3.5. Communications MPI collectives

4 – Communications collectives

4.1 – Notions générales

Notions générales

- Les communications collectives permettent de faire en une seule opération une série de communications point à point.
- Une communication collective concerne toujours tous les processus du communicateur indiqué.
- Pour chacun des processus, l'appel se termine lorsque la participation de celui-ci à l'opération collective est achevée, au sens des communications point-à-point (donc quand la zone mémoire concernée peut être modifiée).
- La gestion des étiquettes dans ces communications est transparente et à la charge du système. Elles ne sont donc jamais définies explicitement lors de l'appel à ces sous-programmes. Cela a entre autres pour avantage que les communications collectives n'interfèrent jamais avec les communications point à point.

Types de communications collectives

Il y a trois types de sous-programmes :

1. celui qui assure les synchronisations globales : `MPI_BARRIER()`.
2. ceux qui ne font que transférer des données :
   - diffusion globale de données : `MPI_BCAST()` ;
   - diffusion sélective de données : `MPI_SCATTER()` ;
   - collecte de données réparties : `MPI_GATHER()` ;
   - collecte par tous les processus de données réparties : `MPI_ALLGATHER()` ;
   - collecte et diffusion sélective, par tous les processus, de données réparties : `MPI_ALLTOALL()`.
3. ceux qui, en plus de la gestion des communications, effectuent des opérations sur les données transférées :
   - opérations de réduction (somme, produit, maximum, minimum, etc.), qu'elles soient d'un type prédéfini ou d'un type personnel : `MPI_REDUCE()` ;
   - opérations de réduction avec diffusion du résultat (équivalent à un `MPI_REDUCE()` suivi d'un `MPI_BCAST()`) : `MPI_ALLREDUCE()`.

4 – Communications collectives

4.2 – Synchronisation globale : MPI_BARRIER()

Synchronisation globale : MPI_BARRIER()

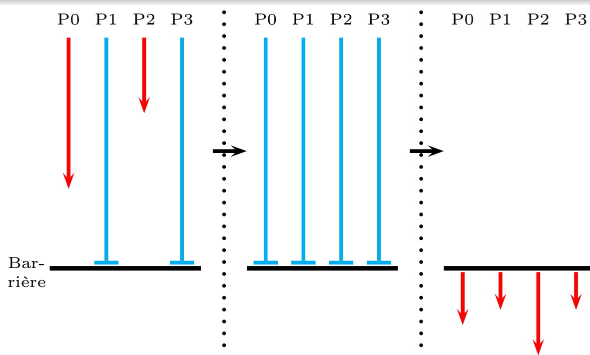


Figure 12 – Synchronisation globale : MPI_BARRIER()

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

call MPI_BARRIER(MPI_COMM_WORLD,code)
```

4 – Communications collectives

4.3 – Diffusion générale : MPI_BCAST()

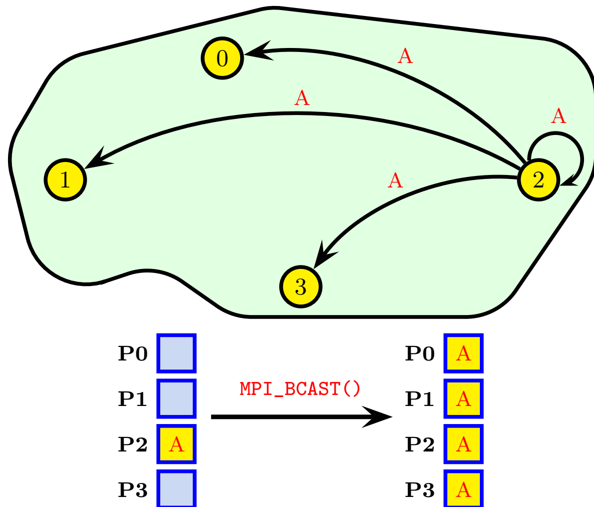


Figure 13 – Diffusion générale : MPI_BCAST()

Diffusion générale : MPI_BCAST()

```
<type et attribut> :: message
integer :: longueur, type, rang_source, comm, code

call MPI_BCAST(message, longueur, type, rang_source, comm, code)
```

1. Envoi, à partir de l'adresse message, d'un message constitué de longueur élément de type type, par le processus rang_source, à tous les autres processus du communicateur comm.
2. Réception de ce message à l'adresse message pour les processus autre que rang_source.

```fortran
program bcast
  use mpi
  implicit none

  integer :: rang,valeur,code

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

  if (rang == 2) valeur=rang+1000

  call MPI_BCAST(valeur,1,MPI_INTEGER,2,MPI_COMM_WORLD,code)

  print *,'Moi, processus ',rang,', j''ai reçu ',valeur,' du processus 2'

  call MPI_FINALIZE(code)

end program bcast
```

```
> mpiexec -n 4 bcast

Moi, processus 2, j'ai reçu 1002 du processus 2
Moi, processus 0, j'ai reçu 1002 du processus 2
Moi, processus 1, j'ai reçu 1002 du processus 2
Moi, processus 3, j'ai reçu 1002 du processus 2
```

4 – Communications collectives

4.4 – Diffusion sélective : MPI_SCATTER()

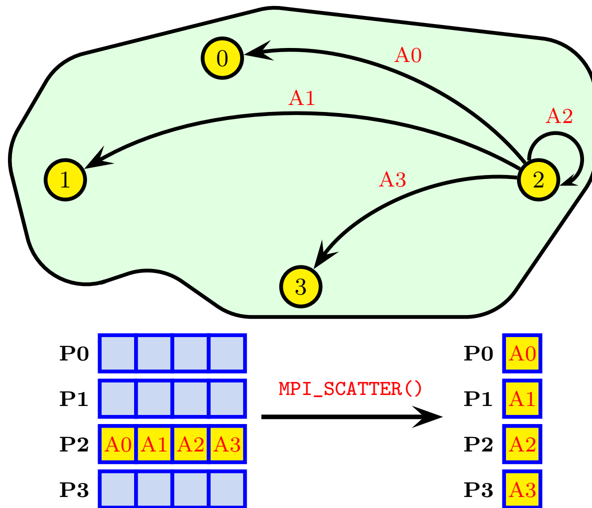


Figure 14 – Diffusion sélective : MPI_SCATTER()

Diffusion sélective : MPI_SCATTER()

```
<type et attribut>:: message_a_repartir, message_recu
integer :: longueur_message_emis, longueur_message_recu
integer :: type_message_emis, type_message_recu
integer :: rang_source, comm, code

call MPI_SCATTER(message_a_repartir,longueur_message_emis,type_message_emis,
         message_recu,longueur_message_recu,type_message_recu,rang_source,comm,code)
```

1. Distribution, par le processus rang_source, à partir de l'adresse message_a_repartir, d'un message de taille longueur_message_emis, de type type_message_emis, à tous les processus du communicateur comm ;
2. réception du message à l'adresse message_recu, de longueur longueur_message_recu et de type type_message_recu par tous les processus du communicateur comm.

Remarques :

- Les couples (longueur_message_emis, type_message_emis) et (longueur_message_recu, type_message_recu) doivent être tels que les quantités de données envoyées et reçues soient égales.
- Les données sont distribuées en tranches égales, une tranche étant constituée de longueur_message_emis éléments du type type_message_emis.
- La ième tranche est envoyée au ième processus.

