrice A recu par le proc :            1
           0           0           0           0
           1           5           9          13
           2           6          10          14
           0           0           0           0
```

5.3 Type non contiguë : MPI_TYPE_CREATE_HVECTOR

C'est la même chose que MPI_TYPE_VECTOR sauf que deplacement sera en octets. Utilisé lorsque le type à remplacer n'est pas un type de base MPI (MPI_INTEGER, MPI_REAL...).

deplacement est de type : INTEGER(kind=MPI_ADDRESS_KIND)

```
MPI_TYPE_CREATE_HVECTOR (nb_blocs, taille_bloc, deplacement, ancien_type,
                         nouveau_type, code)
```

Pour avoir la taille d'un type on fait appel au sous programme : MPI_TYPE_SIZE

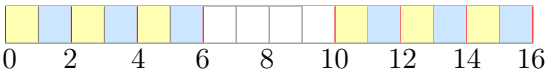
```
MPI_TYPE_SIZE (TYPE_MPI, taille, code)
```

où TYPE_MPI est soit un type de base (MPI_INTEGER, MPI_REAL... etc.) soit un type dérivé

Exemple 11 :

Supposons qu'on crée un type ancien_type qui contient deux données contiguës en mémoire :

On veut créer un nouveau type nouveau_type à partir de ancien_type avec HVECTOR



nb_blocs	:	2
taille_blocs	:	3
deplacement	:	10 * taille_integer, c'est l'écart entre le début du premier bloc et le début du 2ème bloc (ainsi de suite : entre le 2ème et le 3ème ...)
ancien_type	:	ancien_type
nouveau_type	:	nouveau_type
code	:	code

Si on applique nouveau_type à la matrice A_{ij} :

$$A_{ij}^{P0} = \begin{pmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{pmatrix} \qquad A_{ij}^{P1} = \begin{pmatrix} 1 & 5 & 0 & 13 \\ 2 & 6 & 0 & 14 \\ 3 & 0 & 11 & 15 \\ 4 & 0 & 12 & 16 \end{pmatrix}$$

MPI_SEND nouveau_type ⟶ MPI_RECV nouveau_type

$$A_{ij}^{P0} = \begin{pmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{pmatrix} \qquad A_{ij}^{P2} = \begin{pmatrix} 1 & 5 & 13 & 0 \\ 2 & 6 & 14 & 0 \\ 3 & 11 & 15 & 0 \\ 4 & 12 & 16 & 0 \end{pmatrix}$$

MPI_SEND nouveau_type ⟶ MPI_RECV MPI_INTEGER

Remarque :

Lorsque le type reçu est un type de base, les espaces entre les blocs sont supprimés, ces blocs vont donc se suivre

```
1  program hvector
2
3    use MPI
4    implicit none
5    integer                    :: i, j, x
6    integer, parameter         :: M=4, N=4
7    integer, dimension (M,N)   :: A
8    integer        :: nbr_procs, rang, code
9    integer        :: taille_ancien_type, taille_nouveau_type, taille_integer
10   integer        :: ancien_type, nouveau_type
11
12   integer(kind=MPI_ADDRESS_KIND)      :: deplacement
13   integer, dimension(MPI_STATUS_SIZE) :: statut
14
15   call MPI_INIT(code)
16   call MPI_COMM_SIZE(MPI_COMM_WORLD,nbr_procs,code)
17   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
18
19   ! creation du type : ancien_type
20   call MPI_TYPE_CONTIGUOUS(2, MPI_INTEGER, ancien_type, code)
21   call MPI_TYPE_SIZE(MPI_INTEGER, taille_integer, code)
22
23   deplacement=taille_integer*10
24
25   ! creation du type : nouveau_type
26   call MPI_TYPE_CREATE_HVECTOR(2, 3,deplacement,ancien_type, &
27                                nouveau_type, code)
28   call MPI_TYPE_COMMIT(nouveau_type, code)
29
30   ! taille de ancien_type
31   call MPI_TYPE_SIZE(ancien_type, taille_ancien_type, code)
32   if (rang==0) then
33     write (*,*) "le type 'ancien_type' fait : ", &
34                 taille_ancien_type, "octets"
35     write (*,*) "  ce qui fait : ", &
36                 taille_ancien_type/taille_integer,"entiers"
37   end if
38
39   ! taille de nouveau_type
40   call MPI_TYPE_SIZE(nouveau_type, taille_nouveau_type, code)
41   if (rang==0) then
42     write (*,*) "le type 'nouvaeu_type' fait : ", &
43                 taille_nouveau_type,"octets"
44     write (*,*) "  ce qui fait : ", &
45                 taille_nouveau_type/taille_integer,"entiers"
46   end if
47
48   ! initialisation de A sur tous les processus
49   A(:,:)= 0
50
51   if (rang==0) then
52     ! initialisation de A sur le processus 0
53     x=0
54     do j=1,N
55       do i=1,M
56         x=1+x
57         A(i,j)=x
58       enddo
59     enddo
60     write (*,*) "Matrice A sur le processus:", rang
61     do i=1, M
62       write (*,*) A(i,:)
63     end do
64     write (*,*)
65     ! envoie de A sur le processus 1 et sur le processus 2
66     call MPI_SEND(A, 1, nouveau_type, 1, 101, MPI_COMM_WORLD,code)
67     call MPI_SEND(A, 1, nouveau_type, 2, 102, MPI_COMM_WORLD,code)
```

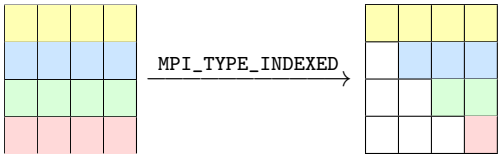
```
68    elseif (rang==1) then
69      call MPI_RECV(A, 1, nouveau_type, 0, 101, MPI_COMM_WORLD,statut, code)
70      write (*,*) "Matrice A sur le processus:", rang
71      do i=1,M
72        print *, A(i,:)
73      end do
74      write (*,*)
75
76    elseif (rang==2) then
77      call MPI_RECV(A, taille_nouveau_type/taille_integer, MPI_INTEGER, &
78               0, 102,MPI_COMM_WORLD,statut, code)
79      write (*,*) "Matrice A sur le processus:", rang
80      do i=1,M
81        print *, A(i,:)
82      end do
83
84    endif
85
86    call MPI_TYPE_FREE(nouveau_type, code)
87    call MPI_FINALIZE(code)
88
89  end program hvector
```

