La programmation parallèle

~~~~~

OpenMP

~~~

MPI

Aurélia Marchand
Aurelia.Marchand@obspm.fr

25 janvier 2018

Plan

-1- Introduction

Le parallélisme est la conséquence :

- besoin des applications
  - calcul scientifique
  - traitement d'images
  - bases de données

qui demandent des ressources en CPU et en temps de calcul de plus en plus importantes

- limites des architectures séquentielles
  - performance
  - capacité d'accès à la mémoire
  - tolérance aux pannes

Définition :

Les ordinateurs parallèles sont des machines qui comportent une architecture parallèle, constituée de plusieurs processeurs identiques, ou non, qui concourent au traitement d'une application.

La performance d'une architecture parallèle est la combinaison des performances de ses ressources et de leur agencement. (latence, débit)

Architectures parallèles :

=> augmentation de la taille mémoire

=> augmentation du nombre de processeurs

=> accélération des calculs complexes ou coûteux en temps d'occupation CPU

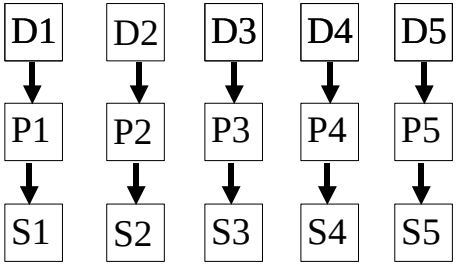
(calcul matriciel, simulation numérique, transformée de fourier...)

=> calcul répétitif sur un large ensemble de données structuré

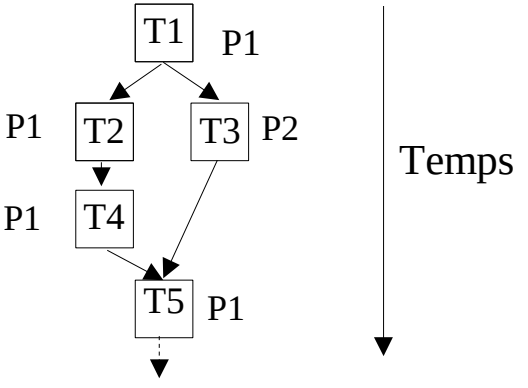
=> traitement indépendant

On peut avoir différentes sources de parallélisme :

- le parallélisme de données : la même opération est effectuée par chaque processeur sur des données différentes.



- Parallélisme de contrôle : des opérations sont réalisées simultanément sur plusieurs processeurs. Le programme présente des séquences d'opérations indépendantes qui peuvent être exécutées en parallèle.



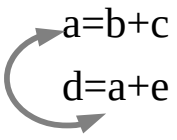
- Parallélisme de flux : Les opérations sur un même flux de données peuvent être enchaînées (pipeline)

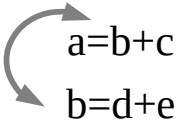
temps	Tâche 1	Tâche 2	Tâche 3	Tâche 4	Tâche 5
t1	Donnée 1				
t2	D2	D1			
t3	D3	D2	D1		
t4	D4	D3	D2	D1	
t5	D5	D4	D3	D2	D1
t6	D6	D5	D4	D3	D2

limites du parallélisme :

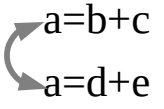
- les dépendances de données :

Dépendance de flot





Dépendance de sortie



=> les instructions doivent être indépendantes :

- exécuter dans un ordre quelconque
- simultanément

- les dépendances de contrôle :

I1	a=b+c;	I1 et I4 sont à priori indépendantes
I2	if (a<0)	mais selon la valeur de la variable 'a'
I3	{d=e+f;}	I3 peut être exécuté ou non
I4	g=d+h;	entraînant une dépendance entre I3 et I4

- les dépendances de ressources :

  nombre insuffisant de processeurs pour effectuer les instructions en parallèle alors qu'elles sont indépendantes les unes des autres.

- rapport temps de communication / temps de calcul :

Il n'est pas toujours avantageux de paralléliser une application. Les communications peuvent dans certain cas augmenter le temps d'exécution.

Mesure de performance :

débit de traitement : P= nb_op/tps

accélération : $a=t_{séquentiel}/t_{parallèle}$

efficacité : e=a/nb_proc

-2- Les architectures parallèles

-2.1- Les types d'architectures parallèles

Classification des architectures parallèles :

	1 flux d'instruction	> 1 flux d'instruction
1 flux de données	SISD	MISD (pipeline)
> 1 flux de données	SIMD	MIMD

S : Single, M : Multiple

I : Instruction, D : Data

SISD : une instruction - une donnée

machine séquentielle (modèle de Von Neumann)

SIMD : plusieurs données traitées en même temps par une seule instruction.

Utilisé dans les gros ordinateurs vectoriels.

Première machine parallèle : l'ILLIAC IV (1966)

MISD : Une donnée unique traitée par plusieurs instructions.

Architecture pipeline.

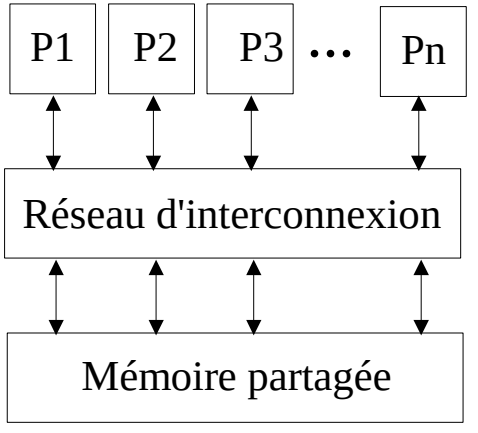
MIMD : exécution d'une instruction différente sur chaque processeurs pour des données différentes.

Pour simplifier la programmation, on exécute la même application sur tous les processeurs. Ce mode d'exécution est appelé : SPMD (Single Program Multiple Data)

Il existe deux types de machine parallèle :

- les machines parallèles à mémoire partagée
- les machines parallèles à mémoire distribuée

-2.2- Les machines à mémoire partagée



Caractéristiques :

- plusieurs processeurs avec des horloges indépendantes
- une seule mémoire commune à tous les processeurs
- programmable par le standard portable OpenMP

Les machines à mémoire partagée permettent de réaliser le parallélisme de données et de contrôle.

Le programmeur n'a pas besoin de spécifier la distribution des données sur chaque processeur. Il définit seulement la partie du programme qui doit être parallélisée (directives) et doit gérer les synchronisations.

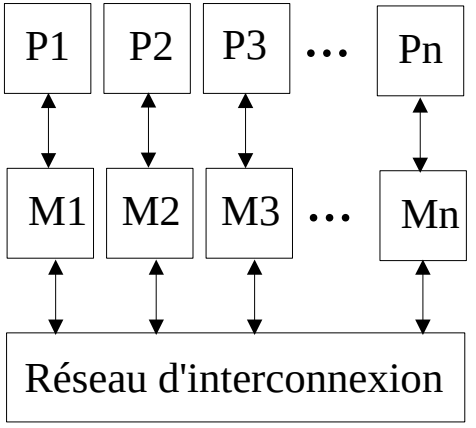
Avantages :

- simplicité d'utilisation
- portabilité au SMP (architecture homogène à mémoire partagée)
- scalabilité
- parallélisation de haut niveau

Inconvénients :

- coût
- limitation du nombre de processeurs (conflit d'accès au niveau matériel)
- la bande passante du réseau est le facteur limitant de ces architectures

-2.3- Les machines à mémoire distribuée



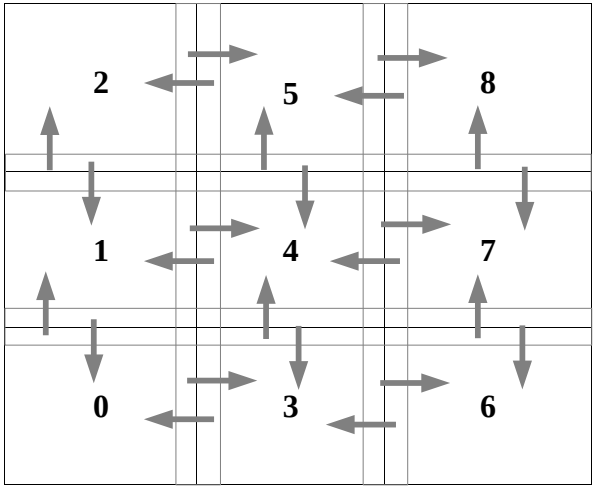
Caractéristiques :

- interconnexion (réseau local rapide) de systèmes indépendants (noeuds)
- mémoire propre locale à chaque processeur
- chaque processeur exécute des instructions identiques ou non, sur des données identiques ou non
- Parallélisme par échange de messages

Les communications entre processeurs sont réalisés par l'appel à des fonctions de bibliothèque standard, par exemple MPI (Message Passing Interface) qui est stable et qui est portable sur les machines parallèles actuelles.