```
 1 program scatter
 2   use mpi
 3   implicit none
 4
 5   integer, parameter                :: nb_valeurs=8
 6   integer                           :: nb_procs,rang,longueur_tranche,i,code
 7   real, allocatable, dimension(:) :: valeurs,donnees
 8
 9   call MPI_INIT(code)
10   call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
11   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
12   longueur_tranche=nb_valeurs/nb_procs
13   allocate(donnees(longueur_tranche))
14   allocate(valeurs(nb_valeurs))
15
16   if (rang == 2) then
17      valeurs(:)=(/(1000.+i,i=1,nb_valeurs)/)
18      print *,'Moi, processus ',rang,'envoie mon tableau valeurs : ',&
19               valeurs(1:nb_valeurs)
20   end if
21
22   call MPI_SCATTER(valeurs,longueur_tranche,MPI_REAL,donnees,longueur_tranche, &
23                    MPI_REAL,2,MPI_COMM_WORLD,code)
24   print *,'Moi, processus ',rang,', j''ai reçu ', donnees(1:longueur_tranche), &
25          ' du processus 2'
26   call MPI_FINALIZE(code)
27
28 end program scatter
```

```
> mpiexec -n 4 scatter
Moi, processus 2 envoie mon tableau valeurs :
1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.

Moi, processus 0, j'ai reçu 1001. 1002. du processus 2
Moi, processus 1, j'ai reçu 1003. 1004. du processus 2
Moi, processus 3, j'ai reçu 1007. 1008. du processus 2
Moi, processus 2, j'ai reçu 1005. 1006. du processus 2
```

4 – Communications collectives

4.5 – Collecte : MPI_GATHER()

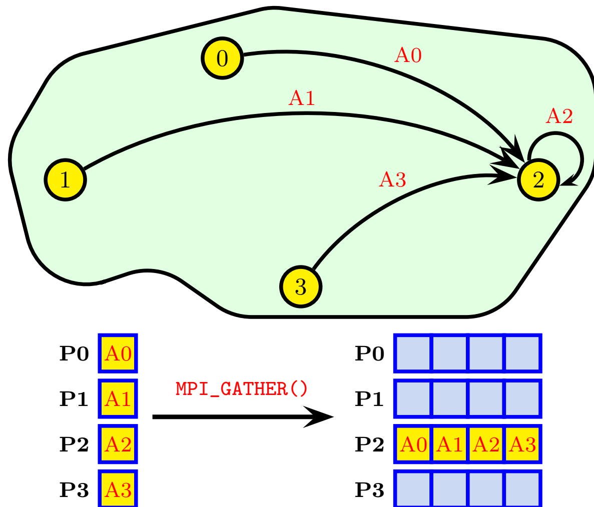


Figure 15 – Collecte : MPI_GATHER()

Collecte : MPI_GATHER()

```
<type et attribut>:: message_emis, message_recu
integer :: longueur_message_emis, longueur_message_recu
integer :: type_message_emis, type_message_recu
integer :: rang_dest, comm, code

call MPI_GATHER(message_emis,longueur_message_emis,type_message_emis,
                message_recu,longueur_message_recu,type_message_recu,rang_dest,comm,code)
```

1. Envoi de chacun des processus du communicateur comm, d'un message message_emis, de taille longueur_message_emis et de type type_message_emis.
2. Collecte de chacun de ces messages, par le processus rang_dest, à partir l'adresse message_recu, sur une longueur longueur_message_recu et avec le type type_message_recu.

Remarques :

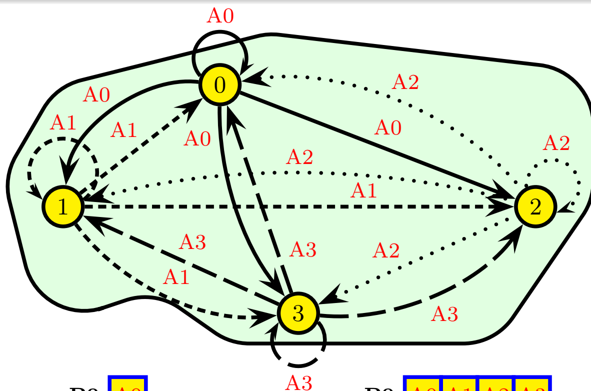
- Les couples (longueur_message_emis, type_message_emis) et (longueur_message_recu, type_message_recu) doivent être tels que les quantités de données envoyées et reçues soient égales.
- Les données sont collectées dans l'ordre des rangs des processus.

```fortran
program gather
  use mpi
  implicit none
  integer, parameter                 :: nb_valeurs=8
  integer                            :: nb_procs,rang,longueur_tranche,i,code
  real, dimension(nb_valeurs)        :: donnees
  real, allocatable, dimension(:)    :: valeurs

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

  longueur_tranche=nb_valeurs/nb_procs

  allocate(valeurs(longueur_tranche))

  valeurs(:)=(/(1000.+rang*longueur_tranche+i,i=1,longueur_tranche)/)
  print *,'Moi, processus ',rang,'envoie mon tableau valeurs : ',&
            valeurs(1:longueur_tranche)

  call MPI_GATHER(valeurs,longueur_tranche,MPI_REAL,donnees,longueur_tranche, &
                  MPI_REAL,2,MPI_COMM_WORLD,code)

  if (rang == 2) print *,'Moi, processus 2', 'j''ai reçu ',donnees(1:nb_valeurs)

  call MPI_FINALIZE(code)

end program gather
```

```
> mpiexec -n 4 gather
Moi, processus 1 envoie mon tableau valeurs :1003. 1004.
Moi, processus 0 envoie mon tableau valeurs :1001. 1002.
Moi, processus 2 envoie mon tableau valeurs :1005. 1006.
Moi, processus 3 envoie mon tableau valeurs :1007. 1008.