```
$ mpif90 11_type_hvector.f90
$ mpirun -n 3 a.out
 le type 'ancien_type' fait :            8 octets
   ce qui fait :            2 entiers
 le type 'nouvaeu_type' fait :           48 octets
   ce qui fait :           12 entiers
 Matrice A sur le processus:           0
           1           5           9          13
           2           6          10          14
           3           7          11          15
           4           8          12          16

 Matrice A sur le processus:           1
           1           5           0          13
           2           6           0          14
           3           0          11          15
           4           0          12          16

 Matrice A sur le processus:           2
           1           5          13           0
           2           6          14           0
           3          11          15           0
           4          12          16           0
```

5.4 Type non contiguë à pas variable : MPI_TYPE_INDEXED

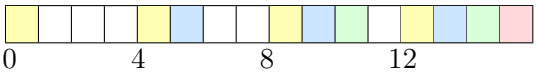
Permet de définir un type sur des données non-contiguës en mémoire avec un déplacement variable

```
MPI_TYPE_INDEXED (nb_blocs, taille_bloc, deplacement, ancien_type,
                  nouveau_type,code)
```



Position en mémoire des éléments avant :

Position en mémoire des éléments après :



`nb blocs`	:	nombre de blocs (colonnes) dans le type indexed, ici 4 colonnes
`taille bloc`	:	un vecteur de taille "nb blocs" qui regroupe la taille de chaque bloc ici `taille_bloc=(1,2,3,4)`
`deplacement`	:	un vecteur de taille "nb blocs" qui regroupe la position en mémoire de chaque début de bloc, ici `deplacement=(0,4,8,12)`
`ancien type`	:	ancien type à remplacer
`nouveau type`	:	nom du nouveau type
`code`	:	code de retour, 0 si tout s'est bien passé

Exemple 12 :
Soit la matrice A de taille MxN sur le processus 0, tel que :

$$A_{(ij)}^{P0} = \begin{pmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{pmatrix}$$

On souhaite créer une nouvelle matrice A sur le processus 1 qui contient les éléments triangulaires supérieurs formés sur la diagonale de la matrice A du processus 0 :

$$A_{(ij)}^{P1} = \begin{pmatrix} 1 & 5 & 9 & 13 \\ 0 & 6 & 10 & 14 \\ 0 & 0 & 11 & 15 \\ 0 & 0 & 0 & 16 \end{pmatrix}$$