MPI => MPP (massively Parallel Processing)

La décomposition de domaine permet d'accélérer le calcul :



Avantages :

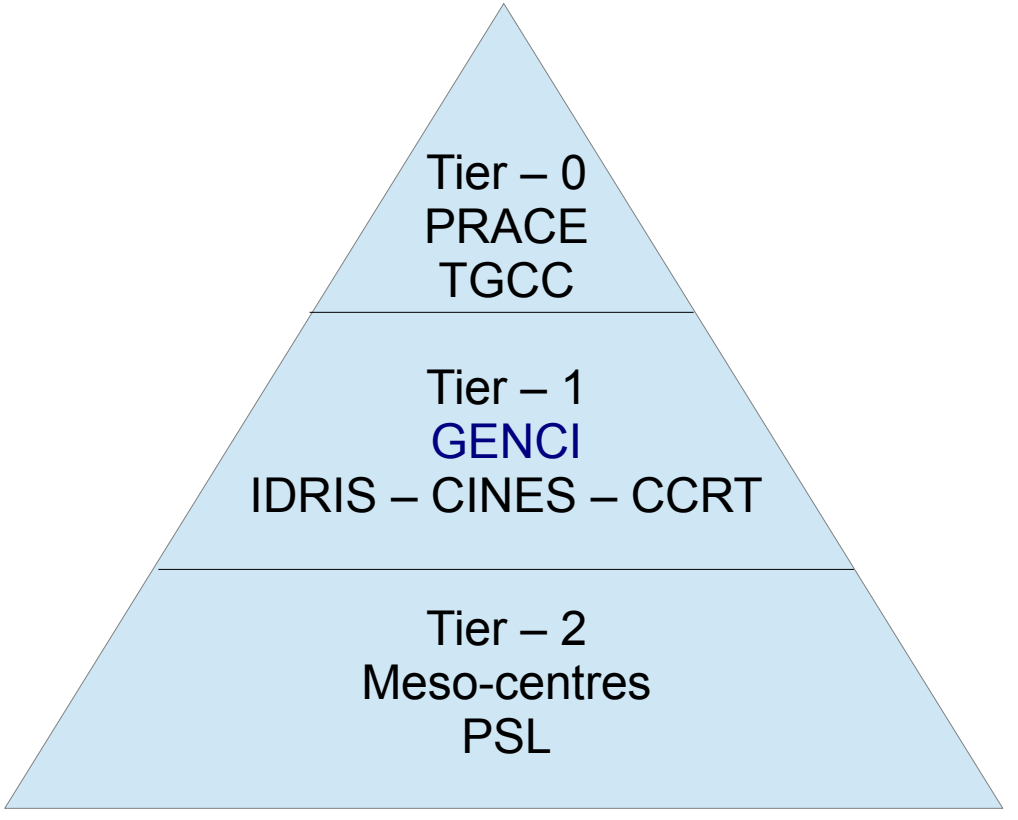
- non limité théoriquement en nombre de processeurs
  => grande puissance de calcul
- réalisation aisée par la conception de cluster
  - mise en réseau d'un certain nombre de machines
  - système d'exploitation Linux
- partage transparent des ressources
- portabilité des applications MPI

Inconvénients :

- plus difficile à programmer que les machines à mémoire partagée
  - MPI (Message Passing Interface)
- efficacité limitée
- administration du cluster

-3- Les machines parallèles de l'observatoire

- MesoPSL1 (2011)
  - 216 cœurs – 3.06 GHz – 2 Go
  - 29 To Lustre
- MesoPSL (2012)
  - 2128 cœurs – 2.6 GHz – 4 Go
  - 128 To
  - Infinband
- MesoPSL.SMP (2018)
  - 12*96 cœurs – 768 Go
  - 400 To
  - Infiniband
- Tycho
  - 1040 cœurs – 4 Go



-4-OpenMP

-4.1-Historique

OpenMP (Open Multi Processing) est adopté comme standard en 1997 pour le calcul scientifique.

2000 : OpenMP-2 permet le parallélisation de certaines constructions Fortran 95.

2008 : OpenMP-3 introduit le concept de tâche.

2013 : OpenMP-4 gère la dépendance entre tâches, le placement de threads

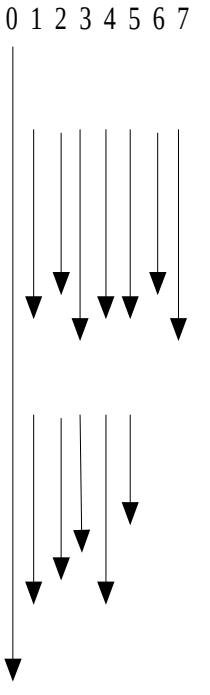
-4.2-Définition

OpenMP offre une interface standard de haut niveau pour une programmation parallèle de type SIMD sur machine à mémoire partagée.

Une application OpenMP est exécutée par un processus unique, ce processus active des processus légers (threads) dans les régions parallèles du programme.

Ces processus léger exécutent des instructions et ont accès à des variables qui peuvent être locales (privées) ou partagées.

Un programme OpenMP est une succession de régions séquentielles (tâche maître de rang 0) et parallèles.



Les régions parallèles peuvent être une répartition :

- de tâches différentes
- de tâches identiques (boucle)
- d'appel d'une même procédure

Le nombre de tâches peut varier au cours du programme.

Il peut être nécessaire de synchroniser les tâches au cours de l’exécution

Compilation : ifort -qopenmp

Nombre de tâches : export OMP_NUM_THREADS=8

Le développeur définit les régions qui doivent être parallélisées par l'ajout de directives dans le programme.

Syntaxe d'une directive :

```
sentinelle directive [clause [clause]...]
```

Fortran :

```
use OMP_LIB
…
!$OMP PARALLEL
…
!$OMP END PARALLEL
```

C et C++ :

```
#include <omp.h>
…
#pragma omp parallel
{…}
```

-4.3-Région parallèle

- La région parallèle s'applique au code contenu entre les directives « PARALLEL » et « END PARALLEL », elle s'étend au code des sous-programmes appelés.

```
program parallel
!$OMP PARALLEL
call sub()
!$OMP END PARALLEL
end program parallel

subroutine sub()
use OMP_LIB
logical :: p
integer :: i
!$ i=OMP_GET_THREAD_NUM()
!$ p = OMP_IN_PARALLEL()
!$ print *,i," dans region parallele ? ", p
end subroutine sub
```

```
> ifort -qopenmp prog.f90
> export OMP_NUM_THREADS=4
> ./a.out
 0 dans region parallele ? T
 2 dans region parallele ? T
 1 dans region parallele ? T
 3 dans region parallele ? T
```

- Le nombre de threads est définit par la variable d'environnement OMP_NUM_THREADS mais elle peut être modifiée dans le programme en utilisant la clause NUM_THREADS

```
program parallel
implicit none

!$OMP PARALLEL NUM_THREADS(3)
print *,"Premiere region parallele"
!$OMP END PARALLEL

!$OMP PARALLEL NUM_THREADS(2)
print *,"Deuxieme region parallele"
!$OMP END PARALLEL

end program parallel
```

```
> ifort -qopenmp prog.f90
> export OMP_NUM_THREADS=4
> export OMP_DYNAMIC =true
> ./a.out
 Premiere region parallele
 Premiere region parallele
 Premiere region parallele
 Deuxieme region parallele
 Deuxieme region parallele
```

- Toutes les tâches d'une même région exécutent le même code.
- Une synchronisation implicite est faite à la fin de la région parallèle.
- Il est interdit d'effectuer des branchements (cycle, goto...) de ou vers une région parallèle.
- Les variables sont partagées par défaut, on peut imposer qu'elles soient privées en utilisant la clause DEFAULT. Une variable privée est indéfinie à l'entrée de la région parallèle, sauf si on précise que la variable doit être initialisée avec la dernière valeur connue de cette variable.

```
program parallel
integer i
i=10
!$OMP PARALLEL
        i=i+5
        print *,'i=',i
!$OMP END PARALLEL
end program parallel
```

```
> ifort -qopenmp prog.f90
> export OMP_NUM_THREADS=4
> ./a.out
 i=           15
 i=           20
 i=           30
 i=           25
```

```
program parallel
integer i
i=10
!$OMP PARALLEL DEFAULT(PRIVATE)
        i=i+5
        print *,'i=',i
!$OMP END PARALLEL
end program parallel
```

```
> ifort -qopenmp prog.f90
> export OMP_NUM_THREADS=4
> ./a.out
 i=           5
 i=           5
 i=           5
 i=           5
```

```
program parallel
integer i
i=10
!$OMP PARALLEL DEFAULT(NONE) FIRSTPRIVATE(i)
        i=i+5
        print *,'i=',i
!$OMP END PARALLEL
end program parallel
```

```
> ifort -qopenmp prog.f90
> export OMP_NUM_THREADS=4
> ./a.out
 i=           15
 i=           15
 i=           15
 i=           15
```