Moi, processus 2, j'ai reçu 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
```

4 – Communications collectives

4.6 – Collecte générale : MPI_ALLGATHER()



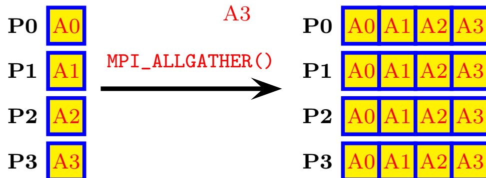


Figure 16 – Collecte générale : MPI_ALLGATHER()

Collecte générale : MPI_ALLGATHER()

Correspond à un MPI_GATHER() suivi d'un MPI_BCAST() :

```
<type et attribut>:: message_emis, message_recu
integer :: longueur_message_emis, longueur_message_recu
integer :: type_message_emis, type_message_recu
integer :: comm, code

call MPI_ALLGATHER(message_emis,longueur_message_emis,type_message_emis,
                   message_recu,longueur_message_recu,type_message_recu,comm,code)
```

1. Envoi de chacun des processus du communicateur comm, d'un message message_emis, de taille longueur_message_emis et de type type_message_emis.
2. Collecte de chacun de ces messages, par tous les processus, à partir l'adresse message_recu, sur une longueur longueur_message_recu et avec le type type_message_recu.

Remarques :

- Les couples (longueur_message_emis, type_message_emis) et (longueur_message_recu, type_message_recu) doivent être tels que les quantités de données envoyées et reçues soient égales.
- Les données sont collectées dans l'ordre des rangs des processus.

```
 1 program allgather
 2   use mpi
 3   implicit none
 4
 5   integer, parameter              :: nb_valeurs=8
 6   integer                         :: nb_procs,rang,longueur_tranche,i,code
 7   real, dimension(nb_valeurs)     :: donnees
 8   real, allocatable, dimension(:) :: valeurs
 9
10   call MPI_INIT(code)
11
12   call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
13   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
14
15   longueur_tranche=nb_valeurs/nb_procs
16   allocate(valeurs(longueur_tranche))
17
18   valeurs(:)=(/(1000.+rang*longueur_tranche+i,i=1,longueur_tranche)/)
19
20   call MPI_ALLGATHER(valeurs,longueur_tranche,MPI_REAL,donnees,longueur_tranche, &
21                      MPI_REAL,MPI_COMM_WORLD,code)
22
23   print *,'Moi, processus ',rang,', j''ai reçu ',donnees(1:nb_valeurs)'
24
25   call MPI_FINALIZE(code)
26
27 end program allgather
```

```
> mpiexec -n 4 allgather

Moi, processus 1, j'ai reçu 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
Moi, processus 3, j'ai reçu 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
Moi, processus 2, j'ai reçu 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
Moi, processus 0, j'ai reçu 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
```

4 – Communications collectives
4.7 – Collecte : MPI_GATHERV()

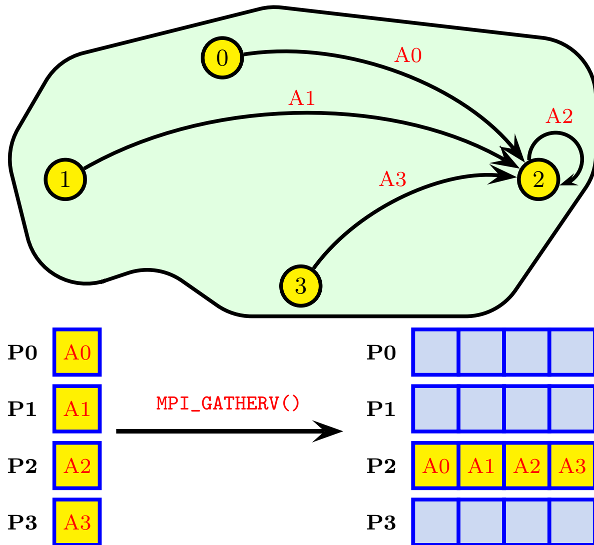


Figure 17 – Collecte : MPI_GATHERV()

Collecte "variable" : MPI_GATHERV()

Correspond à un MPI_GATHER() pour lequel la taille des messages varie :

```
<type et attribut>:: message_emis, message_recu
integer :: longueur_message_emis
integer :: type_message_emis, type_message_recu
integer, dimension(:) :: nb_elts_recus, deplts
integer :: rang_dest, comm, code

call MPI_GATHERV(message_emis,longueur_message_emis,type_message_emis,
                 message_recu,nb_elts_recus,deplts,type_message_recu,
                 rang_dest,comm,code)
```

Le ième processus du communicateur comm envoie au processus rang_dest, un message depuis l'adresse message_emis, de taille longueur_message_emis, de type type_message_emis, avec réception du message à l'adresse message_recu, de type type_message_recu, de taille nb_elts_recus(i) avec un déplacement de deplts(i).

Remarques :

- Les couples (longueur_message_emis, type_message_emis) du ième processus et (nb_elts_recus(i), type_message_recu) du processus rang_dest doivent être tels que les quantités de données envoyées et reçues soient égales.

```
1  program gatherv
2    use mpi
3    implicit none
4    integer, parameter                  :: nb_valeurs=10
5    integer                             :: reste, nb_procs, rang, longueur_tranche, i, code
6    real, dimension(nb_valeurs)         :: donnees
7    real, allocatable, dimension(:)     :: valeurs
8    integer, allocatable, dimension(:) :: nb_elements_recus,deplacements
9
10   call MPI_INIT(code)
11   call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
12   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
13
14   longueur_tranche=nb_valeurs/nb_procs
15   reste = mod(nb_valeurs,nb_procs)
16   if (reste > rang) longueur_tranche = longueur_tranche+1
17
18   ALLOCATE(valeurs(longueur_tranche))
19   valeurs(:) = (/(1000.+(rang*(nb_valeurs/nb_procs))+min(rang,reste)+i, &
20                 i=1,longueur_tranche)/)
21
22   PRINT *, 'Moi, processus ', rang,'envoie mon tableau valeurs : ',&
23            valeurs(1:longueur_tranche)
24
25   ALLOCATE(nb_elements_recus(nb_procs),deplacements(nb_procs))
26   IF (rang == 2) THEN
27     nb_elements_recus(1) = nb_valeurs/nb_procs
28     if (reste > 0) nb_elements_recus(1) = nb_elements_recus(1)+1
29     deplacements(1) = 0
30     DO i=2,nb_procs
31       deplacements(i) = deplacements(i-1)+nb_elements_recus(i-1)
32       nb_elements_recus(i) = nb_valeurs/nb_procs
33       if (reste > i-1) nb_elements_recus(i) = nb_elements_recus(i)+1
34     END DO
35   END IF
```

