

CALCUL PARALLÈLE

Initiation au calcul parallèle

*Khodor.Khadra@math.u-bordeaux.fr
janvier, février, mars 2017*



MCIA

CALCUL PARALLÈLE

- 1. Généralités sur la parallélisation**
- 2. Programmation OpenMP - Généralités**
- 3. Programmation MPI**
 - 3.1. Généralités**
 - 3.2. Environnement**
 - 3.3. Communications point à point**
 - 3.4. Phénomène de « deadlock »**
 - 3.5. Communications collectives**
 - 3.6. Compilation et exécution d'un programme**
- 4. OpenMP versus MPI**
- 5. Programmation hybride MPI/OpenMP**
- 6. Mesures de performance**

CALCUL PARALLÈLE

1.

Généralités sur la
parallélisation

CALCUL PARALLÈLE

Qu'est ce que la parallélisation ?

C'est un ensemble de techniques logicielles et matérielles permettant l'**exécution simultanée de séquences d'instructions « indépendantes » sur plusieurs cœurs de calcul**

CALCUL PARALLÈLE

Pourquoi paralléliser ?

La première question à se poser, c'est savoir si la parallélisation de l'application est nécessaire

Ecrire un programme séquentiel est déjà du travail, souvent difficile; la parallélisation le rendra encore plus dur

Il existe toujours des applications scientifiques qui consomment "trop" de ressources en temps de calcul et/ou en mémoire

CALCUL PARALLÈLE

Pourquoi paralléliser ?

Les temps de calcul en séquentiel sont extrêmement élevés :

- Il y a urgence à **obtenir des résultats dans des délais raisonnables**
- les centres de ressources de calcul **ne permettent pas de monopoliser les nœuds de calcul** pendant des semaines, voire des mois

La taille des données du problème à résoudre devient très élevée et la mémoire des nœuds ne permet plus de faire tourner le logiciel de calcul sur un seul cœur de calcul (donc en mode séquentiel) même en exploitant toute la mémoire vive du nœud de calcul

La seule solution, pour des raisons techniques ou économiques, reste la parallélisation

CALCUL PARALLÈLE

Bénéfice de la parallélisation

Exécution plus rapide du programme (gain en temps de restitution) en distribuant le travail sur différents cœurs de calcul

Résolution de problèmes avec un nombre de degré de libertés très élevé sur le domaine global (plus de ressources matérielles accessibles, notamment la mémoire)

CALCUL PARALLÈLE

Remarques importantes

Bien optimiser le code séquentiel avant de se lancer dans la parallélisation

Paralléliser un code séquentiel ne consiste pas à reécrire le code de calcul :

- ne pas dénaturer les algorithmes de calcul en terme de stabilité et de convergence
- garder le corps des instructions de calcul du code séquentiel
- modifier les boucles de calcul qui parcourent cette fois des données locales de tableaux
- de façon judicieuse placer les instructions de parallélisation au bon endroit pour que lorsqu'elles sont omises, on retrouve les instructions de calcul du code séquentiel

Bien s'assurer que le code parallèle sur plusieurs cœurs de calcul reproduise des résultats « identiques » ou le plus proches possible que ceux obtenus sur 1 cœur

CALCUL PARALLÈLE

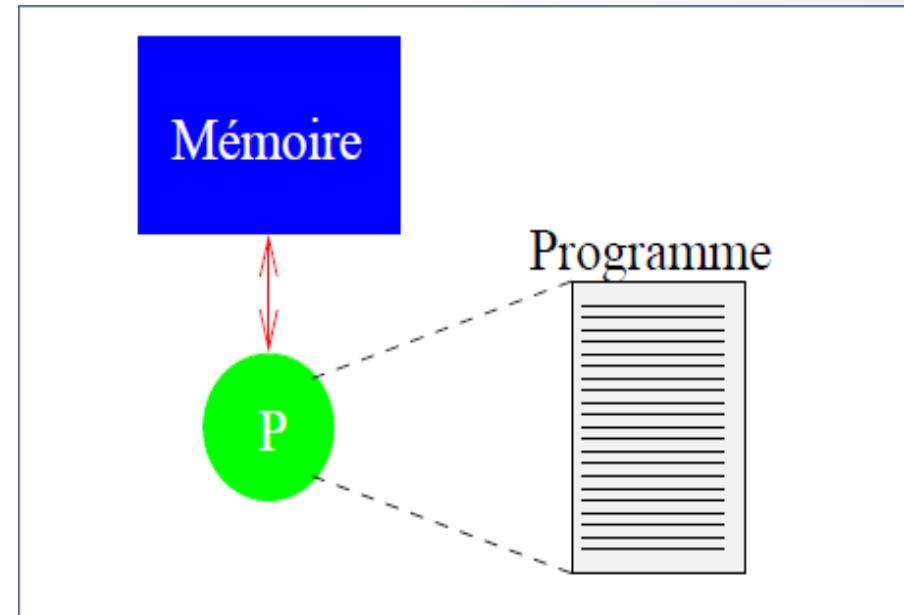
Petit rappel en mode séquentiel

Programme écrit dans un langage classique (Fortran, C, C++, ...)

Le programme est exécuté par un et un seul processus

Toutes les variables du programme sont allouées dans la mémoire allouée au processus

Un processus s'exécute sur un cœur d'un processeur physique de la machine

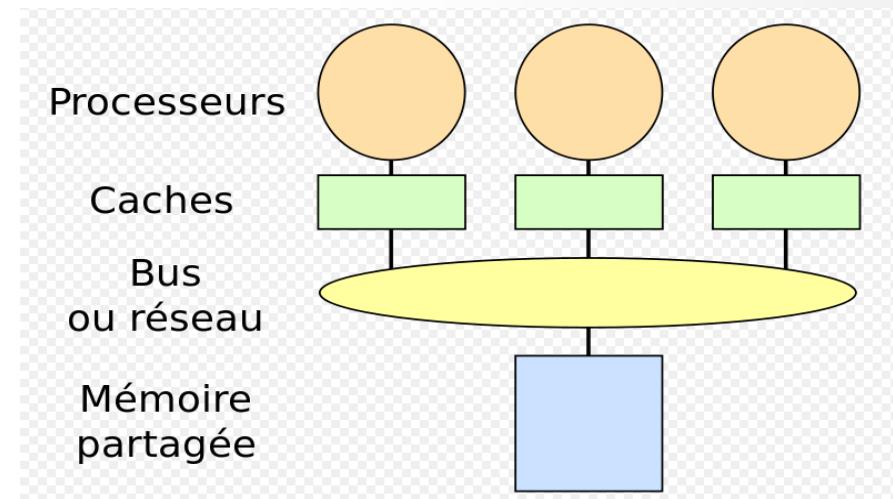


CALCUL PARALLÈLE

Mémoire partagée

Une même zone de la mémoire vive est accédée par plusieurs processus. C'est le comportement de la mémoire de threads issus d'un même processus.