- Les variables sont privées par défaut dans les sous-programmes. Les variables passées en argument conservent le statut définit dans la région parallèle.

```
program parallel
implicit none
integer i,j
i=10
j=100
!$OMP PARALLEL PRIVATE(i) SHARED(j)
call sub(i,j)
print *,'i=',i,' ; j=',j
!$OMP END PARALLEL
print *,'i=',i,' ; j=',j
end program parallel

subroutine sub(x,y)
use OMP_LIB
implicit none
integer :: x, y
print *,'x=',x,' ; y=',y
x = y + OMP_GET_THREAD_NUM()
print *,'x=',x,' ; y=',y
end subroutine sub
```

```
> ifort -qopenmp prog.f90
> export OMP_NUM_THREADS=4
> ./a.out
 x=           0  ; y=         100
 x=           0  ; y=         100
 x=         101  ; y=         100
 x=         100  ; y=         100
 x=           0  ; y=         100
 i=         100  ; j=         100
 x=         103  ; y=         100
 i=         101  ; j=         100
 i=         103  ; j=         100
 x=           0  ; y=         100
 x=         102  ; y=         100
 i=         102  ; j=         100
 i=          10  ; j=         100
```

- Les variables statiques (COMMON, MODULE, PARAMETER, DATA...) sont des variables partagées, mais elles peuvent être privatisé en utilisant la directive THREADPRIVATE

```
program parallel
use OMP_LIB
implicit none
integer :: i
common/MonCommon/i
i = 10
!$OMP PARALLEL
i = i + OMP_GET_THREAD_NUM()
call sub()
!$OMP END PARALLEL
print*,"i = ",i
end program parallel

subroutine sub()
implicit none
integer :: i, j
common/MonCommon/i
j = i + 20
print *,"j = ",j
end subroutine sub
```

```
> ifort -qopenmp prog.f90
> export OMP_NUM_THREADS=4
> ./a.out
 j =           30
 j =           32
 j =           36
 j =           35
 i =           16
```

```
program parallel
use OMP_LIB
implicit none
integer :: i
common/MonCommon/i
!$OMP THREADPRIVATE(/MonCommon/)
i = 10
!$OMP PARALLEL COPYIN(/MonCommon/)
i = i + OMP_GET_THREAD_NUM()
call sub()
!$OMP END PARALLEL
print*,"i = ",i
end program parallel

subroutine sub()
implicit none
integer :: i, j
common/MonCommon/i
!$OMP THREADPRIVATE(/MonCommon/)
j = i + 20
print *,"j = ",j
end subroutine sub
```

```
> ifort -qopenmp prog.f90
> export OMP_NUM_THREADS=4
> ./a.out
 j =           32
 j =           33
 j =           30
 j =           31
 i =           10
```

-4.4-Sections parallèles

- Une section définie par la directive SECTION est une partie du code exécutée par un seul thread.

```
program parallel
use OMP_LIB
implicit none
integer :: i=-1, j=-1, k=-1, l=-1
integer :: rang
!$OMP PARALLEL FIRSTPRIVATE(rang,i,j,k,l)
rang=OMP_GET_THREAD_NUM()
  !$OMP SECTIONS
        !$OMP SECTION
                i=rang
        !$OMP SECTION
                j=rang
        !$OMP SECTION
                k=rang
        !$OMP SECTION
                l=rang
  !$OMP END SECTIONS
print *,rang,i,j,k,l
!$OMP END PARALLEL
end program parallel
```

```
> ifort -qopenmp prog.f90
> export OMP_NUM_THREADS=4
> ./a.out
 1 -1  1 -1 -1
 2 -1 -1  2 -1
 0  0 -1 -1 -1
 3 -1 -1 -1  3
> export OMP_NUM_THREADS=2
> ./a.out
 1 -1 -1  1  1
 0  0  0 -1 -1
```

-4.5-Boucles parallèles

- La directive DO permet de paralléliser une boucle.
- Les boucles conditionnelles ou infinies ne sont pas parallélisable.
- Les indices de la boucle sont des variables privées.
- Une synchronisation est effectuée lors du END DO sauf si la clause NOWAIT précise que les threads doivent continuer l'exécution du programme.
- Les clauses DO peuvent être imbriquées, il faut toujours commencer par la boucle la plus externe.

```
program parallel
use OMP_LIB
implicit none
integer, parameter :: n=100
integer, dimension(n) :: a
integer :: i, i_min, i_max, r, nb_taches
!$OMP PARALLEL PRIVATE(r,nb_taches,i_min,i_max)
r= OMP_GET_THREAD_NUM()
nb_taches= OMP_GET_NUM_THREADS()
i_min=n ; i_max=0
!$OMP DO
do i = 1, n
  a(i)=10+i
  i_min=min(i_min,i)
  i_max=max(i_max,i)
end do
!$OMP END DO NOWAIT
print *,"Rang : ",r,"; min :",i_min,"; max :",i_max
!$OMP END PARALLEL
end program parallel
```

```
> ifort -qopenmp prog.f90
> export OMP_NUM_THREADS=4
> ./a.out
 Rang : 2 ; min : 51 ; max : 75
 Rang : 1 ; min : 26 ; max : 50
 Rang : 3 ; min : 76 ; max : 100
 Rang : 0 ; min :  1 ; max : 25
```

- La clause SCHEDULE permet de répartir la boucle entre les threads

```
program parallel
use OMP_LIB
implicit none
integer, parameter :: n=100000
integer, dimension(n) :: a
integer :: i, i_min, i_max, r
!$OMP PARALLEL &
!$OMP PRIVATE(r,i_min,i_max)
r= OMP_GET_THREAD_NUM()
i_min=n ; i_max=0
!$OMP DO SCHEDULE(RUNTIME)
do i = 1, n
  a(i)=10+i
  i_min=min(i_min,i)
  i_max=max(i_max,i)
end do
!$OMP END DO NOWAIT
print *,"Rang : ",r," ; &
 min : ",i_min," ; max : ",i_max
!$OMP END PARALLEL
end program parallel
```

```
> ifort -qopenmp prog.f90
> export OMP_NUM_THREADS=4
> export OMP_SCHEDULE="STATIC,25000"
> ./a.out
 Rang : 3 ; min : 75001 ; max : 100000
 Rang : 0 ; min :     1 ; max :  25000
 Rang : 2 ; min : 50001 ; max :  75000
 Rang : 1 ; min : 25001 ; max :  50000
> export OMP_SCHEDULE="DYNAMIC,1000"
> ./a.out
 Rang : 2 ; min :     1 ; max : 100000
 Rang : 3 ; min :  3001 ; max :  98000
 Rang : 0 ; min :  1001 ; max :  97000
 Rang : 1 ; min : 11001 ; max :  99000
> export OMP_SCHEDULE="GUIDED,1000"
> ./a.out
 Rang : 3 ; min : 23438 ; max : 100000
 Rang : 0 ; min :     1 ; max :  96088
 Rang : 2 ; min : 33008 ; max :  97088
 Rang : 1 ; min : 12501 ; max :  98088
```

- Opération de réduction
  - arithmétique : +, -, *
  - logique .AND. .OR., ..
  - fonction : MIN, MAX

```
program parallel
implicit none
integer, parameter :: n=10
integer :: i, s=0, p1=1
double precision :: p2=1
!$OMP PARALLEL
!$OMP DO REDUCTION(+:s) REDUCTION(*:p1,p2)
do i = 1, n
  s = s + 1
  p1 = p1 * 2
  p2 = p2 * 10
end do
!$OMP END PARALLEL
print *,"s =",s, "; p1 =",p1, "; p2 =",p2
end program parallel
```

```
> ifort -qopenmp prog.f90
> export OMP_NUM_THREADS=4
> ./a.out
 s =           10 ;
p1 =         1024 ;
p2 =   10000000000.0000
```

- La directive WORKSHARE permet la parallélisation d'instructions intrinsèquement parallèles :
  - FORALL, WHERE
  - fonctions SUM, MATMUL...
  - instructions sur les tableaux

```
program parallel
implicit none
integer, parameter :: m=100000, n=20000
integer :: i, j
real, dimension(m,n) :: a, b
call random_number(b)
a(:,:) = 1.
!$OMP PARALLEL
!$OMP WORKSHARE
WHERE(b(:,:) >= 0.5) a(:,:)=sqrt(b(:,:))
!$OMP END WORKSHARE NOWAIT
!$OMP END PARALLEL
end program parallel
```

```
> ifort -qopenmp prog.f90
> export OMP_NUM_THREADS=16
> time ./a.out
real    0m21.892s
user    0m8.236s
sys     0m19.152s
```

-4.6-Exécution exclusive

- La directive SINGLE permet d'exécuter une partie du code par un seul thread

```
program parallel
use OMP_LIB
implicit none
integer rang, i
i=0
!$OMP PARALLEL FIRSTPRIVATE(i)
PRIVATE(rang)
rang = omp_get_thread_num()
!$OMP SINGLE
read (*,*) i
!$OMP END SINGLE
print *,rang,i
!$OMP SINGLE
read (*,*) i
!$OMP END SINGLE COPYPRIVATE (i)
print *,rang,i
!$OMP END PARALLEL
end program parallel
```

```
> ifort -qopenmp prog.f90
> export OMP_NUM_THREADS=4
> time ./a.out
3
           0           3
           3           0
           1           0
           2           0
5
           3           5
           0           5
           2           5
           1           5
```

- La directive MASTER permet d'exécuter une partie du code par le maître (thread 0)

-4.7- Synchronisation

- La synchronisation permet d'attendre que toutes les tâches soient arrivées à la même instruction. En l'absence de la clause NOWAIT en fin de région parallèle, il y a un synchronisation.