```
  CALL MPI_GATHERV (valeurs,longueur_tranche, MPI_REAL ,donnees,nb_elements_recus,&
       deplacements, MPI_REAL ,2, MPI_COMM_WORLD ,code)
  IF (rang == 2) PRINT *, 'Moi, processus 2 je recois', donnees(1:nb_valeurs)
  CALL MPI_FINALIZE (code)
end program gatherv
```

```
> mpiexec -n 4 gatherv

 Moi, processus  0 envoie mon tableau valeurs :  1001. 1002. 1003.
 Moi, processus  2 envoie mon tableau valeurs :  1007. 1008.
 Moi, processus  3 envoie mon tableau valeurs :  1009. 1010.
 Moi, processus  1 envoie mon tableau valeurs :  1004. 1005. 1006.

 Moi, processus 2 je reçois   1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
1009. 1010.
```

4 – Communications collectives

4.8 – Collectes et diffusions sélectives : MPI_ALLTOALL()

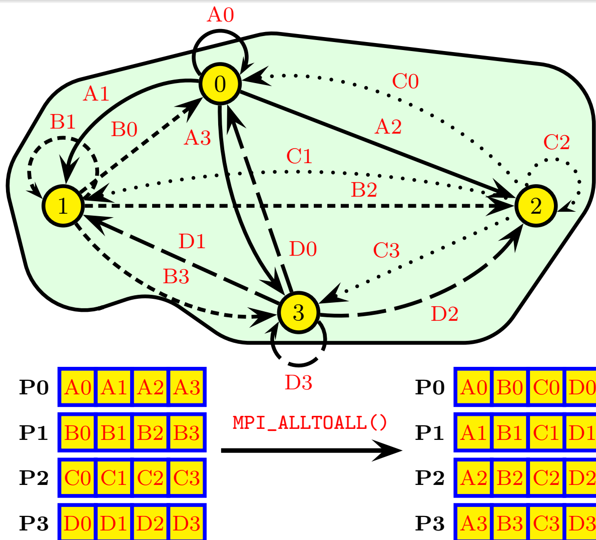


Figure 18 – Collecte et diffusion sélectives : MPI_ALLTOALL()

Collectes et diffusions sélectives : MPI_ALLTOALL()

```
<type et attribut>:: message_emis, message_recu
integer :: longueur_message_emis, longueur_message_recu
integer :: type_message_emis, type_message_recu
integer :: comm, code

call MPI_ALLTOALL(message_emis,longueur_message_emis,type_message_emis,
                  message_recu,longueur_message_recu,type_message_recu,comm,code)
```

Correspond à un MPI_ALLGATHER() où chaque processus envoie des données différentes : le ième processus envoie la jème tranche au jème processus qui le place à l'emplacement de la ième tranche.

Remarque :

- Les couples (longueur_message_emis, type_message_emis) et (longueur_message_recu, type_message_recu) doivent être tels que les quantités de données envoyées et reçues soient égales.

```
 1 program alltoall
 2   use mpi
 3   implicit none
 4
 5   integer, parameter                :: nb_valeurs=8
 6   integer                           :: nb_procs,rang,longueur_tranche,i,code
 7   real, dimension(nb_valeurs)       :: valeurs,donnees
 8
 9   call MPI_INIT(code)
10   call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
11   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
12
13   valeurs(:)=(/(1000.+rang*nb_valeurs+i,i=1,nb_valeurs)/)
14   longueur_tranche=nb_valeurs/nb_procs
15
16   print *,'Moi, processus ',rang,"j'ai envoyé mon tableau valeurs : ", &
17           valeurs(1:nb_valeurs)
18
19   call MPI_ALLTOALL(valeurs,longueur_tranche,MPI_REAL,donnees,longueur_tranche, &
20                     MPI_REAL,MPI_COMM_WORLD,code)
21
22   print *,'Moi, processus ',rang,', j''ai reçu ',donnees(1:nb_valeurs)
23
24   call MPI_FINALIZE(code)
25 end program alltoall
```

```
> mpiexec -n 4 alltoall
Moi, processus 1 j'ai envoyé mon tableau valeurs :
1009. 1010. 1011. 1012. 1013. 1014. 1015. 1016.
Moi, processus 0 j'ai envoyé mon tableau valeurs :
1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
Moi, processus 2 j'ai envoyé mon tableau valeurs :
1017. 1018. 1019. 1020. 1021. 1022. 1023. 1024.
Moi, processus 3 j'ai envoyé mon tableau valeurs :
1025. 1026. 1027. 1028. 1029. 1030. 1031. 1032.

Moi, processus 0, j'ai reçu 1001. 1002.  1009.  1010.  1017.  1018. 1025.  1026.
Moi, processus 2, j'ai reçu 1005. 1006.  1013.  1014.  1021.  1022. 1029.  1030.
Moi, processus 1, j'ai reçu 1003. 1004.  1011.  1012.  1019.  1020. 1027.  1028.
Moi, processus 3, j'ai reçu 1007. 1008.  1015.  1016.  1023.  1024. 1031.  1032.
```

4 – Communications collectives

4.9 – Réductions réparties

Réductions réparties

- Une réduction est une opération appliquée à un ensemble d'éléments pour en obtenir une seule valeur. Des exemples typiques sont la somme des éléments d'un vecteur `SUM(A(:))` ou la recherche de l'élément de valeur maximum dans un vecteur `MAX(V(:))`.
- MPI propose des sous-programmes de haut-niveau pour opérer des réductions sur des données réparties sur un ensemble de processus. Le résultat est obtenu sur un seul processus (`MPI_REDUCE()`) ou bien sur tous (`MPI_ALLREDUCE()`, qui est en fait équivalent à un `MPI_REDUCE()` suivi d'un `MPI_BCAST()`).
- Si plusieurs éléments sont concernés par processus, la fonction de réduction est appliquée à chacun d'entre eux.
- Le sous-programme `MPI_SCAN()` permet en plus d'effectuer des réductions partielles en considérant, pour chaque processus, les processus précédents du communicateur et lui-même.
- Les sous-programmes `MPI_OP_CREATE()` et `MPI_OP_FREE()` permettent de définir des opérations de réduction personnelles.

Opérations

Table 3 – Principales opérations de réduction prédéfinies (il existe aussi d'autres opérations logiques)

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

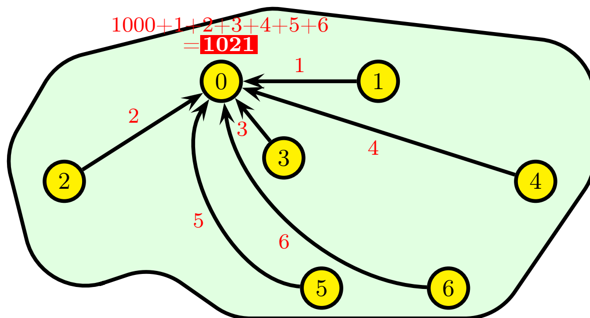


Figure 19 – Réduction répartie : `MPI_REDUCE()` avec l'opérateur somme

Réductions réparties : MPI_REDUCE()