Symmetric shared memory multiprocessor (**SMP**) : architecture parallèle qui consiste à multiplier les processeurs identiques au sein d'un noeud, de manière à augmenter la puissance de calcul avec **une mémoire unique partagée**



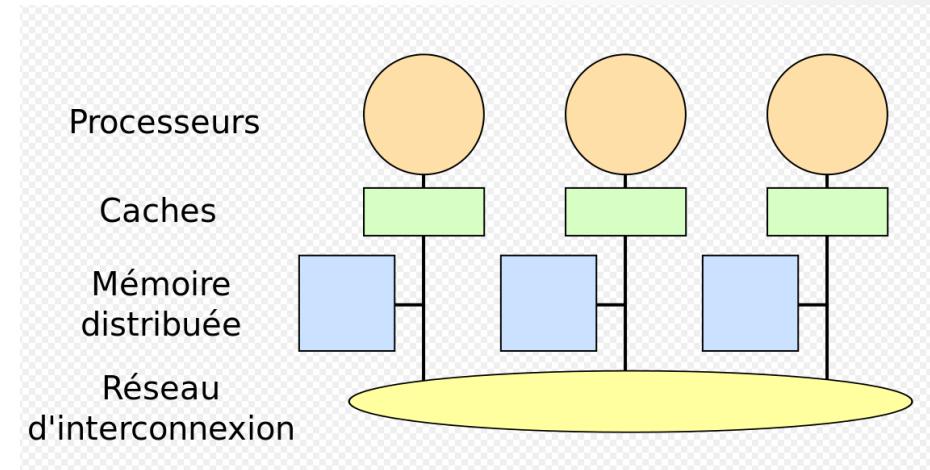
Utilisation de la **bibliothèque OpenMP**
(Open Multi-Processing)

CALCUL PARALLÈLE

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 système **NUMA** (Non Uniform Memory Access ou Non Uniform Memory Architecture, signifiant respectivement accès mémoire non uniforme et architecture mémoire non uniforme) est un système multiprocesseur dans lequel les zones mémoire sont séparées et placées en différents endroits



Un **réseau de communication** relie les différents nœuds

CALCUL PARALLÈLE

Mémoire distribuée

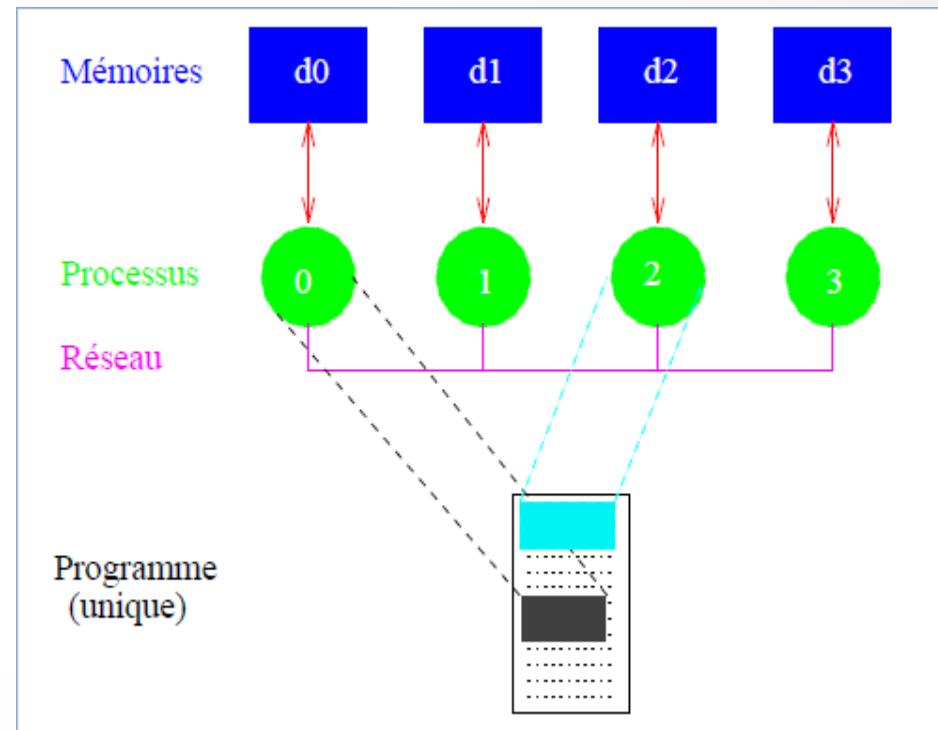
Programme écrit dans un langage classique (Fortran, C, C++, ...)

Le même programme est exécuté par tous les processus selon le modèle SMPD (Single Program Multiple Data)

Bonne règle : un seul processus par cœur

Toutes les variables du programme sont privées et résident dans la mémoire locale allouée à chaque processus

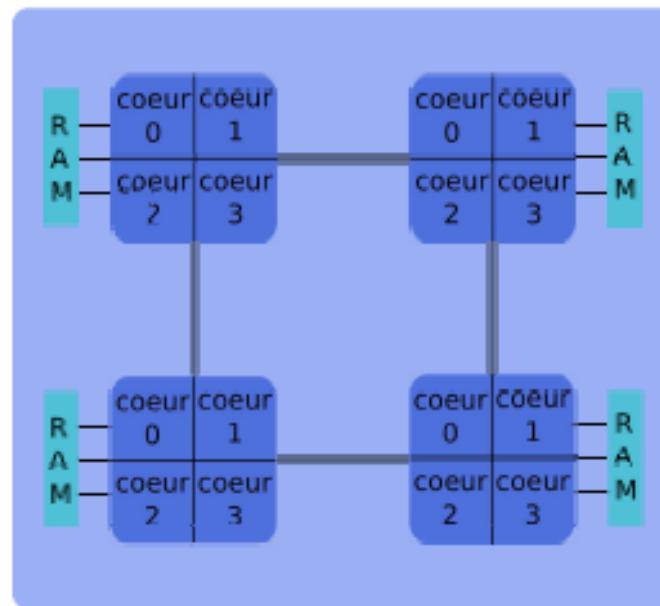
Des données sont échangées entre plusieurs processus via des appels utilisant la bibliothèque MPI



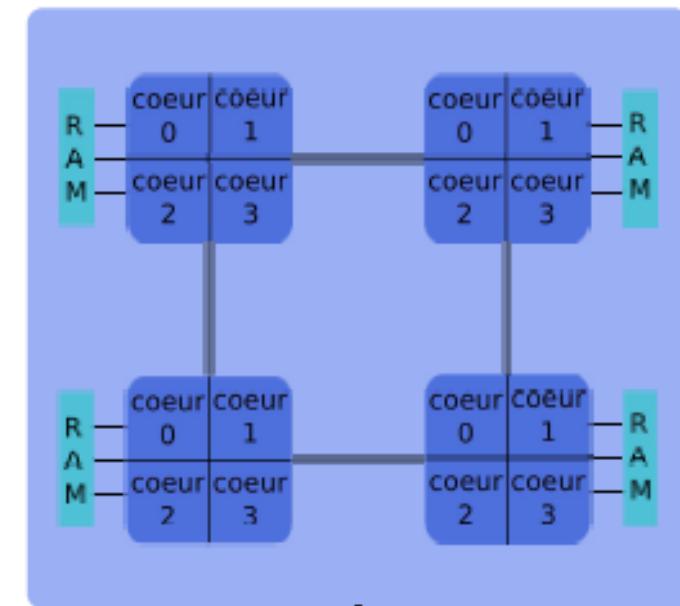
CALCUL PARALLÈLE

Cluster de calcul à mémoire distribuée

Noeud 0



Noeud n



Réseau

CALCUL PARALLÈLE

2.