- La directive BARRIER permet une synchronisation explicite.

- Elle permet d'ordonner les tâches les unes par rapport aux autres, grâce notamment aux directives :
  - REDUCTION
  - DO ORDERED : exécution ordonnée d'une boucle, identique à une exécution séquentielle.
  - ATOMIC : l'instruction qui suit la directive atomic est exécuté par une seule tâche à la fois
  - CRITICAL : définit une région parallèle où les tâches s’exécutent l'une après l'autre dans un ordre indéterminé

-4.8- Conclusion

- OpenMP permet de paralléliser aisément un programme séquentiel de manière progressive.
- La parallélisation va pouvoir se faire sur les boucles et sur des sections indépendantes.
- Minimiser le nombre de régions parallèles dans le code.
- Adapter le nombre de tâches demandé à la taille du problème à traiter afin de minimiser les surcoûts de gestion des tâches par le système.
- Dans la mesure du possible, paralléliser la boucle la plus externe, dernière dimension d'un tableaux.
- Utiliser la clause SCHEDULE(RUNTIME) pour pouvoir changer dynamiquement l'ordonnancement et la taille des paquets d'itérations dans une boucle.

- La directive SINGLE et la clause NOWAIT peuvent permettre de baisser le temps de restitution au prix, le plus souvent, d'une synchronisation explicite.
- La directive ATOMIC et la clause REDUCTION sont plus restrictives mais plus performantes que la directive CRITICAL.
- La clause IF permet une parallélisation conditionnelle

```
!$OMP PARALLEL DO SCHEDULE(RUNTIME)&
!$OMP IF(n.gt.514)
do j = 2, n-1
  do i = 1, n
     b(i,j) = a(i,j+1) - a(i,j-1)
  end do
end do
!$OMP END PARALLEL DO
```

-5- MPI

-5.1- Historique

En Juin 1994, le forum MPI (Message Passing Interface) définit un ensemble de sous-programmes de la bibliothèque d'échange de messages MPI . Une quarantaine d'organisations y participent. Cette norme, MPI-1, vise à la fois la portabilité et la garantie de bonnes performances.

En Juillet 1997, la norme MPI-2 est publiée. Elle permet la gestion dynamique de processus, la copie mémoire à mémoire, la définition de types dérivés et MPI-IO

-5.2- Définition

Une application MPI est un ensemble de processus qui exécutent des instructions indépendamment les unes des autres et qui communiquent via l'appel à des procédures de la bibliothèque MPI.

On peut définir plusieurs catégories de procédure :

- environnement
- communications point à point
- communications collectives
- types de données dérivées
- topologies
- groupes et communicateurs

-5.3- Environnement

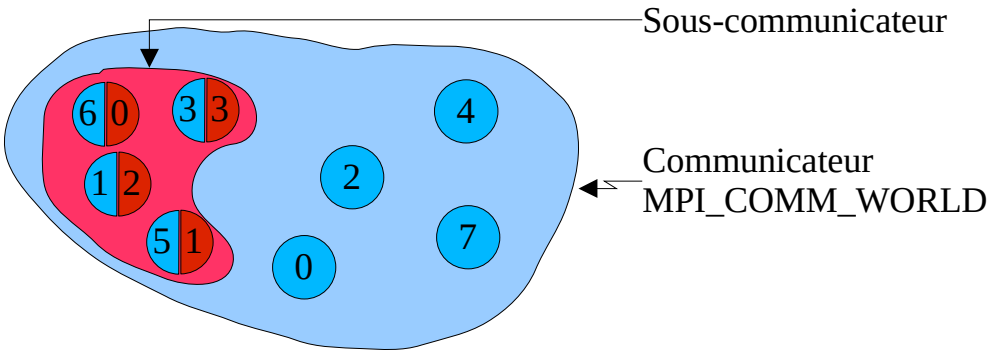
MPI_INIT() permet d'initialiser l'environnement MPI :

```
integer, intent(out) :: code
call MPI_INIT(code)
```

MPI_FINALIZE() permet de désactiver cet environnement :

```
integer, intent(out) :: code
call MPI_FINALIZE(code)
```

MPI_INIT initialise le communicateur par défaut, MPI_COMM_WORLD qui contient tous les processus actifs. Un communicateur est un ensemble de processus qui peuvent communiquer entre eux. Chaque processeur possède un rang unique dans la machine. Les N processeurs sont numérotés de 0 à N-1. Un processus peut appartenir à plusieurs communicateurs avec un rang différent.



La procédure MPI_COMM_SIZE() permet de connaître le nombre de processeurs gérés par un communicateur :

```
integer :: code, nbproc
call MPI_COMM_SIZE(MPI_COMM_WORLD,nbproc,code)
```

Le rang du processus est donné par la procédure MPI_COMM_RANK() :

```
integer :: code, rang
call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
```

Exemple :

```
Program SizeRang
implicit none
include 'mpif.h'
integer :: code, rang, nbproc
call MPI_INIT(code)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nbproc,code)
call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
call MPI_FINALIZE(code)
if (rang==0) print *,'Nombre de processus : ',nbproc
print *,'Je suis le processus de rang ',rang
end program SizeRang
```

```
> mpirun -np 3 SizeRang
 Je suis le processus de rang 1
 Nombre de processus : 3
 Je suis le processus de rang 0
 Je suis le processus de rang 2
```

-5.4- Communication point à point

-5.4.1- Définition

Une communication point à point est effectuée entre deux processus, un émetteur et un récepteur, identifiés par leur rang dans le communicateur au sein duquel se fait la communication.

L'enveloppe du message est constituée :

- du rang du processus émetteur
- du rang du processus récepteur
- du nom du communicateur
- du type de données
- de la longueur du message
- de l'étiquette du message

-5.4.2- Les procédures standard de communication

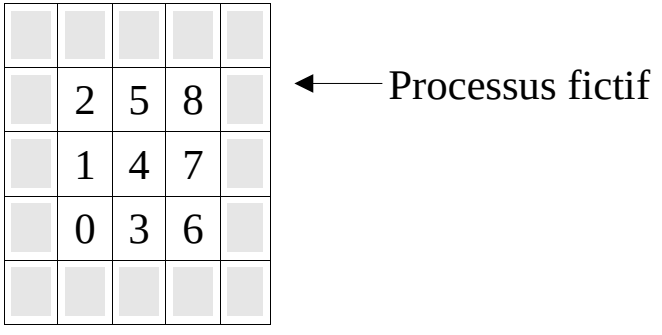
Les deux procédures standards d'envoi et de réception de message sont :

```
MPI_SEND(valeur,taille,type,destination,etiquette,comm,code)
MPI_RECV(valeur,taille,type,source,etiquette,comm,statut,code)
```

Pour la procédure MPI_RECV, le rang du processus récepteur peut être MPI_ANY_SOURCE.

La valeur de l'étiquette MPI_ANY_TAG

On peut également effectuer des communications avec un processus fictif de rang MPI_PROC_NULL.



Il existe d'autres modes de communication : synchrone, bufferisé

Les communications peuvent également être bloquantes ou non bloquantes.

Rq : MPI_SEND est implémenté de façon bloquante ce qui peut entraîner des situations de verrouillage si deux processus s'envoient mutuellement un message à un instant donné. Il faut dans ce cas intervertir MPI_SEND et MPI_RECV pour l'un des processus, ou utiliser la procédure MPI_SENDRECV

-5.4.3- Types de données

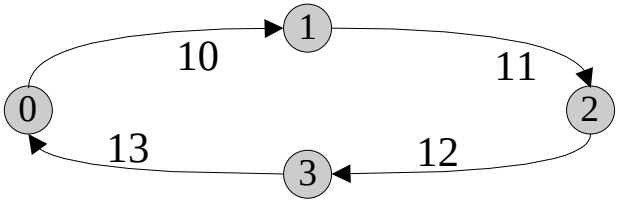
```
MPI_INTEGER
MPI_REAL
MPI_DOUBLE_PRECISION
MPI_COMPLEX
MPI_LOGICAL
MPI_CHARACTER
```

-5.4.4- Exemples

```
Program CommPaP
include 'mpif.h'
integer :: code, rang, val
integer, parameter :: tag=100
integer, dimension(MPI_STATUS_SIZE) :: statut
call MPI_INIT(code)
call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
if (rang==0) then
        val=999
        call MPI_SEND(val,1,MPI_INTEGER,1,tag,MPI_COMM_WORLD,code)
elseif (rang==1) then
        print *,"La valeur de la variable val est ",val
        call MPI_RECV(val,1,MPI_INTEGER,0,tag,MPI_COMM_WORLD,statut,code)
        print *,"Le processus 1 a recu la valeur ",val," du processus 0"
endif
call MPI_FINALIZE(code)
end program CommPaP
```

```
> mpirun -np 2 CommPaP
 La valeur de la variable val est 0
 Le processus 1 a recu la valeur 999 du processus 0
```

Communication en anneaux




```
apres=mod(rang+1,nbproc)
avant=mod(nbproc+rang-1,nbproc)
if (mod(rang,2)==0) then
call MPI_SEND(rang+10,1,MPI_INTEGER,apres,tag,MPI_COMM_WORLD,code)
call MPI_RECV(val,1,MPI_INTEGER,avant,tag,MPI_COMM_WORLD,statut,code)
else
call MPI_RECV(val,1,MPI_INTEGER,avant,tag,MPI_COMM_WORLD,statut,code)
call MPI_SEND(rang+10,1,MPI_INTEGER,apres,tag,MPI_COMM_WORLD,code)
endif
print *,'Moi processus ',rang," j'ai recu ",val,' du processus',avant
```

```
 > mpirun -np 4 CommAnneau
 Moi processus 1 j'ai recu 10 du processus 0
 Moi processus 2 j'ai recu 11 du processus 1
 Moi processus 0 j'ai recu 13 du processus 3
 Moi processus 3 j'ai recu 12 du processus 2
```

Communication SEND_RECV

```
apres=mod(rang+1,nbproc)
avant=mod(nbproc+rang-1,nbproc)
call MPI_SENDRECV(rang+10,1,MPI_INTEGER,apres,tag,
                  val,    1,MPI_INTEGER,avant,tag,MPI_COMM_WORLD,statut,code)
print *,'Moi processus ',rang,'j'ai recu ',val,' du processus',avant
```

-5.4.5- Mode de communications

Synchroneous : l'envoi du message se termine lorsque la réception est achevée.