```
1  program indexed
2    use MPI
3    implicit none
4    integer, dimension(MPI_STATUS_SIZE)   :: statut
5    integer                               :: nbr_procs, rang, code
6    integer                               :: i, j, x=1
7    integer, parameter                    :: M=4, N=4
8    integer, dimension(M,N)               :: matrice_pleine, matrice_triangulaire
9    integer                               :: un_deplacement, une_taille_bloc
10   integer, dimension(M)                 :: deplacement_total, taille_bloc_total
11   integer, dimension(M)                 :: taille_bloc
12   integer                               :: type_triangulaire
13
14   call MPI_INIT(code)
15   call MPI_COMM_SIZE(MPI_COMM_WORLD, nbr_procs, code)
16   call MPI_COMM_RANK(MPI_COMM_WORLD, rang, code)
17
18   if (rang==0) then
19     ! initialisation de la matrice pleine
20     allocate(matrice_pleine(M,N))
21     do i=1,M
22       do j=1,N
23         matrice_pleine(i,j)=x
24         x=x+1
25       enddo
26     enddo
27     ! affichage de la matrice pleine
28     write(*,*) "matrice pleine : "
29     do i=1,M
30       write(*,*) matrice_pleine(i,:)
31     enddo
32   end if
33   ! calcul du deplcement et de la taille des blocs sur chaque processus
34   un_deplacement=rang*M
35   une_taille_bloc=rang+1
36
37   ! regrouper les deplacement et tailles blocs dans un vecteur
38   call MPI_ALLGATHER(un_deplacement, 1, MPI_INTEGER, deplacement_total, &
                        1, MPI_INTEGER, MPI_COMM_WORLD, code)
39   call MPI_ALLGATHER(une_taille_bloc, 1, MPI_INTEGER, taille_bloc_total, &
                        1, MPI_INTEGER, MPI_COMM_WORLD, code)
40
41   ! creation et validation du type derive
42   call MPI_TYPE_INDEXED(4, taille_bloc_total, deplacement_total, &
                           MPI_INTEGER, type_triangulaire, code)
43   call MPI_TYPE_COMMIT(type_triangulaire, code)
44
45   if (rang==0) then
46     ! envoie de la matrice triangulaire au processus 1
47     call MPI_SEND(matrice_pleine, 1, type_triangulaire, 1, 100, &
                     MPI_COMM_WORLD, code)
48   elseif (rang==1) then
49     ! reception de la matrice triangulaire envoye par le processus 0
50     matrice_triangulaire(:,:)=0
51     call MPI_RECV(matrice_triangulaire, 1, type_triangulaire, 0, 100, &
                     MPI_COMM_WORLD, statut, code)
52
53     ! affichage du resultat
54     write(*,*) "matrice triangulaire : "
55     do i=1,M
56       write(*,*) matrice_triangulaire(i,:)
57     enddo
58   endif
59
60   call MPI_TYPE_FREE(type_triangulaire,code)
61   call MPI_FINALIZE(code)
62 end program indexed
```

```
$ mpif90 11_type_indexed.f90
$ mpirun -n 4 a.out
           1           2           3           4
           5           6           7           8
           9          10          11          12
          13          14          15          16

           1           2           3           4
           0           6           7           8
           0           0          11          12
           0           0           0          16
```

5.5 Type non contiguë à pas variable : MPI_TYPE_CREATE_HINDEXED

Tout comme MPI_TYPE_CREATE_HVECTOR, le deplacement de MPI_TYPE_CREATE_HINDEXED est donné en octets

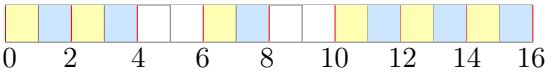
```
MPI_TYPE_CREATE_HINDEXED (nb_blocs, taille_bloc, deplacements, ancien_type,
                          nouveau_type,code)
```

pour obtenir la taille du pas en octets, on peut utiliser :

```
MPI_TYPE_SIZE (TYPE_MPI, taille, code)
```

Supposons qu'on crée un type ancien_type qui contient deux données contiguës en mémoire :

On veut créer un nouveau type nouveau_type à partir de ancien_type



nb_blocs	:	3
taille_blocs	:	(2,1,3) , chaque taille de bloc est un multiple de la taille de ancien_type
deplacements	:	(0,6,10)*taille_integer, position en mémoire (en octets) de chaque bloc
ancien_type	:	ancien_type
nouveau_type	:	nouveau_type
code	:	code

Si on applique nouveau_type à une matrice :

$$A_{ij}^{P0} = \begin{pmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{pmatrix} \qquad A_{ij}^{P1} = \begin{pmatrix} 1 & 0 & 0 & 13 \\ 2 & 0 & 0 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{pmatrix}$$

MPI_SEND nouveau_type $\longrightarrow$ MPI_RECV nouveau_type

$$A_{ij}^{P0} = \begin{pmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{pmatrix} \qquad A_{ij}^{P2} = \begin{pmatrix} 1 & 7 & 13 & 0 \\ 2 & 8 & 14 & 0 \\ 3 & 11 & 15 & 0 \\ 4 & 12 & 15 & 0 \end{pmatrix}$$

MPI_SEND nouveau_type $\longrightarrow$ MPI_RECV MPI_INTEGER

Remarque :
l'utilisation de MPI_TYPE_HINDEXED à la place de MPI_TYPE_CREATE_ HINDEXED est obsolète

```
program indexed
  use MPI
  implicit none
  integer                                :: nbr_procs, rang, code
  integer, dimension(MPI_STATUS_SIZE):: statut
  integer                                :: i, j, x=1
  integer, parameter                     :: M=4, N=4
  integer, dimension(M,N)                :: A=0
  integer, dimension(3)                  :: deplacements, taille_blocs
  integer                                :: taille_nouveau_type, taille_integer
  integer                                :: ancien_type, nouveau_type

  call MPI_INIT(code)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, nbr_procs, code)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rang, code)
  ! creation du type derive : ancien type
  call MPI_TYPE_CONTIGUOUS(2, MPI_INTEGER, ancien_type, code)
  call MPI_TYPE_SIZE(MPI_INTEGER, taille_integer, code)
  if (rang==0) then
    ! initialisation de la matrice A
    x=0
    do j=1,N
      do i=1,M
        x=1+x
        A(i,j)=x
      enddo
    enddo
    ! affichage de la matrice A
    write(*,*) "matrice A sur le processus :", rang
    do i=1,M
      write(*,*) A(i,:)
    enddo
  end if
  ! initialisation de taille_blocs et deplacements
  taille_blocs=(/2,1,3/)
  deplacements=(/0,6*taille_integer,10*taille_integer/)