Programmation OpenMP Généralités

CALCUL PARALLÈLE

Présentation d'OpenMP

Paradigme de parallélisation pour architecture à mémoire partagée basé sur des directives à insérer dans le code source (C, C++, Fortran).

OpenMP est constitué d'un jeu de directives, d'une bibliothèque de fonctions et d'un ensemble de variables d'environnement.

OpenMP fait partie intégrante de tout compilateur Fortran/C/C++ récent.

CALCUL PARALLÈLE

**Modèle de programmation multi-tâches
sur architecture à mémoire partagée**

Plusieurs tâches (threads) s'exécutent en parallèle

La mémoire est partagée (physiquement ou virtuellement)

Les communications entre tâches se font par lectures et écritures dans la mémoire partagée

Les processeurs multi-coeurs partagent une mémoire commune

Les tâches peuvent être attribuées à des coeurs distincts

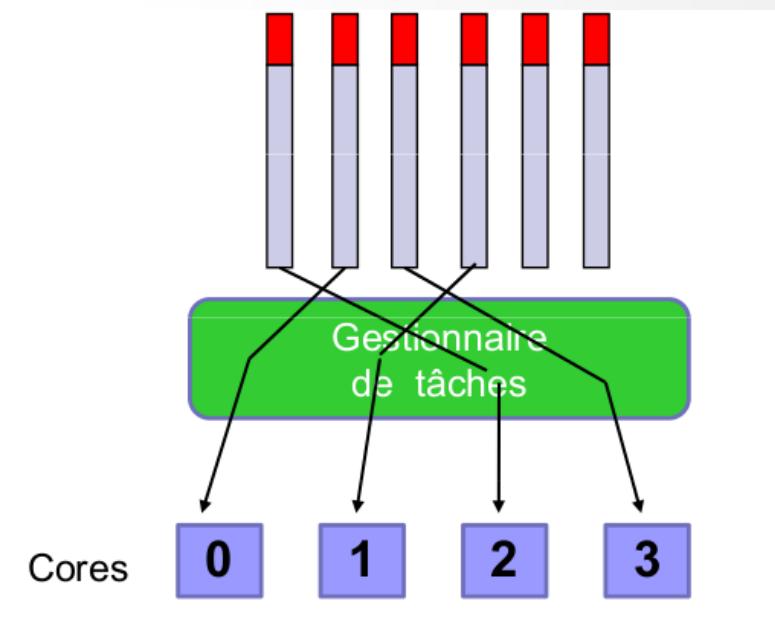
CALCUL PARALLÈLE

Exécution d'un programme OpenMP sur un nœud multicoeur

Un programme OpenMP est exécuté par un processus unique (sur un ou plusieurs coeurs)

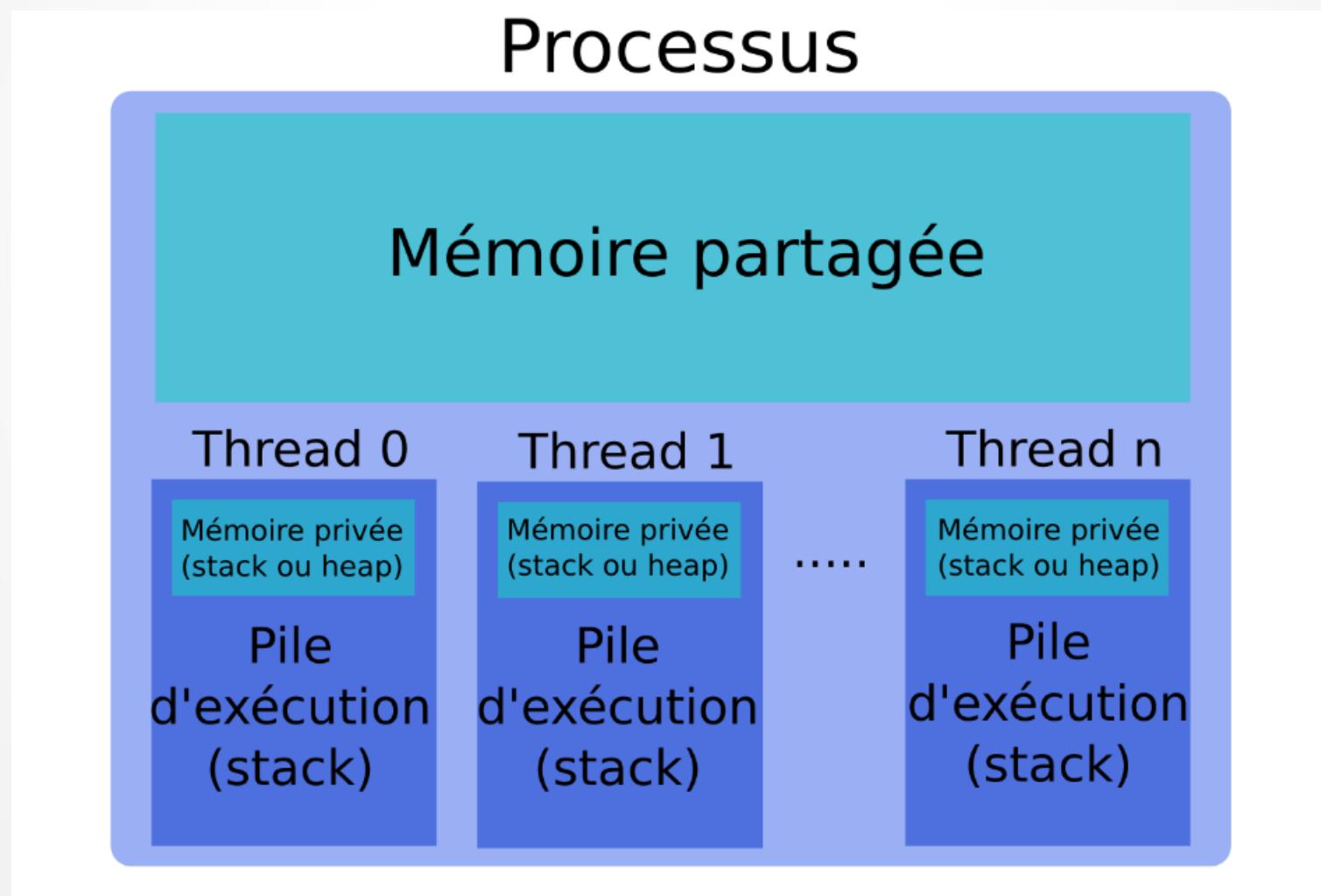
Les threads accèdent aux mêmes ressources que le processus.