```
<type et attribut>:: message_emis, message_recu
integer :: longueur, type, rang_dest
integer :: operation, comm, code

call MPI_REDUCE(message_emis,message_recu,longueur,type,operation,rang_dest,comm,code)
```

1. Réduction répartie des éléments situés à partir de l'adresse message_emis, de taille longueur, de type type, pour les processus du communicateur comm,
2. Ecrit le résultat à l'adresse message_recu pour le processus de rang rang_dest.

```
1  program reduce
2    use mpi
3    implicit none
4    integer :: nb_procs,rang,valeur,somme,code
5
6    call MPI_INIT(code)
7    call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
8    call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
9
10   if (rang == 0) then
11      valeur=1000
12   else
13      valeur=rang
14   endif
15
16   call MPI_REDUCE(valeur,somme,1,MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORLD,code)
17
18   if (rang == 0) then
19      print *,'Moi, processus 0, j''ai pour valeur de la somme globale ',somme
20   end if
21
22   call MPI_FINALIZE(code)
23 end program reduce
```

```
> mpiexec -n 7 reduce

Moi, processus 0, j'ai pour valeur de la somme globale 1021
```

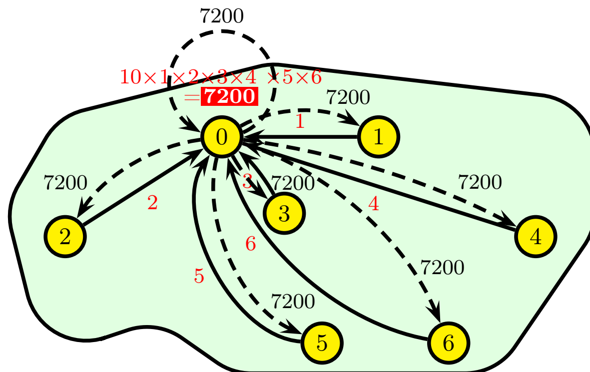


Figure 20 – Réduction répartie avec diffusion du résultat : `MPI_ALLREDUCE` (utilisation de l'opérateur produit)

Réductions réparties avec diffusion du résultat : MPI_ALLREDUCE()

```
<type et attribut>:: message_emis, message_recu
integer :: longueur, type
integer :: operation, comm, code

call MPI_ALLREDUCE(message_emis,message_recu,longueur,type,operation,comm,code)
```

1. Réduction répartie des éléments situés à partir de l'adresse message_emis, de taille longueur, de type type, pour les processus du communicateur comm,
2. Ecrit le résultat à l'adresse message_recu pour tous les processus du communicateur comm.

```
1  program allreduce
2
3    use mpi
4    implicit none
5
6    integer :: nb_procs,rang,valeur,produit,code
7
8    call MPI_INIT(code)
9    call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
10   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
11
12   if (rang == 0) then
13      valeur=10
14   else
15      valeur=rang
16   endif
17
18   call MPI_ALLREDUCE(valeur,produit,1,MPI_INTEGER,MPI_PROD,MPI_COMM_WORLD,code)
19
20   print *,'Moi, processus ',rang,', j''ai reçu la valeur du produit global ',produit
21
22   call MPI_FINALIZE(code)
23
24 end program allreduce
```

```
> mpiexec -n 7 allreduce

Moi, processus 6, j'ai reçu la valeur du produit global 7200
Moi, processus 2, j'ai reçu la valeur du produit global 7200
Moi, processus 0, j'ai reçu la valeur du produit global 7200
Moi, processus 4, j'ai reçu la valeur du produit global 7200
Moi, processus 5, j'ai reçu la valeur du produit global 7200
Moi, processus 3, j'ai reçu la valeur du produit global 7200
Moi, processus 1, j'ai reçu la valeur du produit global 7200
```

3.6.

Compilation et exécution d'un programme MPI

Compilation et exécution MPI

Charger respectivement pour le Fortran et C dans le code les bibliothèques suivantes :

use mpi

#include "mpi.h"

Pour compiler et exécuter on peut par exemple charger les modules :

```
module  add  intel/compiler  openmpi/intel
```

ou

```
module add intel-mpi
```

On utilise respectivement pour le Fortran et C les compilateurs mpif90 et mpicc

Lancer l'exécutable sur nc coeurs de calcul (nc processus)