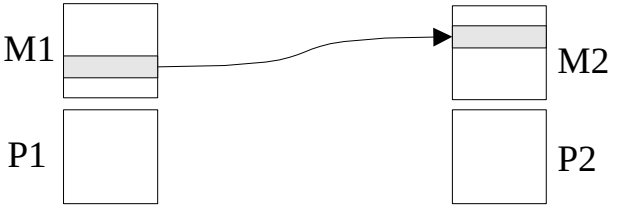
Buffered : le programmeur doit effectuer une recopie temporaire du message. L'envoi se termine lorsque la recopie est achevée. L'envoi est ainsi découplé de la réception.

Standard : MPI effectue une recopie temporaire ou non suivant la taille du message.

Si il y a une recopie, l'envoi est découplé de la réception dans le cas contraire, l'envoi se termine lorsque la réception est, elle même, terminée. Cela peut entraîner des situations de blocage. MPI_SEND est remplacé implicitement par un MPI_BSEND pour les messages de petite taille.

Ready : l'envoi ne peut être effectué que si la réception à été initiée. (utilisé pour les applications clients-serveurs)

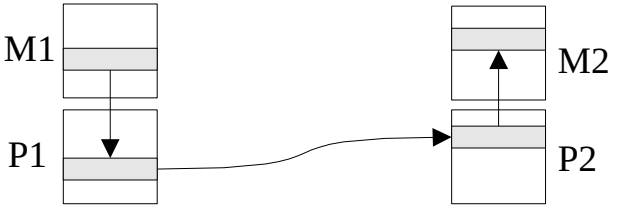
Envoi bloquant : sans recopie temporaire du message. L'envoi et la réception sont couplées.

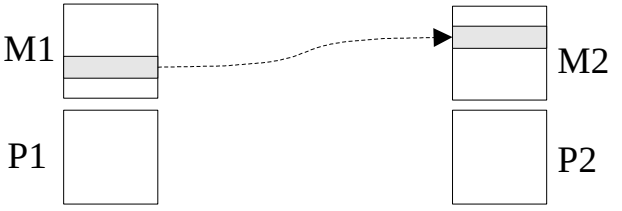


Envoi non bloquant :

- avec recopie temporaire du message. L'envoi et la réception ne sont pas couplées.

- sans recopie temporaire du message. L'envoi et la réception sont couplées. Attention à ne pas modifier la donnée avant qu'elle n'ait quittée la mémoire du processus émetteur. Ce sont des tâches de fond.





modes	bloquant	Non bloquante
Envoi standard	MPI_SEND	MPI_ISEND
Envoi synchroneous	MPI_SSEND	MPI_ISSEND
Envoi buffered	MPI_BSEND	MPI_IBSEND
Réception	MPI_RECV	MPI_IRECV

L'utilisation des procédures MPI_ISSEND et MPI_IRECV permettent de recouvrir les communications par des calculs => gains de performance.

Pour les procédures non bloquantes, il existe des procédures qui permettent de synchroniser les processus (MPI_WAIT, MPI_WAITALL), ou de tester si une requête est bien terminée (MPI_TEST, MPI_TESTALL)

-5.5- Communications collectives

Les communications collectives permettent par l'appel à une procédure de faire plusieurs communications point à point.

Ces communications sont :

- effectuées sur l'ensemble des processus appartenant à un communicateur.
- Synchronisées : l'appel à la procédure se termine lorsque la communication a été achevée pour l'ensemble des processus.

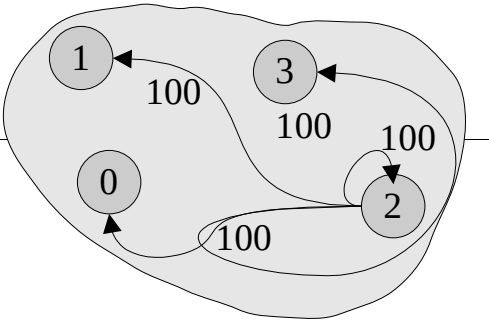
-5.5.1- Synchronisation des processus

La procédure MPI_BARRIER() réalise une synchronisation globale.

```
interger, intent(out) :: code
call MPI_BARRIER(MPI_COMM_WORLD,code)
```

-5.5.2- Diffusion générale

On envoi la même donnée à chaque processus.

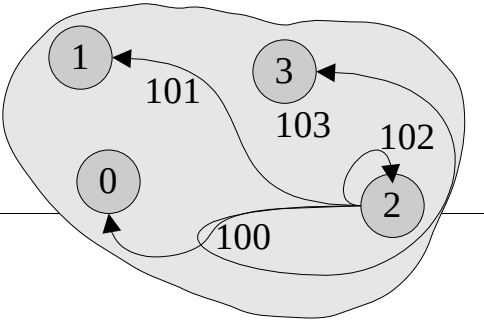



```
Program bcast
include 'mpif.h'
integer valeur, rang, code
call MPI_INIT(code)
call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
if (rang==2) valeur=100
call MPI_BCAST(valeur,1,MPI_INTEGER,2,MPI_COMM_WORLD,code)
print *,'Le processus ',rang,' a recu la valeur ',valeur,' du processus 2'
call MPI_FINALIZE(code)
end Program bcast
```

```
 > mpirun -np 4 bcast
 Le processus 2 a recu la valeur 100 du processus 2
 Le processus 3 a recu la valeur 100 du processus 2
 Le processus 0 a recu la valeur 100 du processus 2
 Le processus 1 a recu la valeur 100 du processus 2
```

-5.5.3- Diffusion sélective

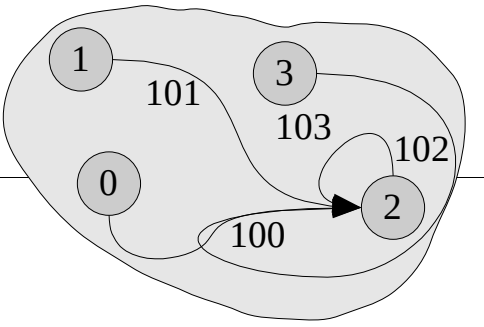
Envoi d'éléments contigus en mémoire à chaque processus.




```
Program scatter
include 'mpif.h'
integer valeurs(4), val, rang, code
call MPI_INIT(code)
call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
if (rang==2) valeurs(:)=(/(100+i,i=0,3)/)
call MPI_SCATTER(valeurs,1,MPI_INTEGER,val,1,MPI_INTEGER,2,MPI_COMM_WORLD,code)
print *,'Le processus ',rang,' a recu la valeur ',val,' du processus 2'
call MPI_FINALIZE(code)
end Program scatter
```

```
 > mpirun -np 4 scatter
 Le processus 2 a recu la valeur 102 du processus 2
 Le processus 3 a recu la valeur 103 du processus 2
 Le processus 0 a recu la valeur 100 du processus 2
 Le processus 1 a recu la valeur 101 du processus 2
```

-5.5.4-Collecte de données réparties

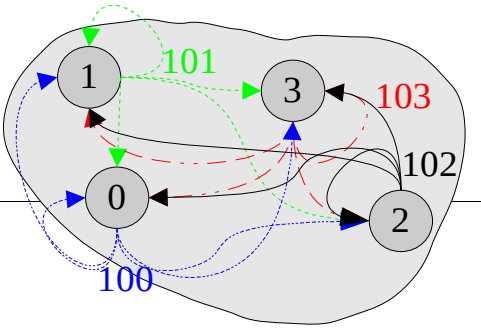



```
Program gather
include 'mpif.h'
integer valeurs(4), val, rang, code
call MPI_INIT(code)
call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
val=100+rang
call MPI_GATHER(val,1,MPI_INTEGER,valeurs,1,MPI_INTEGER,2,MPI_COMM_WORLD,code)
if (rang==2 ) print *,'Le processus 2 a recu les valeurs ',valeurs
call MPI_FINALIZE(code)
end Program gather
```

```
 > mpirun -np 4 gather
 Le processus 2 a recu les valeurs 100 101 102 103
```

Les données sont rangées dans le tableau valeurs selon le rang des processus émetteurs, par ordre croissant.