Le gestionnaire de tâches du système d'exploitation affecte les tâches aux coeurs



CALCUL PARALLÈLE

Schéma de principe OpenMP



CALCUL PARALLÈLE

Construction d'une région parallèle OpenMP

Exemple en Fortran :

PROGRAM example

!\$USE OMP_LIB

! Declaration des variables

! Partie de code sequentiel

!\$OMP PARALLEL

! Zone parallele executee par tous les threads

\$OMP END PARALLEL

! Partie de code séquentiel

END PROGRAM example

Pour une meilleure efficacité, limiter le nombre d'ouvertures de zones parallèles

CALCUL PARALLÈLE

Directives DO et PARALLEL DO OpenMP

Exemple en Fortran :

```
Program loop
implicit none
integer, parameter :: n=1024
integer :: i, j
real, dimension(n, n) :: tab
!$OMP PARALLEL
...
      ! Code répliqué
!$OMP DO
do j=1, n    ! Boucle partagée
  do i=1, n    ! Boucle répliquée
    tab(i, j) = i*j
  end do
end do
!$OMP END DO
!$OMP END PARALLEL
end program loop
```

```
Program parallelloop
implicit none
integer, parameter :: n=1024
integer :: i, j
real, dimension(n, n) :: tab
!$OMP PARALLEL DO
do j=1 n    ! Boucle partagée
  do i=1, n    ! Boucle répliquée
    tab(i, j) = i*j
  end do
end do
!$OMP END PARALLEL DO
end program parallelloop
```

PARALLEL DO est une fusion des 2 directives

Attention : END PARALLEL DO inclut une barrière de synchronisation

CALCUL PARALLÈLE

Threads OpenMP

Définition du nombre de threads via la variable d'environnement **OMP_NUM_THREADS**

Les threads sont numérotées

- le nombre de threads n'est pas nécessairement égale au nombre de coeurs
- le thread de numéro 0 est la tâche maître

OMP_GET_NUM_THREADS() : nombre de threads

OMP_GET_THREAD_NUM() : numéro de la thread

OMP_GET_MAX_THREADS() : nombre max de threads

CALCUL PARALLÈLE

Compilation et exécution OpenMP

Option de compilation : **-openmp**

Définir le nombre de threads avant exécution

```
export OMP_NUM_THREADS=2
```

Lancer l'exécutable :

```
./monexec
```

CALCUL PARALLÈLE

Avantages de la programmation OpenMP

Simplicité de mise en œuvre et facilité de programmation

La parallélisation ne dénature pas le code ; une seule version du code à gérer pour la version séquentielle et parallèle

On peut gagner en temps de calcul avec quelques directives OpenMP très faciles à implémenter

Gestion de « threads » transparente et portable

CALCUL PARALLÈLE

Inconvénients et limitations de la programmation OpenMP

Problème de localité des données

Mémoire partagée mais non hiérarchique

Les surcoûts dus au partage du travail et à la création/gestion des threads peuvent se révéler importants, particulièrement lorsque la granularité du code parallèle est petite/faible (boucles de petites tailles de tableaux)

Passage à l'échelle limité (efficacité sur plusieurs coeurs), parallélisme modéré : dans la pratique, on observe une extensibilité limitée des codes (en tout cas inférieure à celle du même code parallélisé avec MPI), même lorsqu'ils sont bien optimisés. Plus la granularité du code est fine, plus ce phénomène s'accentue.

CALCUL PARALLÈLE

3.

Programmation MPI

CALCUL PARALLÈLE

3.1.

Généralités sur MPI

CALCUL PARALLÈLE

Echanges de messages

Les messages échangés sont interprétés et gérés par un environnement qui peut être comparé au courrier postal, à la messagerie électronique, ...

Le message est envoyé à une adresse déterminée

Le processus récepteur doit pouvoir classer et interpréter les messages qui lui ont été adressés

L'environnement en question est MPI (Message Passing Interface). Une application MPI est un ensemble de processus autonomes exécutant chacun leur propre code et communiquant via des appels à des sous-programmes de la bibliothèque MPI

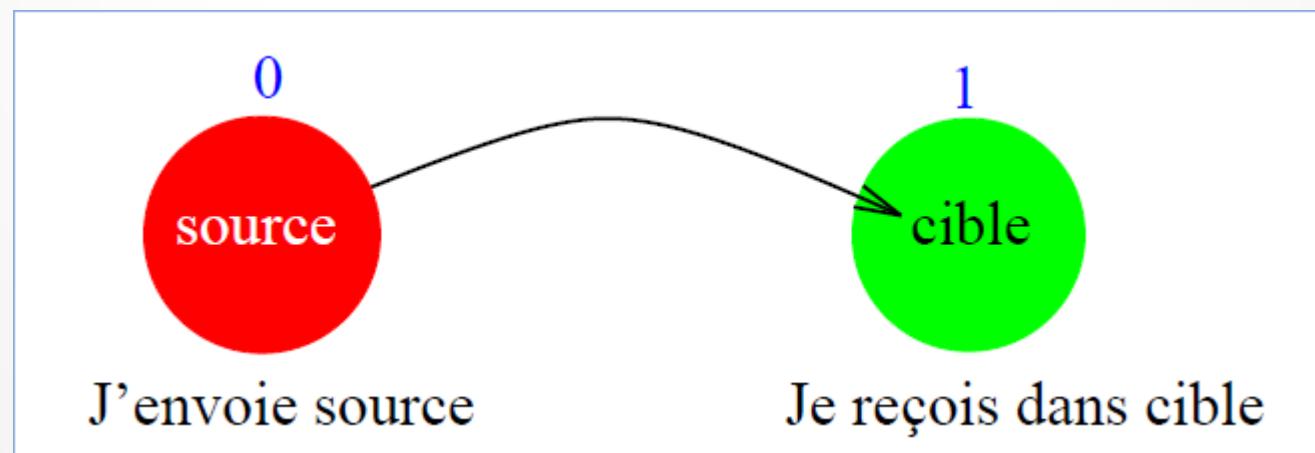
Une parallélisation efficace doit minimiser les communications par rapport aux calculs

CALCUL PARALLÈLE

Echanges de messages

Si un message est envoyé à un processus, celui-ci doit ensuite le recevoir

Un message est constitué de paquets de données transitant du processus **émetteur** au processus **récepteur**