  ! creation du type derive : nouveau type
  call MPI_TYPE_CREATE_HINDEXED(3, taille_blocs, deplacements, &
                    ancien_type, nouveau_type, code)
  call MPI_TYPE_SIZE(nouveau_type, taille_nouveau_type, code)
  call MPI_TYPE_COMMIT(nouveau_type, code)

  if (rang==0) then
    ! envoie de la matrice A au processus 1 et 2
    call MPI_SEND(A, 1, nouveau_type, 1, 101, MPI_COMM_WORLD, code)
    call MPI_SEND(A, 1, nouveau_type, 2, 102, MPI_COMM_WORLD, code)
  elseif (rang==1) then
    ! reception de la matrice A envoye par le processus 0
    call MPI_RECV(A, 1, nouveau_type, 0, 101, MPI_COMM_WORLD, statut, code)
    write(*,*) "matrice A sur le processus :", rang
    do i=1,M
      write(*,*) A(i,:)
    enddo
  elseif (rang==2) then
    ! reception de la matrice A envoye par le processus 0
    call MPI_RECV(A, taille_nouveau_type/taille_integer, &
            MPI_INTEGER, 0, 102, MPI_COMM_WORLD, statut, code)
    write(*,*) "matrice A sur le processus :", rang
    do i=1,M
      write(*,*) A(i,:)
    enddo
  endif
```

```
64    call MPI_TYPE_FREE(nouveau_type,code)
65    call MPI_FINALIZE(code)
66
67  end program indexed
```

```
$ mpif90 13_type_hindexed.f90
$ mpirun -n 3 a.out
 matrice A sur le processus :            0
           1            5            9           13
           2            6           10           14
           3            7           11           15
           4            8           12           16
 matrice A sur le processus :            1
           1            0            0           13
           2            0            0           14
           3            7           11           15
           4            8           12           16
 matrice A sur le processus :            2
           1            7           13            0
           2            8           14            0
           3           11           15            0
           4           12           16            0
```

6 Bibliographie

Le site officiel de la librairie open-mpi :
https ://www.open-mpi.org

L'IDRIS :
http ://www.idris.fr/data/cours/parallel/mpi/

MPI : A Message-Passing Interface Standard Version 3.1, june 2015 :
http ://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

Exercices

Exercice 1 : Calcul de π

On souhaite paralléliser un programme séquentiel pour le calcul de π par la méthode des rectangles sur `n` intervalles avec un pas `h`

$$\pi = \int_0^1 \frac{4}{1+x^2}\, dx$$

1) Calculer le temps réel de l'exécution du programme en utilisant MPI_WTIME()

2) Afficher le nombre de processus utilisé dans ce programme

3) Paralléliser le calcul de PI avec MPI

Programme séquentiel :

```
1  program pi
2
3  implicit none
4
5      integer, parameter :: n=100000000
6      double precision   :: f, x, a, h
7      double precision   :: Pi_calcule, Pi_reel, ecart
8      integer            :: i
9      integer            :: t1, t2, ir
10     real               :: temps
11     integer            :: code, rang, nbr_procs
12
13     ! Fonction instruction a integrer
14     f(a) = 4.0_16 / ( 1.0_16 + a*a )
15
16     ! valeur reel de Pi avec 15 chiffres apres la virgule
17     Pi_reel = 3.141592653589793
18
19     ! Longueur de l'intervalle d'integration.
20     h = 1.0_8 / real(n,kind=8)
21
22     ! calcul du temps de l'execution du programme (T1 = temps initial)
23     call system_clock(count=t1, count_rate=ir)
24
25     ! calcul de PI
26     Pi_calcule = 0.0_16
27     do i=rang+1, n
28       x = h * ( real(i,kind=16) - 0.5_16 )
29       Pi_calcule = Pi_calcule + f(x)
30     end do
31     Pi_calcule = h * Pi_calcule
32
33     ! calcul du temps de l'execution du programme (T2 = temps final)
34     call system_clock(count=t2, count_rate=ir)
35     temps=real(t2-t1)/real(ir)
36
37     ! Ecart entre la valeur reelle et la valeur calculee de Pi.
38     ecart = abs(Pi_reel - Pi_calcule)
39
40     ! Impression du resultat.
41     write (*,*) "Nombre d intervalles : ",n
42     write (*,*) "|Pi_reel-Pi_calcule| : ",ecart
43     write (*,*) "Temps reel           : ",temps
44
45 end program pi
```