-5.5.5- Collecte générale



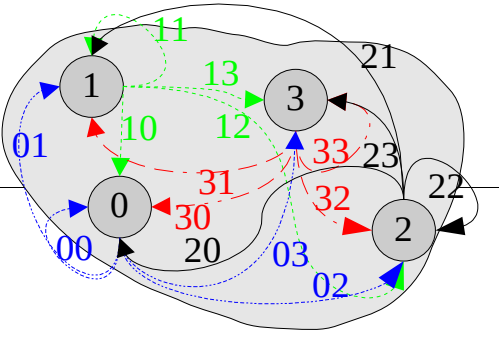
Collecte par l'ensemble des processus de données réparties.

```
Program allgather
include 'mpif.h'
integer valeurs(4), val, rang, code
call MPI_INIT(code)
call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
val=100+rang
call MPI_ALLGATHER(val,1,MPI_INTEGER,valeurs,1,MPI_INTEGER,MPI_COMM_WORLD,code)
print *,'Le processus ',rang,' a recu les valeurs ',valeurs
call MPI_FINALIZE(code)
end Program allgather
```

```
 > mpirun -np 4 allgather
 Le processus 2 a recu les valeurs 100 101 102 103
 Le processus 1 a recu les valeurs 100 101 102 103
 Le processus 3 a recu les valeurs 100 101 102 103
 Le processus 0 a recu les valeurs 100 101 102 103
```

-5.5.6- Alltoall

Diffusion sélective de données réparties, par tous les processus.




```
Program alltoall
include 'mpif.h'
integer vals(4), rang, code
call MPI_INIT(code)
call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
vals(:)=(/(10*rang+i,i=0,3)/)
call MPI_ALLTOALL(vals,1,MPI_INTEGER,vals,1,MPI_INTEGER,MPI_COMM_WORLD,code)
print *,'Le processus ',rang,' a recu les valeurs ',vals
call MPI_FINALIZE(code)
end Program alltoall
```

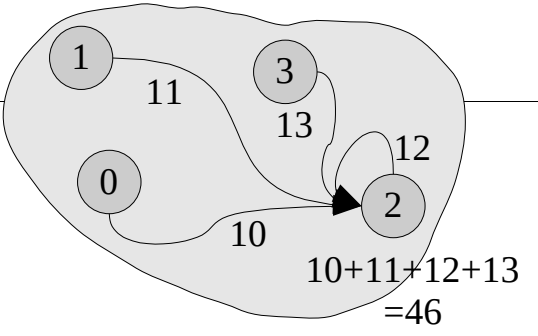
```
 > mpirun -np 4 allgather
 Le processus 2 a recu les valeurs 2 12 22 32
 Le processus 1 a recu les valeurs 1 11 21 31
 Le processus 3 a recu les valeurs 3 13 23 33
 Le processus 0 a recu les valeurs 0 10 20 30
```

-5.5.7- Les opérations de réduction

Ces procédures effectuent, en plus du transfère de données, une opération sur ces données (somme, produit, minimum, maximum...) : MPI_REDUCE , MPI_ALLREDUCE (MPI_REDUCE+MPI_BCAST: diffusion du résultat)

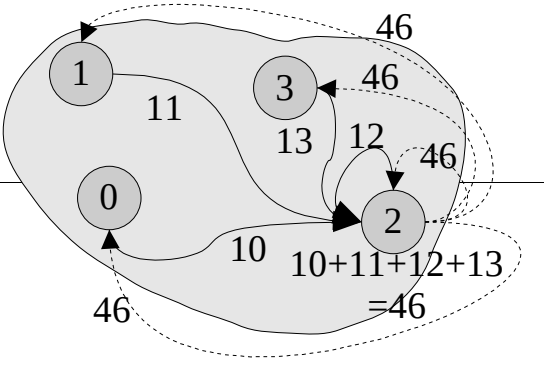
Principales opérations de réduction pré-définies :

MPI_SUM	Somme des données
MPI_PROD	Produit des données
MPI_MAX	Recherche du maximum
MPI_MIN	Recherche du minimum
MPI_MAXLOC	Recherche de l'indice du minimum
MPI_MINLOC	Recherche de l'indice du maximum
MPI_LAND	ET logique
MPI_LOR	OU logique
MPI_LXOR	OU exclusif logique




```
Program reduce
include 'mpif.h'
integer somme, val, rang, code
call MPI_INIT(code)
call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
val=10+rang
call MPI_REDUCE(val,somme,1,MPI_INTEGER,MPI_SUM,2,MPI_COMM_WORLD,code)
print *,'Pour le processus ',rang,' la somme est de ',somme
call MPI_FINALIZE(code)
end Program reduce
```

```
 > mpirun -np 4 reduce
 Pour le processus 0 la somme est de 0
 Pour le processus 1 la somme est de 0
 Pour le processus 3 la somme est de 0
 Pour le processus 2 la somme est de 46
```




```
Program allreduce
include 'mpif.h'
integer somme, val, rang, code
call MPI_INIT(code)
call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
val=10+rang
call MPI_ALLREDUCE(val,somme,1,MPI_INTEGER,MPI_SUM,MPI_COMM_WORLD,code)
print *,'Pour le processus ',rang,' la somme est de ',somme
call MPI_FINALIZE(code)
end Program allreduce
```

```
 > mpirun -np 4 allreduce
 Pour le processus 0 la somme est de 46
 Pour le processus 1 la somme est de 46
 Pour le processus 3 la somme est de 46
 Pour le processus 2 la somme est de 46
```

-5.6- Types de données dérivées

Il existe des types de données pré-définis par MPI : MPI_INTEGER, MPI_REAL...

On peut en définir de nouveau en utilisant les procédures MPI_TYPE_CONTIGUOUS, MPI_TYPE_VECTOR, MPI_TYPE_HVECTOR.

Ces types sont activés par la procédure MPI_TYPE_COMMIT et détruits par la procédure MPI_TYPE_FREE.

-5.6.1- Types contigus

MPI_TYPE_CONTIGUOUS crée une structure de données à partir d'un ensemble homogène de données de type pré-défini qui sont contiguës en mémoire.

```
Program colonne
include 'mpif.h'
integer :: code, type_colonne, tag, rang, i, j, tab(3,4), statut(MPI_STATUS_SIZE)
call MPI_INIT(code)
call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
do i=1,3 do j=1,4
tab(i,j)=3*(j-1)+i-1+(100*rang+1)
enddo enddo
call MPI_TYPE_CONTIGUOUS(3,MPI_INTEGER,type_colonne,code)
call MPI_TYPE_COMMIT(type_colonne,code)
```

```
if (rang==0) then
        call MPI_SEND(tab(1,2),1,type_colonne,1,tag,MPI_COMM_WORLD,code)
elseif (rang==1) then
        call MPI_RECV(tab(2,3),1,type_colonne,0,tag,MPI_COMM_WORLD,statut,code)
endif
call MPI_TYPE_FREE(type_colonne,code)
call MPI_FINALIZE(code)
end Program colonne
```

1	4	7	10
2	5	8	11
3	6	9	12

tab processus 0

101	104	107	110
102	105	108	111
103	106	109	112

tab processus 1, t1

101	104	107	6
102	105	4	111
103	106	5	112

tab processus 1, t2

-5.6.2- Types avec un pas constant

MPI_TYPE_VECTOR crée une structure de données à partir d'un ensemble homogène de données de type pré-défini qui sont distantes d'un pas constant en mémoire.

```
MPI_TYPE_VECTOR(nb_donnees,taille,pas,ancien_type,nouveau_type,code)
```

```
Program ligne
include 'mpif.h'
integer :: code, type_ligne, tag, rang, i, j, tab(3,4), statut(MPI_STATUS_SIZE)
call MPI_INIT(code)
call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
do i=1,3
        do j=1,4
                tab(i,j)=3*(j-1)+i-1+(100*rang+1)
        enddo
enddo
call MPI_TYPE_VECTOR(4,1,3,MPI_INTEGER,type_ligne,code)
call MPI_TYPE_COMMIT(type_ligne,code)
if (rang==0) then
        call MPI_SEND(tab(1,1),1,type_ligne,1,tag,MPI_COMM_WORLD,code)
elseif (rang==1) then
        call MPI_RECV(tab(3,1),1,type_ligne,0,tag,MPI_COMM_WORLD,statut,code)
endif
call MPI_TYPE_FREE(type_ligne,code)
call MPI_FINALIZE(code)
end Program ligne
```