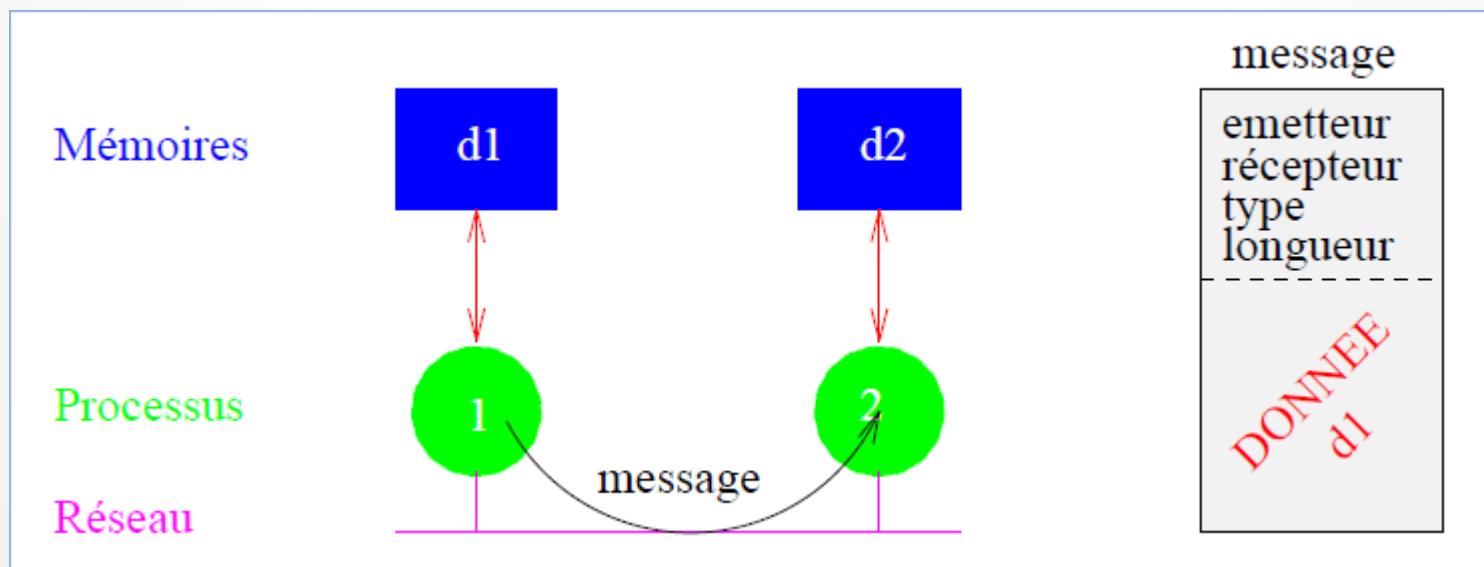


CALCUL PARALLÈLE

Echanges de messages

En plus des données (variables scalaires, tableaux, etc.) à transmettre, un message doit contenir les informations suivantes :

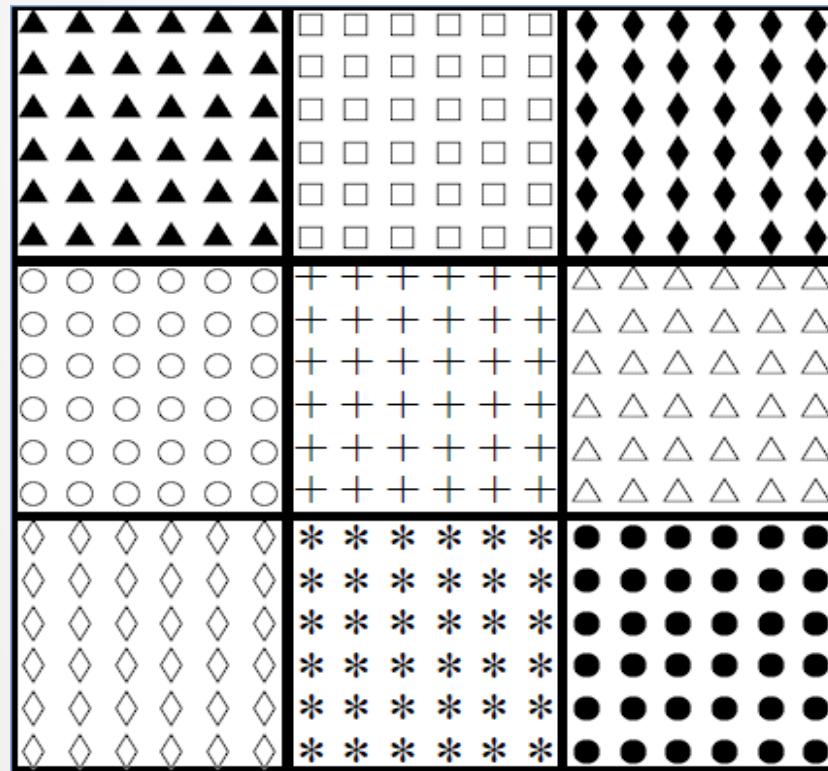
- l'identificateur du processus émetteur
- le type de la donnée
- sa longueur
- l'identificateur du processus récepteur



CALCUL PARALLÈLE

Décomposition de domaine

Un schéma que l'on rencontre très souvent avec MPI est la décomposition de domaine. Chaque processus possède une partie du domaine global, et effectue principalement des échanges avec ses processus voisins.



Découpage en sous-domaines

CALCUL PARALLÈLE

Résolution numérique d'un problème sur un maillage global avec échanges de messages MPI

On découpe le domaine global de résolution en N sous-domaines de tailles le plus homogènes possible

On souhaite faire tourner le code de calcul sur N cœurs

On définit une topologie de numérotation des sous-domaines et ceux-ci sont numérotées de 0 à N-1

On affecte de façon bijective un processus à un sous-domaine et à un cœur de calcul

Les tableaux sont alloués localement à la taille des sous-domaines et résident dans une mémoire locale

Les processus sont identifiés par les numéros et coordonnées des sous-domaines

Les processus exécutent tous le même programme en parallèle et s'échangent des données aux interfaces des sous-domaines

CALCUL PARALLÈLE

Temps de calcul versus temps de communication pour MPI

Quand on découpe un domaine global en N sous-domaines de calcul, pour des performances de parallélisation optimales, choisir N cœurs de calcul et N processus de calcul (1 processus par cœur).

S'assurer que par sous-domaine, il y ait suffisamment de degrés de libertés pour que les temps de communications ne soient pas supérieurs aux temps réels de calcul (solveur ou autre).

CALCUL PARALLÈLE

3.2.

Environnement MPI

Description

- Toute unité de programme appelant des sous-programmes MPI doit inclure un fichier d'en-têtes. En Fortran, il faut utiliser le *module* `mpi` introduit dans MPI-2 (dans MPI-1, il s'agissait du fichier `mpif.h`).
- Le sous-programme `MPI_INIT()` permet d'initialiser l'environnement nécessaire :

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

- Réciproquement, le sous-programme `MPI_FINALIZE()` désactive cet environnement :

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

Différence entre le c et Fortran

Concernant le c/c++ :

- Il faut inclure le fichier `mpi.h`;
- L'argument code est la valeur de retour de l'appel;
- Uniquement le préfix MPI ainsi que la première lettre suivante sont en majuscules ;
- Hormis `MPI_INIT()`, les arguments des appels sont identiques au Fortran.

```
int MPI_Init(int *argc, char ***argv);
int MPI_Finalize(void);
```

Communicateurs

- Toutes les opérations effectuées par MPI portent sur des **communicateurs**. Le communicateur par défaut est `MPI_COMM_WORLD` qui comprend tous les processus actifs.