```
$ ifort pi.f90
$ a.out
 Nombre d intervalles :     100000000
 |Pi_reel-Pi_calcule| :     8.742214685497629E-008
 Temps reel           :      16.00360
```

Exercice 2 : Produit vecteur matrice

On se propose de faire le produit d'un vecteur V et une matrice A tel que :

$$V_i = \begin{pmatrix} V_1, & V_2, & V_3, & V_4 \end{pmatrix}; \quad A_{ij} = \begin{pmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{pmatrix}$$

$$\begin{aligned} B(1,j) &= V(1,i) \times A(i,j) \\ &= \begin{pmatrix} V_1 \times A_{11} + V_2 \times A_{21} + V_3 \times A_{31} + V_4 \times A_{41} \\ V_1 \times A_{12} + V_2 \times A_{22} + V_2 \times A_{32} + V_2 \times A_{42} \\ V_1 \times A_{13} + V_3 \times A_{23} + V_3 \times A_{33} + V_3 \times A_{43} \\ V_1 \times A_{14} + V_4 \times A_{24} + V_4 \times A_{34} + V_4 \times A_{44} \end{pmatrix} \end{aligned}$$

1) Allouer et initialiser la matrice A et le vecteur B sur le processus 0

2) Chaque processus doit calculer une valeur du vecteur B

3) Regrouper toutes les valeurs calculées dans le vecteur B sur le processus 0

```
1  program vecteur_matrice
2      integer, parameter      :: M=4, N=4
3      integer                 :: i, j, x
4      integer, dimension(M)   :: V=0
5      integer, dimension(:,:), allocatable :: A
6      integer, dimension(:), allocatable :: B
7
8      V=(/1,2,3,4/)
9      allocate(A(M,N))
10     allocate(B(M))
11     A=1
12     ! affichage la matrice A
13     write(*,*) "La matrice A : "
14     do i=1,M
15         write(*,*) A(i,:)
16     enddo
17     ! affichage du vecteur V
18     write(*,*) "Vecteur V : "
19     write(*,*)  V
20
21     ! calcul de B
22     do j=1,N
23         B(j)=0
24         do i=1,M
25             B(j)=V(i)*A(i,j)+B(j)
26         end do
27     end do
28
29     ! affichage le vecteur B
30     write(*,*) "B = V * A :"
31     write(*,*) B
32 end program vecteur_matrice
```

```
$ ifort produit_vacteur_matrice.f90
$ a.out
 La matrice A :
           1           1           1           1
           1           1           1           1
           1           1           1           1
           1           1           1           1
 Vecteur V :
           1           2           3           4
 B = V * A :
          10          10          10          10
```

Exercice 3 : `MPI_ALLGATHERV`
Réécrire l'exemple 8 en emplaçant `MPI_GATHERV` par `MPI_ALLGATHERV`

Exercice 4 : Diagonale d'une matrice
Soit les matrices A_{ij} et B_{ij} de taille MxN, tel que la diagonale de B_{ij} est égale à la diagonale de A_{ij}. Pour faire simple, M=N

$$A_{ij} = \begin{pmatrix} A_{11} & 1 & 1 & ... & 1 \\ 1 & A_{22} & 1 & ... & . \\ 1 & 1 & A_{33} & ... & . \\ 1 & 1 & 1 & ... & A_{MM} \end{pmatrix} ; \quad B_{ij} = \begin{pmatrix} A_{11} & 0 & 0 & ... & 0 \\ 0 & A_{22} & 0 & ... & . \\ 0 & 0 & A_{33} & ... & . \\ 0 & 0 & 0 & ... & A_{MM} \end{pmatrix}$$

- Ecrire un code parallélisé en MPI pour que la diagonale de A soit égale à la diagonale de B en utilisant les types dérivés.

Programme séquentiel :

```
1  program diagonale
2
3    implicit none
4    integer, parameter                      :: M=5, N=5
5    integer, dimension (:,:), allocatable :: A
6    integer, dimension (:,:), allocatable :: B
7    integer                :: t1, t2, ir, i
8    real                   :: temps
9
10
11   ! Temps initial
12   call system_clock(count=t1, count_rate=ir)
13
14   ! Allocation et initialisation de A et B
15   allocate(A(M,N))
16   allocate(B(M,N))
17   A(:,:)=1
18   B(:,:)=0
19
20   ! diag A = diag B
21   do i=1, M
22     B(i,i)=A(i,i)
23   end do
24
25   ! Temps final
26   call system_clock(count=t2, count_rate=ir)
27   temps=real(t2-t1)/real(ir)
28
29   write (*,*) "Temps reel : ", temps
30
31 end program diagonale
```

```
$ ifort diagonale_seq.f90
$ a.out
 Temps reel :     9.045900
```

Exercice 5 : Transposé d'une matrice

Dans cet exercice, on se propose de se familiariser avec les types dérivés On se donne une matrice A(5x4) sur le processus 0.

Il s'agit pour le processus 0 d'envoyer au processus 1 cette matrice mais d'en faire automatiquement la transposition au cours de l'envoi.

Pour ce faire, on va devoir se construire deux types dérivés, un type `type_ligne` et un type `type_transpose`

$$A = \begin{pmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \\ A_{51} & A_{52} & A_{53} & A_{54} \end{pmatrix} \longrightarrow A^T \begin{pmatrix} A_{11} & A_{21} & A_{31} & A_{41} & A_{51} \\ A_{12} & A_{22} & A_{32} & A_{42} & A_{52} \\ A_{13} & A_{23} & A_{33} & A_{43} & A_{53} \\ A_{14} & A_{24} & A_{34} & A_{44} & A_{54} \end{pmatrix}$$

Solutions

Exercice 1 : Calcul de π