1	4	7	10
2	5	8	11
3	6	9	12

tab processus 0

101	104	107	110
102	105	108	111
103	106	109	112

tab processus 1, t1

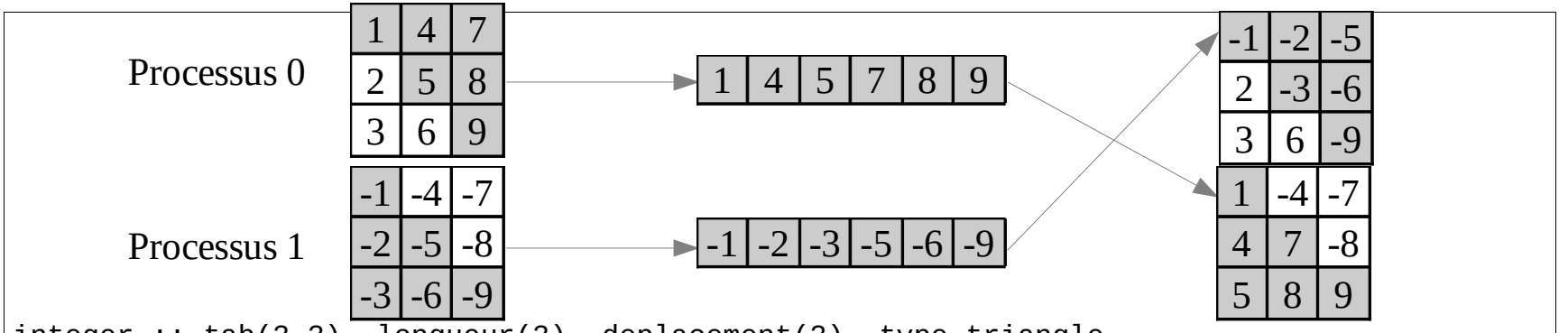
101	104	107	110
102	105	108	111
1	4	7	10

tab processus 1, t2

Le pas passé en paramètre de la procédure MPI_TYPE_HVECTOR est donné en octet.

-5.6.3- Type avec un pas variable

MPI_TYPE_INDEXED permet de définir un type qui comporte des blocs de données de type pré-défini, de taille variable, repérés par un déplacement par rapport au début de la structure. Ce déplacement est donné en nombre d'occurrence du type pré-définit. MPI_TYPE_HINDEXED : déplacement en octet.




```
integer :: tab(3,3), longueur(3), deplacement(3), type_triangle
if (rang==0) then longueur(:)=(/1,2,3/); deplacement=(/0,3,6/); endif
if (rang==1) then longueur(:)=(/3,2,1/); deplacement=(/0,4,8/); endif
call MPI_TYPE_INDEXED(3,longueur,deplacement,MPI_INTEGER,type_triangle,code)
call MPI_TYPE_COMMIT(type_triangle,code)
call MPI_SENDRECV_REPLACE(tab,1,type_triangle,mod(rang+1,2),tag,
                          mod(rang+1,2),tag,MPI_COMM_WORLD,statut,code)
```

-5.6.4- Type de données hétérogènes

MPI_TYPE_STRUCT est le constructeur de type le plus général. Il permet de faire la même chose que MPI_TYPE_INDEXED mais sur des données hétérogènes : le champ type de données est un tableau de type de données. Les déplacements sont donnés en octet, ils peuvent être calculés grâce à la procédure MPI_ADDRESS.

```
Integer, dimension(4) :: types, longueur, adresse, deplacement
integer :: type_personne

Type personne
        character(len=20)         :: nom
        character(len=30)         :: prenom
        integer                   :: age
        real                      :: taille
end type personne
type(personne) :: p
types = (/MPI_CHARACTER,MPI_CHARACTER,MPI_INTEGER,MPI_REAL/)
longueur = (/20,30,1,1/)
call MPI_ADDRESS(p%nom,adresse(1),code)
call MPI_ADDRESS(p%prenom,adresse(2),code)
call MPI_ADDRESS(p%age,adresse(3),code)
call MPI_ADDRESS(p%taille,adresse(4),code)
```

```
do i=1,4
        deplacement(i)=adresse(i)-adresse(1)
enddo
call MPI_TYPE_STRUCT(4,longueur,deplacement,types,type_personne,code)
call MPI_TYPE_COMMIT(type_personne,code)
if (rang==0) then
        p%nom='Dupond'; p%prenom='Pierre'; p%age=35; p%taille=1.85
        print *,"Le processus 0 envoi : ",p
        call MPI_SEND(p%type,1,type_personne,1,tag,MPI_COMM_WORLD,code)
else
        call MPI_RECV(p%type,1,type_personne,0,tag,MPI_COMM_WORLD,statut,code)
        print *,"Le processus 1 a recu : ",p
endif
call MPI_TYPE_FREE(type_personne,code)
```

-5.6.5-Quelques procédures utiles

Taille d'un type de données :

1	4	7
2	5	8
3	6	9

```
integer, intent(in)  :: type
integer, intent(out) :: taille,code
MPI_TYPE_SIZE(type,taille,code)
```

```
Taille type = 20 (5*4)
```

```
MPI_TYPE_EXTEND : taille d'un type aligné en mémoire
```

Adresse de début et de fin du type de données :

```
MPI_TYPE_LB(type,adresse,code)
MPI_TYPE_UB(type,adresse,code)
```

```
Borne inférieure : 0
Borne supérieure : 32 (8*4)
```

-5.7- Topologies

Pour les applications qui nécessite une décomposition de domaine il est intéressant de pouvoir définir une topologie qui organise les processus de façon régulière.

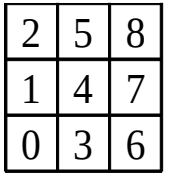
Il existe deux types de topologie :

- cartésienne

  définition d'une grille

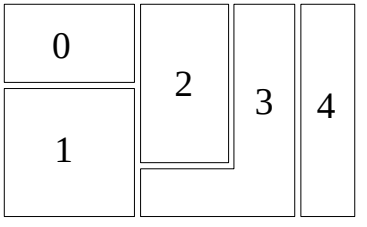
  périodique ou non

  identification des processus par leurs coordonnées



- graphe

  généralisation à des topologies plus complexes.



-5.7.1- Topologies cartésiennes

Une topologie cartésienne est définit par l'appel de la procédure MPI_CART_CREATE. Elle crée un nouveau communicateur.

```
Integer, parameter                :: nb_dim=2
integer, dimension(nb_dim)        :: dims
logical, dimension(nb_dim)        :: period
integer                           :: nom_comm, code
logical                           :: reorganise
call MPI_CART_CREATE(MPI_COMM_WORLD,nb_dim,dims,period,reorganise,nom_comm,code)
```

reorganise : false => le processus conserve le rang qu'il avait dans l'ancien communicateur.

Le choix du nombre de processus pour chaque dimension peut être laissé au soin du processus en utilisant la procédure MPI_DIMS_CREATE :

```
integer, intent(in)                         :: nb_proc, nb_dim
integer, dimension(nb_dim), intent(out)     :: dims
integer, intent(out)                        :: code
call MPI_DIMS_CREATE(nb_proc,nb_dim,dims,code)
```

La procédure MPI_CART_COORDS fournit les coordonnées d'un processus de rang donné dans la grille

```
integer, intent(in)                          :: nom_comm, rang, nb_dim
integer, dimension(nb_dim), intent(out)      :: coords
integer, intent(out)                         :: code
call MPI_CART_COORDS(nom_comm,rang,nb_dim,coords,code)
```

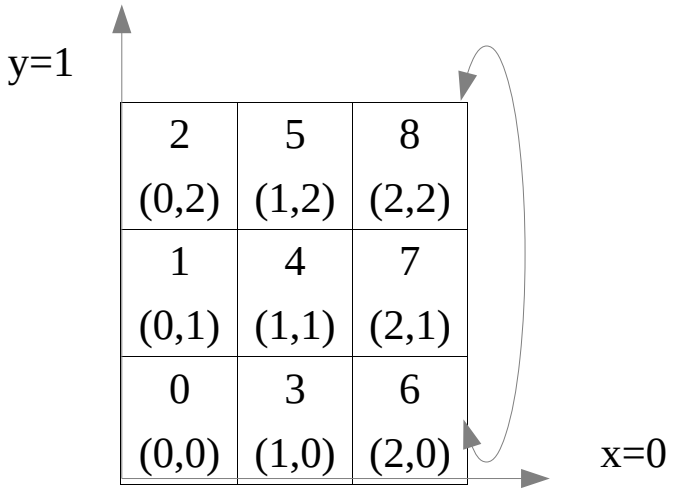
La procédure MPI_CART_RANK permet de connaître le rang du processus associé aux coordonnées dans la grille.

```
integer, intent(in)                          :: nom_comm
integer, dimension(nb_dim), intent(out)      :: coords
integer, intent(out)                         :: rang, code
call MPI_CART_RANK(nom_comm,coords,rang,code)
```

La procédure MPI_CART_SHIFT permet de connaître le rang des voisins d'un processus dans une direction donnée.

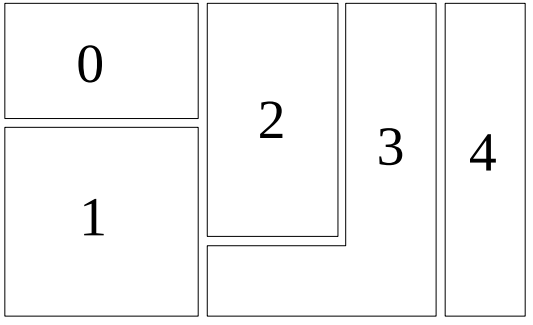
```
integer, intent(in)       :: nom_comm, direction, pas
integer, intent(out)      :: avant, apres, code
call MPI_CART_SHIFT(nom_comm,direction,pas,avant,apres,code)

call MPI_CART_SHIFT(nom_comm,0,1,gauche,droite,code)
call MPI_CART_SHIFT(nom_comm,1,1,dessous,dessus,code)
```



-5.7.2- Topologies graphes

Décomposition de domaine sur une grille non régulière : un processus peut avoir un nombre quelconque de voisins.