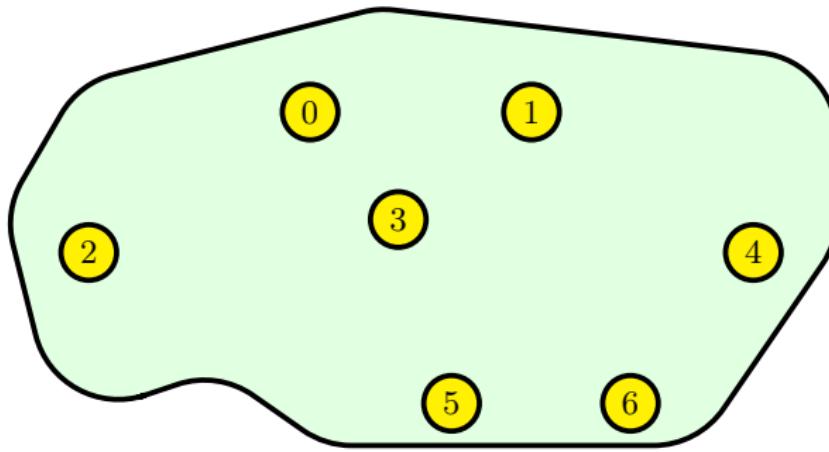


FIGURE 9 – Communicateur `MPI_COMM_WORLD`

Arrêt d'un programme

Parfois un programme se trouve dans une situation où il doit s'arrêter sans attendre la fin normale. C'est typiquement le cas si un des processus ne peut pas allouer la mémoire nécessaire à son calcul. Dans ce cas il faut utiliser le sous-programme **MPI_ABORT()** et non l'instruction Fortran *stop*.

```
integer, intent(in) :: comm, erreur
integer, intent(out) :: code
call MPI_ABORT(comm, erreur, code)
```

- **comm** : tous les processus appartenant à ce communicateur seront stoppés, il est donc conseillé d'utiliser **MPI_COMM_WORLD** ;
- **erreur** : numéro d'erreur retourné à l'environnement UNIX.

Code

Il n'est pas nécessaire de tester la valeur de **code** après des appels aux routines MPI. Par défaut, lorsque MPI rencontre un problème, le programme s'arrête comme lors d'un appel à **MPI_ABORT()**.

Rang et nombre de processus

- À tout instant, on peut connaître le nombre de processus gérés par un communicateur par le sous-programme **MPI_COMM_SIZE()** :

```
integer, intent(in) :: comm
integer, intent(out) :: nb_procs,code

call MPI_COMM_SIZE(comm,nb_procs,code)
```

- De même, le sous-programme **MPI_COMM_RANK()** permet d'obtenir le rang d'un processus (i.e. son numéro d'instance, qui est un nombre compris entre 0 et la valeur renvoyée par **MPI_COMM_SIZE() – 1**) :

```
integer, intent(in) :: comm
integer, intent(out) :: rang,code

call MPI_COMM_RANK(comm,rang,code)
```

```
1 program qui_je_suis
2   use mpi
3   implicit none
4   integer :: nb_procs,rang,code
5
6   call MPI_INIT(code)
7
8   call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
9   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
10
11  print *,'Je suis le processus ',rang,' parmi ',nb_procs
12
13  call MPI_FINALIZE(code)
14 end program qui_je_suis
```

```
> mpiexec -n 7 qui_je_suis
```

```
Je suis le processus 3 parmi 7
Je suis le processus 0 parmi 7
Je suis le processus 4 parmi 7
Je suis le processus 1 parmi 7
Je suis le processus 5 parmi 7
Je suis le processus 2 parmi 7
Je suis le processus 6 parmi 7
```

CALCUL PARALLÈLE

3.3.

Communications MPI
point à point

3 – Communications point à point

3.1 – Notions générales

Notions générales

Une communication dite **point à point** a lieu entre deux processus, l'un appelé processus **émetteur** et l'autre processus **récepteur** (ou **destinataire**).

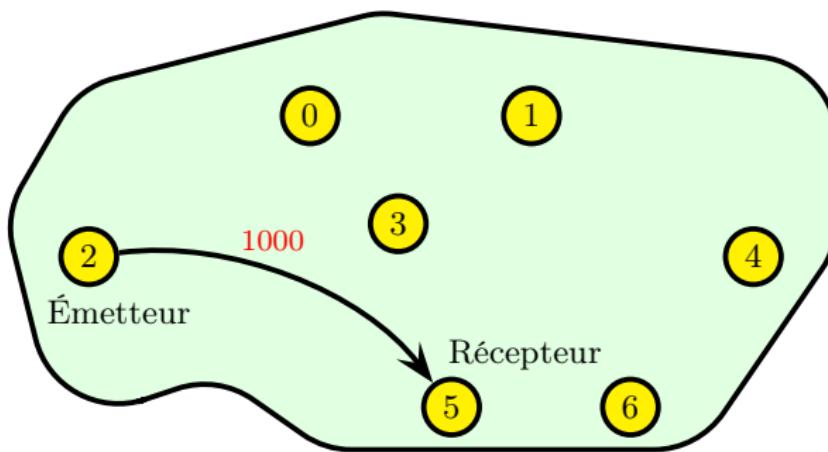


FIGURE 10 – Communication point à point

Notions générales

- L'émetteur et le récepteur sont identifiés par leur **rang** dans le communicateur.
- Ce que l'on appelle l'**enveloppe d'un message** est constituée :
 - du rang du processus émetteur ;
 - du rang du processus récepteur ;
 - de l'étiquette (*tag*) du message ;
 - du communicateur qui définit le groupe de processus et le contexte de communication.
- Les données échangées sont **typées** (entiers, réels, etc ou types dérivés personnels).
- Il existe dans chaque cas plusieurs **modes** de transfert, faisant appel à des protocoles différents.

3 – Communications point à point

3.2 – Opérations d'envoi et de réception bloquantes

Opération d'envoi **MPI_SEND**

```
<type et attribut>:: message
integer :: longueur, type
integer :: rang_dest, etiquette, comm, code
call MPI_SEND(message,longueur,type,rang_dest,etiquette,comm,code)
```

Envoi, à partir de l'adresse **message**, d'un message de taille **longueur**, de type **type**, étiqueté **etiquette**, au processus **rang_dest** dans le communicateur **comm**.

Remarque :

Cette opération est bloquante : l'exécution reste bloquée jusqu'à ce que le contenu de **message** puisse être réécrit sans risque d'écraser la valeur qui devait être envoyée.

Opération de réception MPI_RECV

```
<type et attribut>:: message
integer :: longueur, type
integer :: rang_source, etiquette, comm, code
integer, dimension(MPI_STATUS_SIZE) :: statut
call MPI_RECV(message, longueur, type, rang_source, etiquette, comm, statut, code)
```

Réception, à partir de l'adresse **message**, d'un message de taille **longueur**, de type **type**, étiqueté **etiquette**, du processus **rang_source**.