```
mpirun  -n  nc  ./nom_executable
```

4.
OpenMP versus MPI

CALCUL PARALLÈLE

MPI vs OpenMP

MPI utilise un schéma à mémoire distribuée

OpenMP utilise un schéma à mémoire partagée

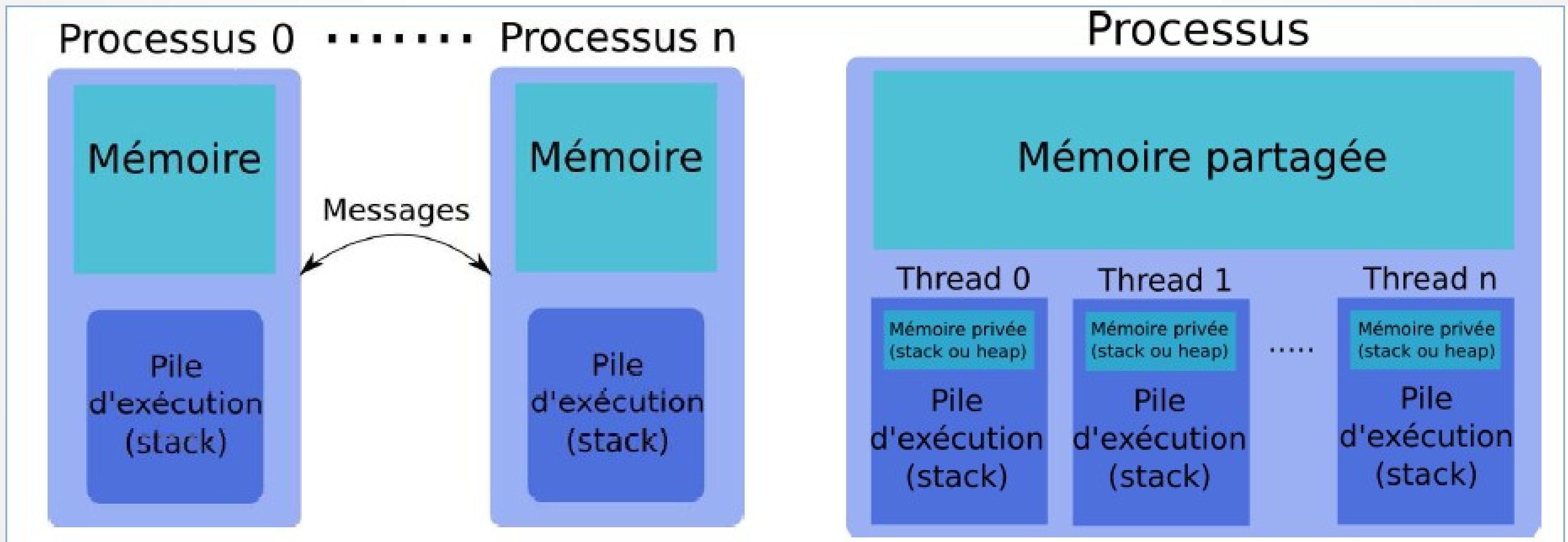


Schéma MPI **Schéma OpenMP**

MPI vs OpenMP

Ce sont des modèles de programmation adaptées à deux architectures parallèles différentes :

MPI

- **modèle de programmation à mémoire distribuée.**
- **la communication entre les processus est explicite et sa gestion est à la charge de l'utilisateur.**
- **la parallélisation d'un code est plus « intuitive ».**
- **on peut obtenir de très bonnes performances sur des milliers de coeurs**

OpenMP

- **modèle de programmation à mémoire partagée, chaque thread a une vision globale de la mémoire.**
- **Si on se limite à quelques boucles parallélisées, on peut obtenir des performances avec un moindre côut de développement**
- **selon les codes, ca peut être plus compliqué à développer que MPI**
- **limitations des performances sur un nombre de threads**

5.
Programmation hybride MPI/OpenMP

Principe de la programmation hybride MPI/OpenMP

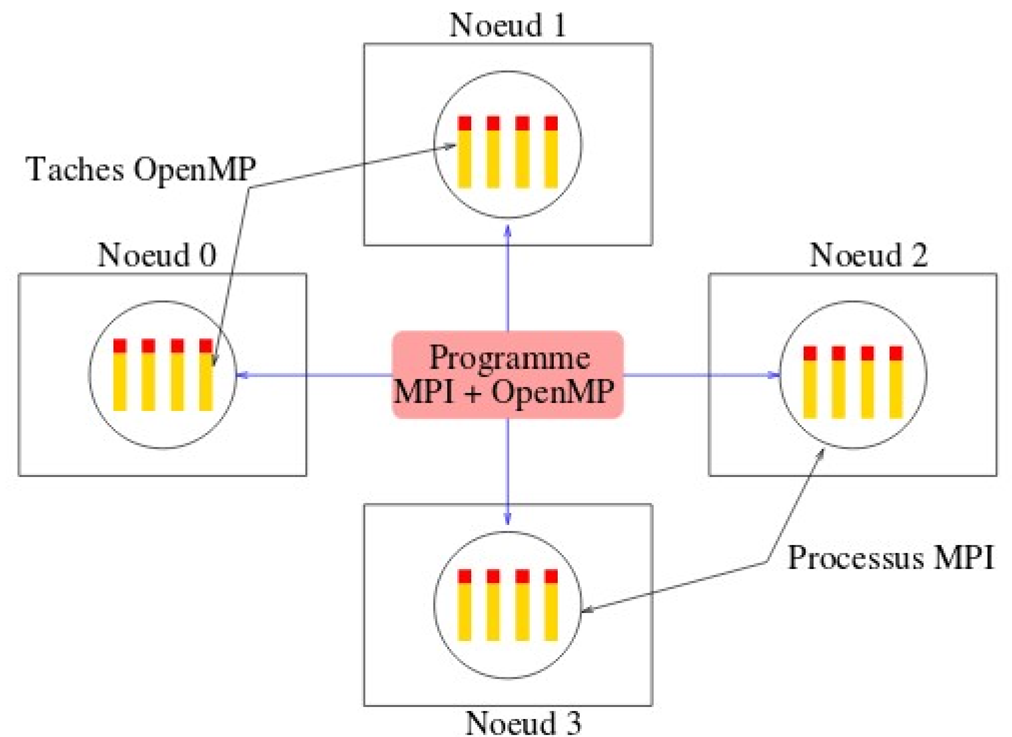
La programmation hybride parallèle consiste à mélanger plusieurs paradigmes de programmation parallèle dans le but de tirer parti des avantages des différentes approches.

Généralement, MPI est utilisé au niveau des processus et un autre paradigme comme OpenMP à l'intérieur de chaque processus.

CALCUL PARALLÈLE

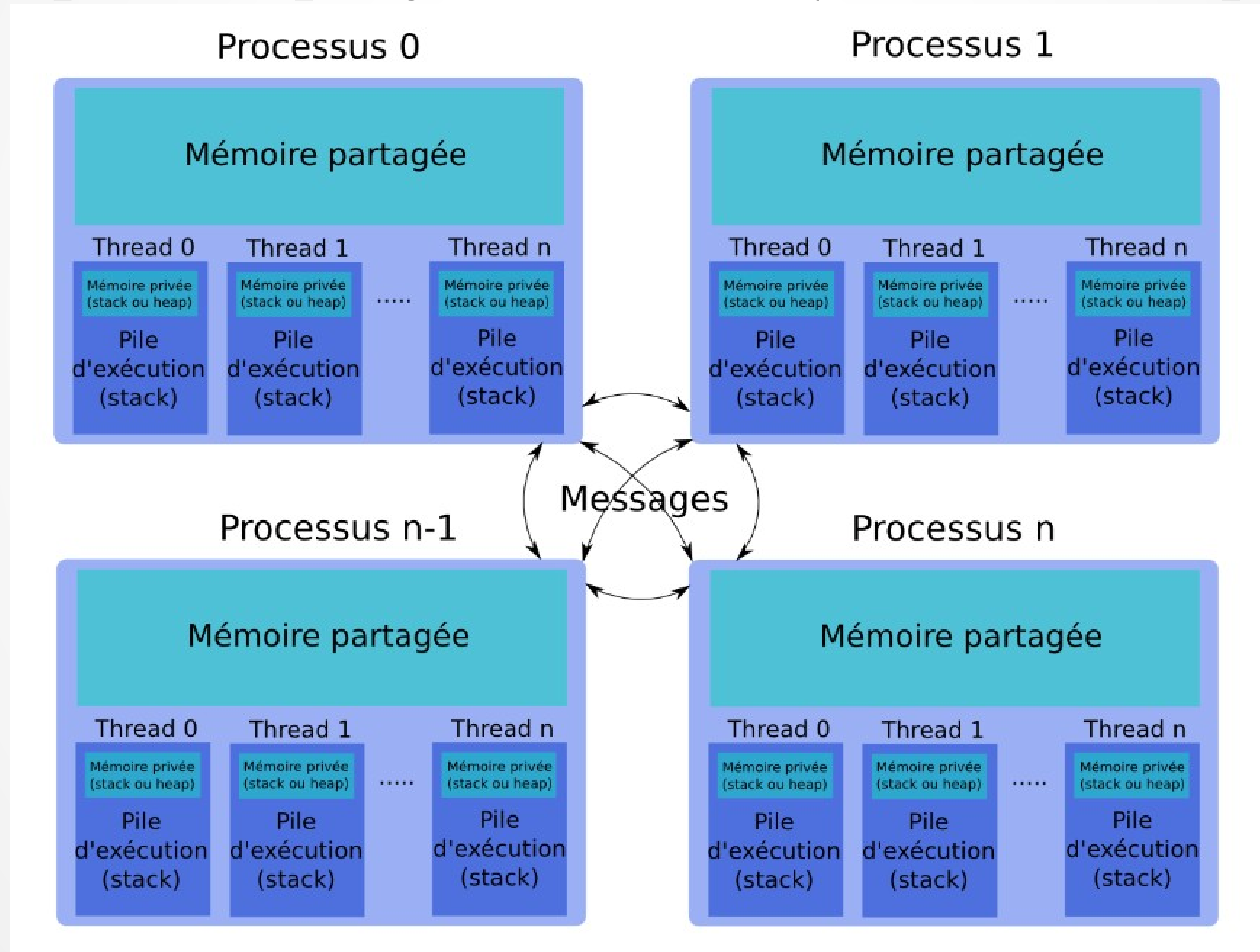
Principe de la programmation hybride MPI/OpenMP

Sur une grappe de machines indépendantes (nœuds) multicoeurs à mémoire partagée, la mise en œuvre d'une parallélisation à deux niveaux MPI et OpenMP dans un même programme peut être un atout majeur pour les performances parallèles du code.



CALCUL PARALLÈLE

Principe de la programmation hybride MPI/OpenMP



6.

Mesures de performances

Mesure du temps réel d'exécution, elapsed time

Temps réel écoulé (ou elapsed time) est le temps depuis le début du programme jusqu'à la fin. Le temps réel écoulé englobe les temps de calcul, de communications (pour MPI) et d'entrées/sorties (IO)

On mesure le temps écoulé entre le début du calcul (séquentiel ou parallèle) et la terminaison de la dernière tâche (séquentiel ou parallèle)

Possibilité de mettre des directives de mesures de temps directement dans le code ou alors de lancer les exécutables précédés de la commande time et mesurer le temps real