No proc	Liste des voisins	index
0	1, 2	2
1	0, 2, 3	5
2	0, 1, 3	8
3	1, 2, 4	11
4	3	12

La liste des voisins et le tableau d'index permet de définir les voisins d'un processus.

```
integer :: nom_comm, nb_proc, reorg, code
integer, dimension(nb_proc) :: index
integer, dimension(nb_max_voisin) :: voisins
index =   (/  2,   5,     8,    11,  12/)
voisins = (/1,2, 0,2,3, 0,1,3, 1,2,4, 3/)
call MPI_GRAPH_CREATE(MPI_COMM_WORLD,nb_proc,index,voisins,reorg,nom_comm,code)
```

La procédure MPI_GRAPH_NEIGHBORS_COUNT permet de connaître le nombre de voisins d'un processus donné.

```
integer, intent(in)   :: nom_comm, rang
integer, intent(out)  :: nb_voisins, code

MPI_GRAPH_NEIGHBORS_COUNT(nom_comm,rang,nb_voisins,code)
```

La procédure MPI_GRAPH_NEIGHBORS donne la liste des voisins d'un processus.

```
integer, intent(in)   :: nom_comm, rang, nb_voisins
integer, intent(out)  :: code
integer, dimension(nb_max_voisin), intent(out) :: voisins

MPI_GRAPH_NEIGHBORS(nom_comm,rang,nb_voisins,voisins,code)
```

-5.8- Gestion des groupes de processus

Pour construire l'espace de communication, on utilise un communicateur qui est constitué :

- d'un groupe de processus,
- d'un contexte de communication, i.e une propriété des communicateurs qui permet de partager l'espace de communication, gérer les communications point-à-point et collectives de telle sorte qu'elles n'interfèrent pas.

Il y a deux manières de construire un communicateur :

- à partir d'un autre communicateur,
- par l'intermédiaire d'un groupe de processus.

Le communicateur par défaut est MPI_COMM_WORLD. Il contient tous les processus actifs.

Il est créé avec MPI_INIT et détruit avec MPI_FINALIZE.

-5.8.1-Communicateur issu d'un communicateur

On peut créer des sous-ensembles de communication, on partage le communicateur initial en plusieurs sous-communicateurs distincts mais de même nom.

Pour cela on définit :

- une couleur associant à chaque processus le numéro du communicateur auquel il appartiendra
- une clé permettant de numéroter les processus dans chaque communicateur.

```
Integer, intent(in) :: nom_comm ! Communicateur courant à partager
integer, intent(in) :: couleur, cle
integer, intent(out) :: new_comm, code
MPI_COMM_SPLIT (nom_comm,couleur,cle,new_comm,code)
```

L'intérêt de ce découpage de communicateur est de pouvoir effectuer une communication collective que sur une partie des processus de l'application.

Exemple : définition de deux communicateurs content les processus pairs et impairs de MPI_COMM_WORLD

Rang dans MPI_COMM_WORLD	0	1	2	3	4	5	6	7
couleur	0	1	0	1	0	1	0	1
clé	5	2	3	3	8	0	3	7
Rang dans le nouveau communicateur	2	1	0	2	3	0	1	3

La fonction de destruction d'un communicateur est MPI_COMM_FREE :

```
integer, intent(inout) :: nom_comm
integer, intent(out)   :: code
MPI_COMM_FREE(nom_comm,code)
```

-5.8.2-Communicateur issu d'un groupe

Un groupe est un ensemble ordonné de processus identifiés par leurs rangs.

Initialement tous les processus appartiennent au groupe associé au communicateur MPI_COMM_WORLD.

Tout communicateur est associé à un groupe. La procédure MPI_COMM_GROUP permet de connaître ce groupe.

```
Integer, intent(in)  :: nom_comm
integer, intent(out) :: groupe, code
MPI_COMM_GROUP(nom_comm,groupe,code)
```

A partir de ce groupe on peut construire un sous-groupe. Pour cela on doit au préalable définir un vecteur contenant les rangs des processus qui formeront ce sous-groupe. Ensuite on utilise MPI_GROUP_INCL :

```
integer, intent(in)                          :: groupe,nb_proc_grp
integer, dimension(nb_proc_grp), intent(in) :: rangs
integer, intent(out)                         :: newgroupe,code
MPI_GROUP_INCL (groupe,nb_proc_grp,rangs,newgroupe,code)
```

On peut également utiliser la procédure MPI_GROUP_EXCL, qui elle exclue du sous-groupe les processus qui ne sont pas dans le tableau rangs :

```
integer, intent(in)                          :: groupe, nb_proc_grp
integer, dimension(nb_proc_grp), intent(in) :: rangs
integer, intent(out)                         :: newgroupe, code
MPI_GROUP_EXCL (groupe,nb_proc_grp,rangs,newgroupe,code)
```

Une fois le sous-groupe crée, on construit le communicateur associé à l'aide de MPI_COMM_CREATE :

```
integer, intent(in)  :: nom_comm, newgroupe
integer, intent(out) :: new_comm, code
MPI_COMM_CREATE (nom_comm,newgroup,new_comm,code)
```

Il reste à libérer le sous-groupe crée avec MPI_GROUP_FREE :

```
integer, intent(inout) :: groupe
integer, intent(out)   :: code
MPI_GROUP_FREE (groupe, code)
```

Comme pour le communicateur on dispose de procédures pour déterminer le nombre d'éléments du groupe :

```
integer, intent(in)  :: groupe
integer, intent(out) :: taille, code
MPI_GROUP_SIZE(groupe,taille,code)
```

et le rang de ses processus :

```
integer, intent(in)  :: groupe
integer, intent(out) :: rang, code
MPI_GROUP_RANK(groupe,rang,code)
```

Opérations sur les groupes :

```
integer, intent(in)  :: groupe1, groupe2
integer, intent(out) :: newgroup, resultat, code
MPI_GROUP_COMPARE(groupe1,groupe2,resultat,code)
MPI_GROUP_UNION(groupe1,groupe2,newgroup,code)
MPI_GROUP_INTERSECTION(groupe1,groupe2,newgroup,code)
MPI_GROUP_DIFFERENCE(groupe1,groupe2,newgroup,code)
```

-5.9- Quelques procédures

-5.9.1-Temps de communication

```
MPI_WTIME() ou MPI_WTICK()
```

temps (sec) = calcul+préparation (latence+surcoût)+transfert

latence : temps d'initialisation des paramètres réseaux

surcoût : temps de préparation du message lié à l'implémentation MPI et au mode de transfert.

-5.9.2- Nom d'un processeur

```
Integer, intent(out) :: longueur, code
character(MPI_MAX_PROCESSOR_NAME), intent(out) :: nom
if (rang==0) then
        MPI_GET_PROCESSOR_NAME(nom,longueur,code)
        print *,'le nom du processeur 0 est : ',nom(1:longueur)
endif
```

```
le nom du processeur 0 est : tycho01.obspm.fr
```

-5.10- MPI-2

Une nouvelle norme est disponible depuis Juillet 1997 : MPI-2.

Elle permet :

- la gestion dynamique des processus,
- la communication de mémoire à mémoire,
- entrées/sorties parallèles.

-5.11- Bibliographie

- Les spécifications de la norme MPI :
  a message passing interface standard, Mars 1994.
  ftp://ftp.irisa.fr/pub/netlib/mpi/drafts/draft-final.ps

- Marc Snir & al. MPI :
  The Complete Reference. Second Edition. MIT Press, 1998.
  Volume 1, The MPI core. Volume 2, MPI-2

- MPI subroutine reference :
  - http://www.ccr.jussieu.fr/ccr/Documentation/Calcul/ppe.html/d3d80mst.html
  - http://www.lam-mpi.org/tutorials/bindings/

-5.12- Bibliothèques

- LAPACK (Linear Algebra PACKage) :
  librairie Fortran de calcul numérique dédiée à l'algèbre linéaire.

- SCALAPACK (SCAlable LAPACK) :
  inclut un sous-ensemble des routines LAPACK, optimisé pour les architectures à mémoire distribuée

- BLACS (Basic Linear Algebra Communication Subprogram) :
  propose une interface pour l'utilisation du transfert de messages pour les applications en algèbre linéaire.

- FFTW : transformée de Fourier

-6- OpenMP versus MPI

- OpenMP permet la parallélisation multitâches, la communication entre les tâches est implicite.
- OpenMP est utilisé sur le machine à mémoire partagée.
- MPI permet l'exécution en parallèle de plusieurs processus, la communication entre ces processus doit être effectuée explicitement par le programmeur.
- MPI est utilisé sur les machines à méoire distribuée.
- Programmation mixte OpenMP et MPI

-7- Index

Index des procédures MPI