Remarques :

- **statut** reçoit des informations sur la communication : **rang_source**, **etiquette**, **code**,
- L'appel **MPI_RECV** ne pourra fonctionner avec une opération **MPI_SEND** que si ces deux appels ont la même enveloppe (**rang_source**, **rang_dest**, **etiquette**, **comm**).
- Cette opération est bloquante : l'exécution reste bloquée jusqu'à ce que le contenu de **message** corresponde au message reçu.

```
1 program point_a_point
2   use mpi
3   implicit none
4
5   integer, dimension(MPI_STATUS_SIZE) :: statut
6   integer, parameter :: etiquette=100
7   integer :: rang,valeur,code
8
9   call MPI_INIT(code)
10
11  call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
12
13  if (rang == 2) then
14    valeur=1000
15    call MPI_SEND(valeur,1,MPI_INTEGER,5,etiquette,MPI_COMM_WORLD,code)
16  elseif (rang == 5) then
17    call MPI_RECV(valeur,1,MPI_INTEGER,2,etiquette,MPI_COMM_WORLD,statut,code)
18    print *, 'Moi, processus 5, j''ai reçu ',valeur,' du processus 2.'
19  end if
20
21  call MPI_FINALIZE(code)
22
23 end program point_a_point
```

```
> mpiexec -n 7 point_a_point
```

Moi, processus 5, j'ai reçu 1000 du processus 2

3 – Communications point à point

3.3 – Types de données de base

Types de données de base Fortran

TABLE 1 – Principaux types de données de base (Fortran)

Type MPI	Type Fortran
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER

Types de données de base C

TABLE 2 – Principaux types de données de base (C)

Type MPI	Type C
MPI_CHAR	<code>signed char</code>
MPI_SHORT	<code>signed short</code>
MPI_INT	<code>signed int</code>
MPI_LONG	<code>signed long int</code>
MPI_UNSIGNED_CHAR	<code>unsigned char</code>
MPI_UNSIGNED_SHORT	<code>unsigned short</code>
MPI_UNSIGNED	<code>unsigned int</code>
MPI_UNSIGNED_LONG	<code>unsigned long int</code>
MPI_FLOAT	<code>float</code>
MPI_DOUBLE	<code>double</code>
MPI_LONG_DOUBLE	<code>long double</code>

CALCUL PARALLÈLE

3.4.

Phénomène de
« deadlock » avec MPI

CALCUL PARALLÈLE

Deadlock ou inter-blocage

2 tâches MPI veulent échanger des messages, mais toutes les 2 veulent envoyer leur message respectif et ne sont pas prêtes à recevoir.

L'exécution d'un programme faisant appel à la fonction **MPI_Send** suivie de **MPI_Recv** peut rester bloquée sur **MPI_Send** quant deux processus s'échangent des messages de tailles importantes.

Ce problème est dû à l'utilisation du mode d'envoi MPI dit standard **MPI_Send**. Ce mode autorise l'implémentation MPI à choisir la façon d'envoyer les messages.

En général, les petits messages sont recopier dans un espace mémoire temporaire avant d'être expédiés (**bufferisés**).

Les gros messages sont envoyés en mode **synchrone**. Ce mode synchrone implique que pour que le message parte (càd pour que l'appel **MPI_Send** puisse se terminer), il faut que la réception de ce message ait été postée (càd que l'appel à **MPI_Recv** ait été effectué).

CALCUL PARALLÈLE

Deadlock ou inter-blocage

1er scénario possible :

Process 0	Process 1
Send(1)	Send(0)
Recv(1)	Recv(0)

Ca peut marcher pour les envois de petite taille qui seront « bufferisés » mais pas pour les envois de grandes taille qui fonctionnent en mode synchrone.

Ce schéma de communication n'est pas du tout conseillé et est d'ailleurs considéré comme erroné par la norme MPI.

La valeur par défaut de la taille à partir de laquelle les envois ne sont pas « bufferisés » (donc synchrones) varie d'une architecture de machine à une autre. Cette valeur peut être changée selon les architectures.

Si vous voulez vous assurer que votre application ne risque pas de souffrir de ce problème, il est conseillé de la tester en mettant cette valeur à 0 afin de forcer le mode synchrone pour tous les envois standards. Si tout se passe bien (pas de blocage), votre application ne devrait pas avoir ce souci.

CALCUL PARALLÈLE

Deadlock ou inter-blocage

2ème scénario possible :

Process 0	Process 1
Recv(1)	Recv(0)
Send(1)	Send(0)

Ca bloque définitivement quelque soit la taille des messages envoyés.

Les deux processus sont bloqués au Recv et les Send ne s'exécutent pas.

CALCUL PARALLÈLE

Deadlock ou inter-blocage

Le 3ème scénario qui évite le phénomène de deadlock :

Process 0	Process 1
Send(1)	Recv(0)
Recv(1)	Send(0)

Attention aux étiquettes : si un processus de rang R1 envoie un message M au processus R2 avec l'étiquette E1, le processus R2 reçoit le message M avec la même étiquette E1. Même avec ce scénario, si les étiquettes ne sont pas bien mises, ça peut bloquer quand même !

CALCUL PARALLÈLE

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()

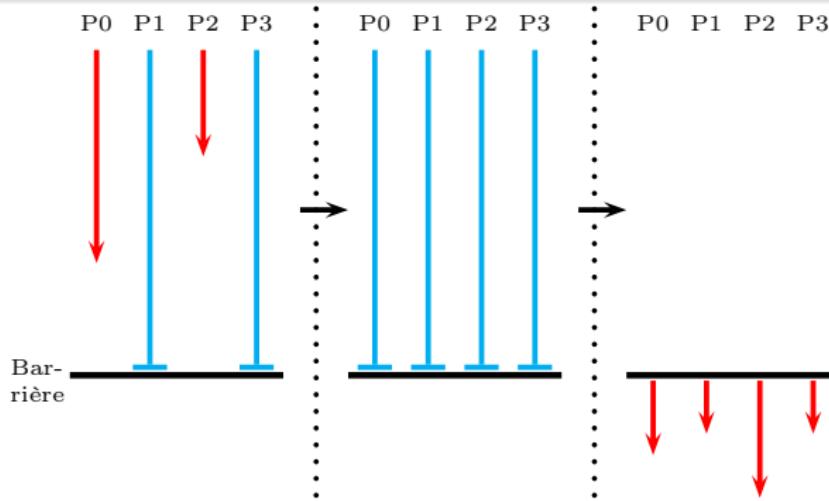


FIGURE 12 – Synchronisation globale : MPI_BARRIER()

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

4 – Communications collectives

4.3 – Diffusion générale : MPI_BCAST()

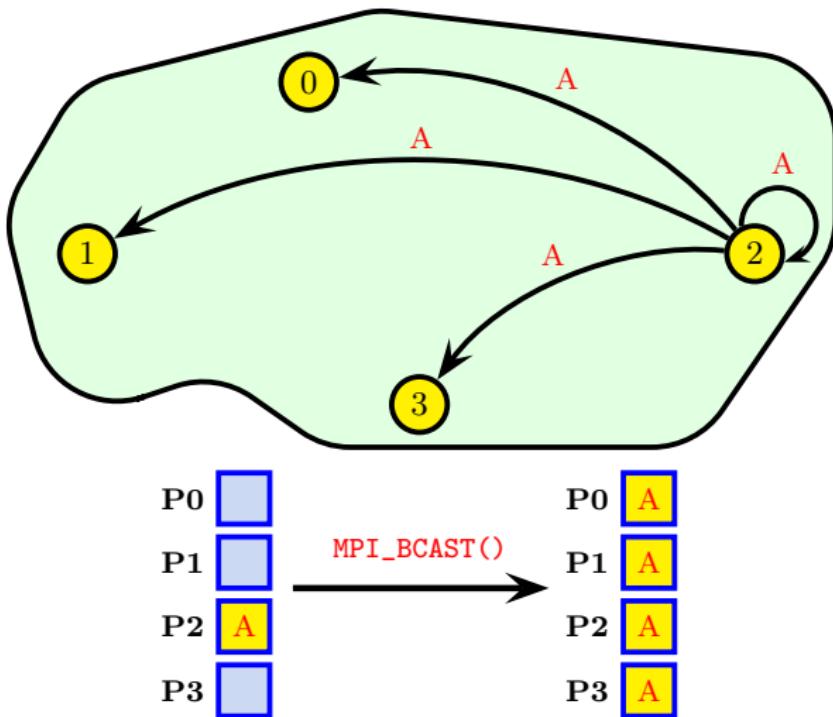


FIGURE 13 – Diffusion générale : MPI_BCAST()