```
program pi
    !                    _
    !  But : calcul de  || par la methode des rectangles (point milieu).
    !
    !                     / 1
    !                     |       4              _
    !                     |   ----------   dx = ||
    !                     |    1 + x**2
    !                    / 0
    use mpi
    implicit none

    integer, parameter :: n=100000000
    double precision   :: f, x, a, h
    double precision   :: Pi_calcule, Pi_final, Pi_reel, ecart
    integer            :: i
    double precision   :: t1, t2, temps
    integer            :: code, rang, nbr_procs

    ! Fonction instruction a integrer
    f(a) = 4.0_16 / ( 1.0_16 + a*a )

    ! valeur reel de Pi avec 15 chiffres apres la virgule
    Pi_reel = 3.141592653589793

    ! Longueur de l'intervalle d'integration
    h = 1.0_16 / real(n,kind=16)

    ! debut de la region parallele
    call MPI_INIT(code)
    call MPI_Comm_size(MPI_COMM_WORLD,nbr_procs, code)
    call MPI_Comm_rank(MPI_COMM_WORLD,rang, code)

    ! calcul du temps de l'execution du programme (T1 = temps initial)
    t1=MPI_WTIME()

    Pi_calcule = 0.0_16
    do i=rang+1, n, nbr_procs
      x = h * ( real(i,kind=16) - 0.5_16 )
      Pi_calcule = Pi_calcule + f(x)
    end do
    Pi_calcule = h * Pi_calcule

    call MPI_Reduce(Pi_calcule, Pi_final, 1, MPI_DOUBLE_PRECISION, MPI_SUM,
        0, MPI_COMM_WORLD, code);

    ! calcul du temps de l'execution du programme (T2 = temps final)
    t2=MPI_WTIME()
    temps = t2-t1

    ! Ecart entre la valeur reelle et la valeur calculee de Pi.
    ecart = abs(Pi_reel - Pi_final)

    if (rang==0) then
        write (*,*) "Nombre de processus     : ",nbr_procs
        write (*,*) "Nombre d intervalles    : ",n
        write (*,*) "| Pi_reel - Pi_final |  : ",ecart
        write (*,*) "Temps reel              : ",temps
    endif

    ! fin de la region parallele
    call MPI_Finalize(code)

end program pi
```

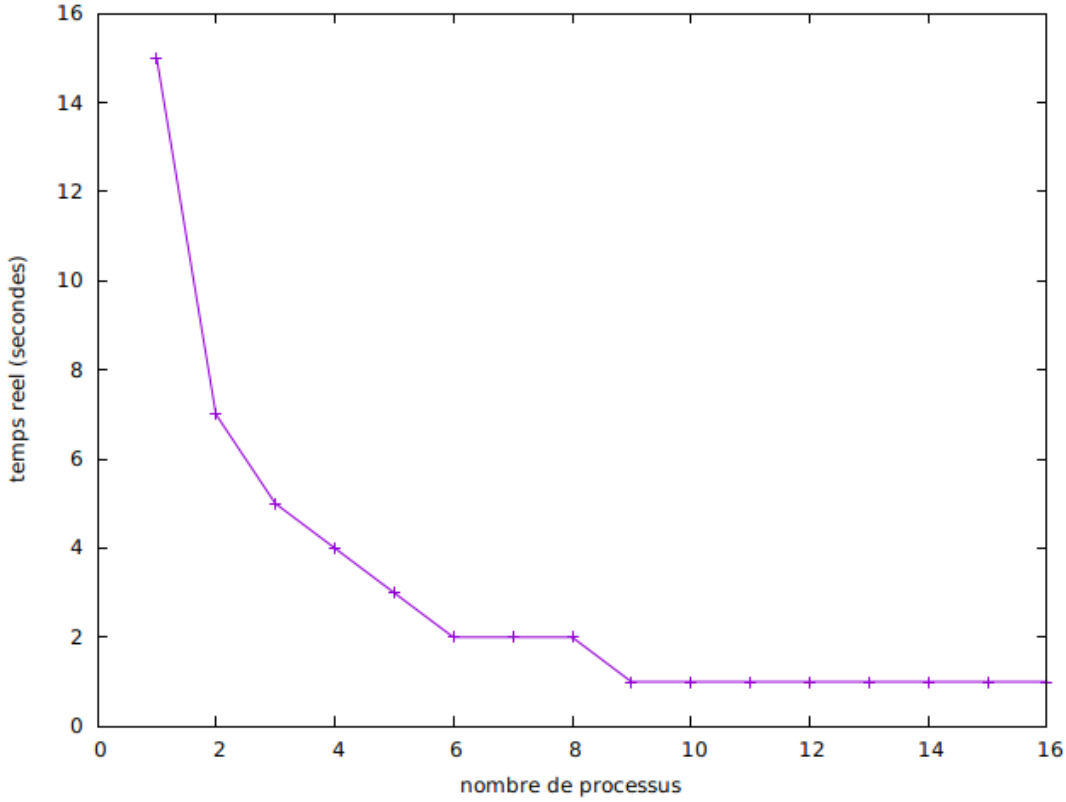


FIGURE 1 – Temps de l'exécution du programme en fonction du nombre de processus

Exercice 2 : Produit vecteur matrice