Certains logiciels de « profiling » de codes permettent de mesurer les temps de calcul passées dans chaque portion du code, les temps de calcul par rapport aux temps de communications, ...

Mesure de la performance d'un programme parallèle

Question naturelle : quel est le gain de la parallélisation en temps de calcul quand on augmente le nombre de coeurs (processus MPI) et/ou le nombre de threads (OpenMP) ?

Diviser le problème ↔ diviser le temps de calcul ?

Les performances d'un code parallèle vont dépendre du nombre de coeurs et des communications entre processsus (mouvements de données, attentes, synchronisations)

Un programme séquentiel « rapide » peut s'avérer, une fois parallélisé, peu efficace ; inversement, un programme séquentiel "moins rapide" peut s'avérer mieux parallélisable

Les définitions qui vont suivre en matière de performance seront décrites dans le cas MPI sur les coeurs (processus) mais elles s'appliquent également dans le cas OpenMP sur les threads

Accélération

L'accélération d'un programme parallèle (ou speedup) représente le gain en rapidité d'exécution obtenu par son exécution sur plusieurs cœurs de calcul.

On la mesure par le rapport entre le temps d'exécution du programme séquentiel et le temps d'exécution sur C cœurs de calcul.

Soit T(C) le temps d'exécution sur C cœurs

L'accélération A(C) est définie comme étant :

$$A(C) = T(1) / T(C) \quad (C = 1, 2, 3, \ldots)$$

L'efficacité E(C) est définie comme étant :

$$E(C) = A(C) / C$$

CALCUL PARALLÈLE

Accélération

Appréciation de l'accélération :

- accélération linéaire : parallélisation optimale

- accélération sur-linéaire : attention !

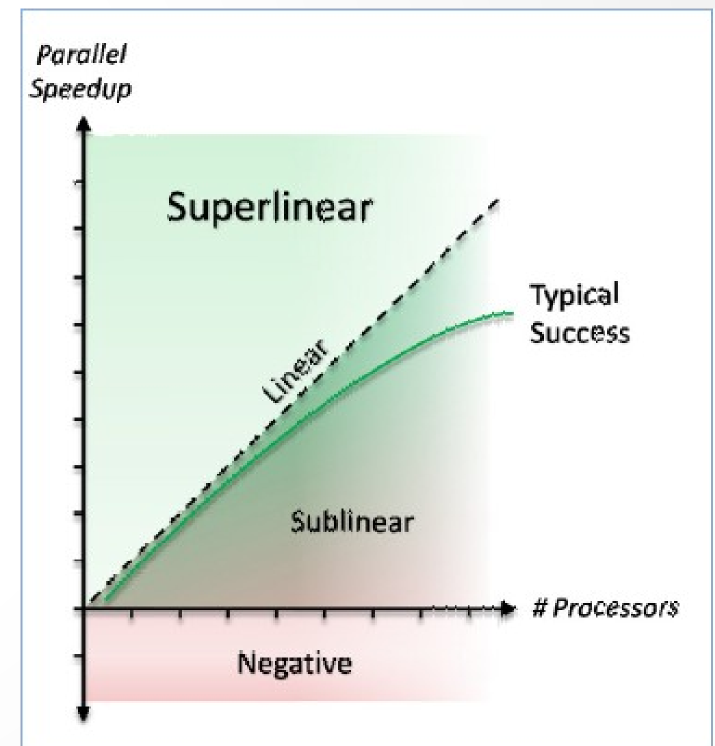
- accélération sub-linéaire : ralentissement dû au parallélisation

Pour une accélération parallèle parfaite on obtient :

$T(C) = T(1) / C$

$A(C) = T(1) / T(C) = T(1) / (T(1) / C) = C$

$E(C) = A(C) / C = C / C = 1$



La loi d'Amdahl

Décomposition du temps d'exécution d'un programme

T(PS) : temps de calcul en séquentiel de la partie non parallélisable du code

T(PP) : temps de calcul en séquentiel de la partie parallélisable du code

Sur un cœur de calcul (en séquentiel) , on a :

$T(1) = T(PS)+T(PP)$

Sur C cœurs de calcul, pour une parallélisation optimale, on a :

- **$T(C) = T(PS) + (T(PP) / C)$**
- **Si $C \to \infty$, on a $A(C) = T(1) / T(C) \sim T(1) / T(PS)$**