```
1  program vecteur_matrice
2
3      use MPI
4
5      integer, parameter       :: M=4, N=4
6      integer                  :: i, j, x
7      integer, dimension(M)    :: V=0
8      integer, dimension(:,:), allocatable:: A
9      integer, dimension(:), allocatable  :: B
10     integer, dimension (N)              :: A_ligne
11     integer                             :: B_colonne=0
12     integer                             :: code,rang
13
14     call MPI_INIT(code)
15     call MPI_COMM_RANK(MPI_COMM_WORLD, rang, code)
16
17     V=(/1,2,3,4/)
18     if (rang==0) then
19         allocate(A(M,N))
20         allocate(B(N))
21         A=1
22
23         ! affichage la matrice A
24         write(*,*) "La matrice A : "
25         do i=1,M
26                 write(*,*) A(i,:)
27         enddo
28
29         ! affichage le vecteur V
30         write(*,*) "Vecteur V : "
31         write (*,*) V
32
33     end if
34
35     call MPI_SCATTER(A, M, MPI_INTEGER, A_ligne, M, MPI_INTEGER, 0,
           MPI_COMM_WORLD, code)
36
37     do i=1,M
38         B_colonne=V(i)*A_ligne(i)+B_colonne
39     end do
40
41     call MPI_GATHER(B_colonne, 1, MPI_INTEGER, B, 1, MPI_INTEGER, 0,
           MPI_COMM_WORLD, code)
42
43     if (rang==0) then
44         ! affichage le vecteur B
45         write(*,*) "B = V * A :"
46         write(*,*) B
47     endif
48
49     call MPI_FINALIZE(code)
50
51 end program vecteur_matrice
```

```
$ ifort produit_vacteur_matrice.f90
$ a.out
 La matrice A :
           1           1           1           1
           1           1           1           1
           1           1           1           1
           1           1           1           1
 Vecteur V :
           1           2           3           4
 B = V * A :
          10          10          10          10
```

Exercice 3 : MPI_ALLGATHERV

Réécrire l'exemple 8 en remplaçant `MPI_GATHER` par `MPI_ALLGATHERV`

```
program allgatherv

  use MPI
  implicit none
  integer                                  :: x, nbr_procs, rang
  integer                                  :: code,i
  integer, dimension(:), allocatable       :: vecteur, nb_elements_recus,
      deplacement
  integer, dimension(:,:), allocatable     :: matrice

  call MPI_INIT(code)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, nbr_procs, code)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rang, code)

  ! allouer et initialiser "vecteur" sur tous les processus
  allocate(vecteur(rang+1))
  x=1
  do i=1,rang+1
        vecteur(i)=x
        x=x+1
  enddo
  ! afficher la valeur de chaque vecteur sur les differents processus
  write(*,*) rang, ":", vecteur

  ! alouer : matrice, nb_elements_recus et deplacement
  allocate(matrice(nbr_procs,nbr_procs))
  allocate(nb_elements_recus(nbr_procs))
  allocate(deplacement(nbr_procs))

  ! initialiser : matrice, nb_elements_recus et deplacement
  matrice(:,:)=0
  do i=1,nbr_procs
    nb_elements_recus(i)=i
  enddo
  deplacement(1)=0
  do i=2,nbr_procs
    deplacement(i)=deplacement(i-1)+nbr_procs
  enddo

  ! regrouper les vecteurs dans la mtrice sur tous les processus
  call MPI_ALLGATHERV(vecteur, rang+1, MPI_INTEGER, matrice,
      nb_elements_recus, deplacement,MPI_INTEGER, &
    MPI_COMM_WORLD,code)

  call sleep(1)
  ! afficher la matrice
  write(*,*) "processus :", rang
  do i=1,nbr_procs
    write(*,*) matrice(i,:)
  enddo
  write(*,*)

  call MPI_FINALIZE(code)

end program allgatherv
```

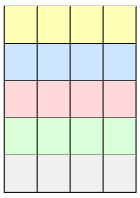
Exercice 4 : Diagonale d'une matrice

```
1  program diagonale
2
3    use mpi
4    implicit none
5    integer, parameter                      :: M=5, N=5
6    integer, dimension (:,:), allocatable :: A
7    integer, dimension (:,:), allocatable :: B
8    integer                                 :: code, nbr_procs, rang
9    integer, dimension(MPI_STATUS_SIZE)   :: statut
10   integer                                 :: type_diagonale
11   double precision                        :: t1, t2, temps
12
13   call MPI_INIT(code)
14   call MPI_COMM_SIZE(MPI_COMM_WORLD,nbr_procs,code)
15   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
16
17   call MPI_TYPE_VECTOR(N,1,M+1, MPI_INTEGER, type_diagonale, code)
18   call MPI_TYPE_COMMIT (type_diagonale, code)
19
20   t1=MPI_WTIME()
21   if (rang==0) then
22     allocate(A(M,N))
23     A(:,:)=1
24     call MPI_SEND (A(1,1), 1, type_diagonale,1, 100, MPI_COMM_WORLD,code)
25   else if (rang==1) then
26     allocate(B(M,N))
27     B(:,:)=0
28     call MPI_RECV (B(1,1), 1, type_diagonale,0, 100,MPI_COMM_WORLD, statut,
         code)
29   end if
30     t2=MPI_WTIME()
31     temps = t2-t1
32
33     write (*,*) "Temps reel du proc : ", rang, temps
34
35     call MPI_TYPE_FREE(type_diagonale, code)
36     call MPI_FINALIZE(code)
37
38 end program diagonale
```

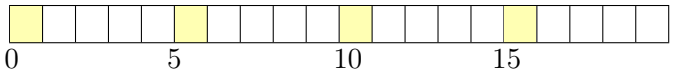
```
$ mpif90 diagonale.SOLUTION.f90
$ mpirun -n 2 a.out
 Temps reel du proc :            0   4.29847407341003
 Temps reel du proc :            1   4.29847311973572
```

Exercice 5 : Transposé d'une matrice

Pour créer le type dérivé `type_ligne` il faut utiliser `MPI_TYPE_VECTOR` :



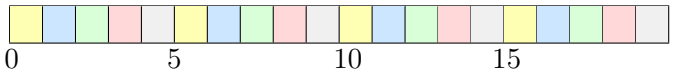
Position en mémoire de la première ligne :



`nb_elements`	:	nombre d'éléments dans une ligne : `N=5`
`nb_lignes`	:	1
`deplacement`	:	nombre d'élément dans une colonne : `M=4`
`ancien_type`	:	`MPI_INTEGER`
`nouveau_type`	:	`type_ligne`
`code`	:	`code`

```
MPI_TYPE_VECTOR (N, 1, M, MPI_INTEGER, type_ligne, code)
```

On regroupe ensuite toutes les lignes avec `MPI_TYPE_CREATE_HVECTOR`



`nb_blocs`	:	`4=M`
`taille_blocs`	:	1
`deplacement`	:	1 * taille_integer (écart entre le début de chaque bloc)
`ancien_type`	:	`ancien_type`
`nouveau_type`	:	`nouveau_type`
`code`	:	`code`

```
MPI_TYPE_CREATE_HVECTOR (M, 1, deplacement, type_ligne, nouveau_type, code)
```

$$A_{ij}^{P0} = \begin{pmatrix} 1 & 6 & 11 & 16 \\ 2 & 7 & 12 & 17 \\ 3 & 8 & 13 & 18 \\ 4 & 9 & 14 & 19 \\ 5 & 10 & 15 & 20 \end{pmatrix} \qquad A_{ij}^{P1} = \begin{pmatrix} 1 & 5 & 9 & 13 & 17 \\ 2 & 6 & 10 & 14 & 18 \\ 3 & 7 & 11 & 15 & 19 \\ 4 & 8 & 12 & 16 & 20 \end{pmatrix}$$

`MPI_SEND nouveau_type` $\longrightarrow$ `MPI_RECV nouveau_type`

$$A_{ij}^{P0} = \begin{pmatrix} 1 & 6 & 11 & 16 \\ 2 & 7 & 12 & 17 \\ 3 & 8 & 13 & 18 \\ 4 & 9 & 14 & 19 \\ 5 & 10 & 15 & 20 \end{pmatrix} \qquad A_{ij}^{P2} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \end{pmatrix}$$

`MPI_SEND nouveau_type` $\longrightarrow$ `MPI_RECV MPI_INTEGER`

```
program transpose

        use MPI
        implicit none
        integer                             :: i, j, b
        integer, parameter                  :: M=5, N=4
        integer, dimension (M,N)            :: A
        integer, dimension (N,M)            :: At
        integer                             :: nbr_procs, rang, code, statut,
            taille_integer
        integer                             :: type_ligne, type_transpose
        integer(kind=MPI_ADDRESS_KIND)      :: deplacement


        call MPI_INIT(code)
        call MPI_COMM_SIZE(MPI_COMM_WORLD,nbr_procs,code)
        call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)

        At(:,:)= 0

        call MPI_TYPE_SIZE(MPI_INTEGER, taille_integer, code)
        call MPI_TYPE_VECTOR(N, 1, M, MPI_INTEGER, type_ligne, code)

        if (rang==0) then
                write (*,*) "un entier fait : ",taille_integer, "octets"
        end if
        deplacement=1*taille_integer
        call MPI_TYPE_CREATE_HVECTOR(M, 1, deplacement, type_ligne,
            type_transpose, code)

        call MPI_TYPE_COMMIT(type_transpose, code)

        if (rang==0) then
        b=0
        do j=1,N
                do i=1,M
                        b=1+b
                        A(i,j)=b
                enddo
        enddo
        do i=1, M
                write (*,*) A(i,:)
        end do
        write (*,*)

        call MPI_SEND(A, 1, type_transpose, 1, 100, MPI_COMM_WORLD, code)

        elseif (rang==1) then
        call MPI_RECV(At, N*M, MPI_INTEGER, 0, 100, MPI_COMM_WORLD, statut,
             code)
        do i=1,N
                print *, At(i,:)
        end do
        endif

        call MPI_TYPE_FREE(type_transpose, code)
        call MPI_FINALIZE(code)

end program transpose
```