  Ce qui signifie que l'accélération est toujours limitée par la partie non-parallélisable du programme
- **Si $T(PS) \to 0$, on a $T(PP) \sim T(1)$, $A(C) = T(1) / T(C) \sim C$**

  Ce qui signifie que l'accélération est linéaire

CALCUL PARALLÈLE

La loi d'Amdahl

Exemple où la partie parallélisable du code représente 80 % du temps de calcul en séquentiel :

T(1) = (T(partie parallèle) = 80)

+ T(partie non parallèle) = 20)

Sur C coeurs, on obtient pour une parallélisation optimale :

$T(C) = (80 / C) + 20$

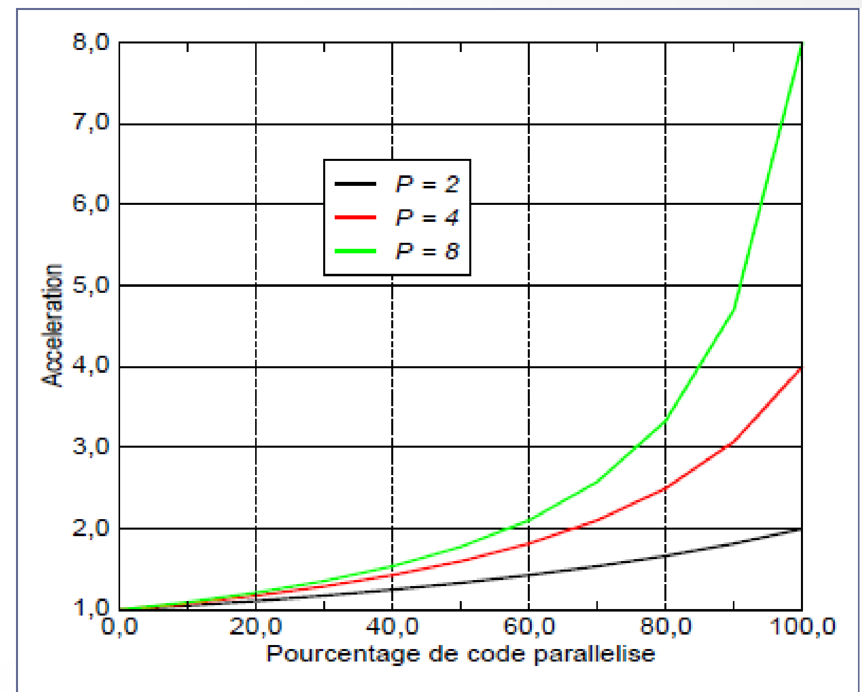
Quel que soit C, $T(C) > 20$

$A(C) = T(1) / T(C)$

$= 100 / ((80 / C) + 20) < 100/20 = 5$

$E(C) = (A(C) / C) < (5 / C) \rightarrow 0$!!!

quand $C \rightarrow +\infty$



Dans le schéma, p = C

Passage à l'échelle - Scalabilité

On a vu avec la loi d'Amdahl que la performance augmente théoriquement lorsque l'on ajoute des cœurs de calcul.

- Comment augmente-t-elle en réalité ?
- Y a-t-il des facteurs limitants (goulet d'étranglement ...) ?
- Augmente-t-elle à l'infini ?

Le passage à l'échelle (scalabilité) d'un programme parallèle désigne l'augmentation des performances obtenues lorsque l'on ajoute des cœurs de calcul.

Obstacles à la scalabilité :

- Synchronisations
- Algorithmes ne passant pas à l'échelle (complexité en opérations, complexité en communications)

Passage à l'échelle - Scalabilité

Scalabilité forte : on fixe la taille du problème et on augmente le nombre de coeurs

- **Relative au speedup**
- **Si on a une hyperbole : scalabilité forte parfaite**

  On augmente le nombre de coeurs pour calculer plus vite

Scalabilité faible : on augmente la taille du problème avec le nombre de coeurs

- **Le problème est à taille constante par coeur**
- **Si le temps de calcul est constant : scalabilité faible parfaite**

  On augmente le nombre de coeurs pour résoudre des problèmes de plus grande taille

Speedup et scalabilité en mesure relative

Les mesures de speedup et scalabilité vues précedemment sont absolues car rapportées par rapport à l'exécution séquentielle sur 1 coeur de calcul.

Pour un problème de grande taille, pour des raisons de RAM, il n'est pas possible de faire tourner le code sur 1 coeur de calcul, par conséquent on défint ncmin comme étant le nombre minimum de coeurs sur lesquels le code peut tourner et on mesure des performances relatives à partir d'un nombre de coeurs $C \geq$ ncmin

MERCI POUR VOTRE ATTENTION

Khodor.Khadra@math.u-bordeaux.fr