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**.

```
1 program bcast
2 use mpi
3 implicit none
4
5 integer :: rang,valeur,code
6
7 call MPI_INIT(code)
8 call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
9
10 if (rang == 2) valeur=rang+1000
11
12 call MPI_BCAST(valeur,1,MPI_INTEGER,2,MPI_COMM_WORLD,code)
13
14 print *,'Moi, processus ',rang,' j''ai reçu ',valeur,' du processus 2'
15
16 call MPI_FINALIZE(code)
17
18 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()

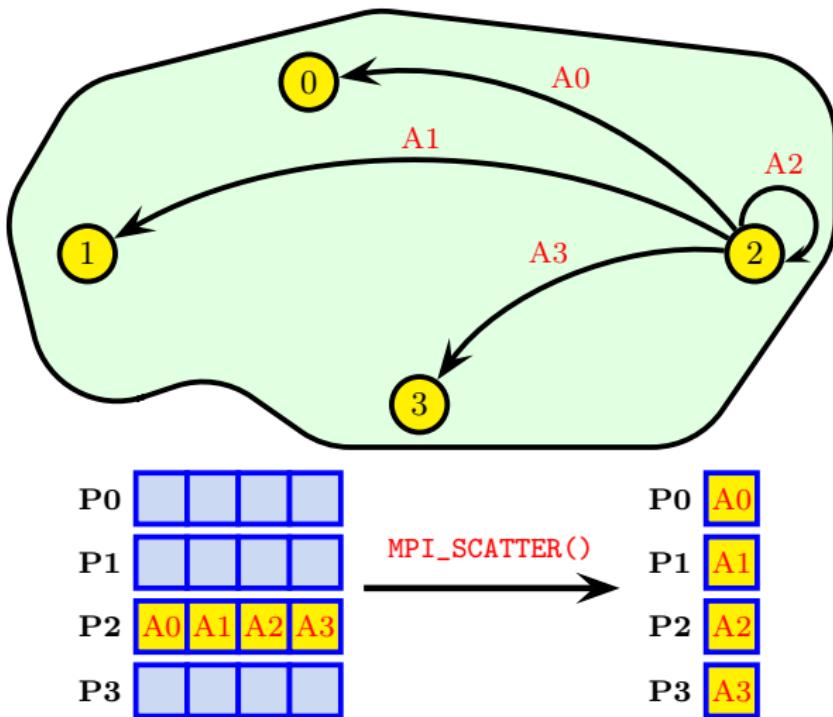


FIGURE 14 – Diffusion sélective : MPI_SCATTER()

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,
                 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()

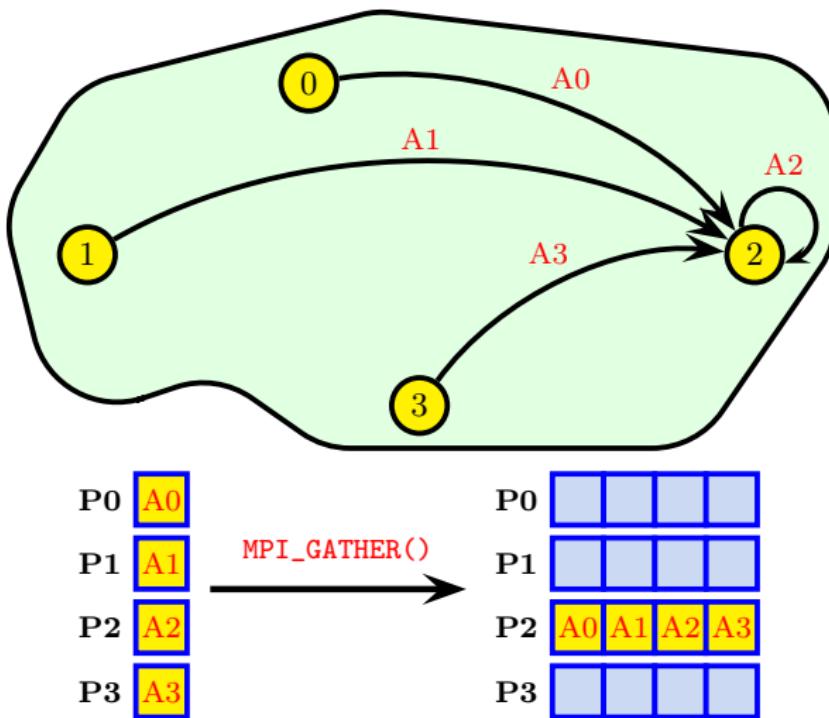


FIGURE 15 – Collecte : MPI_GATHER()

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.

```

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

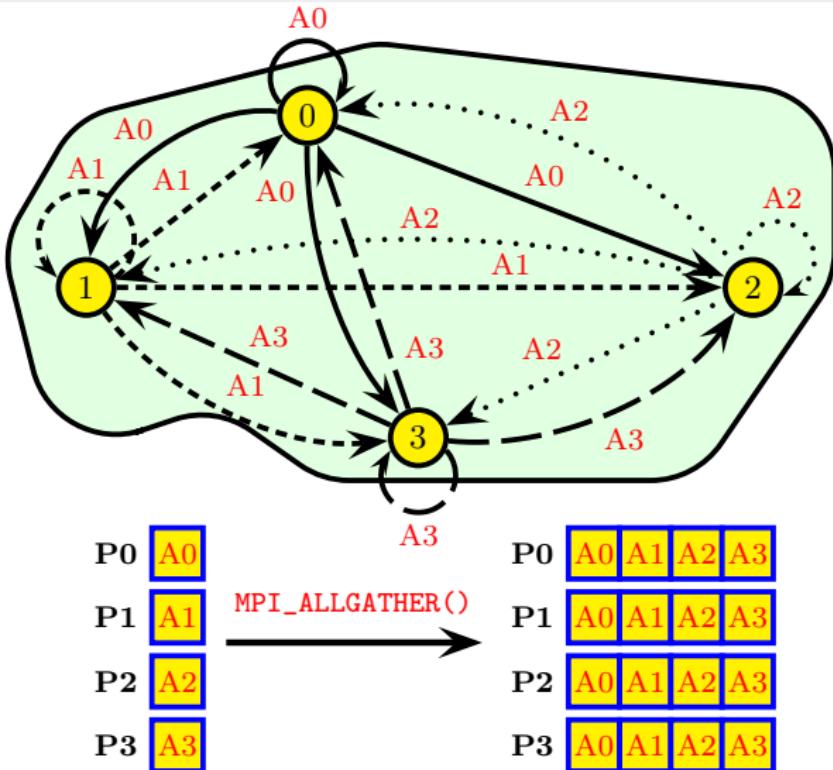


FIGURE 16 – Collecte générale : MPI_ALLGATHER()

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()

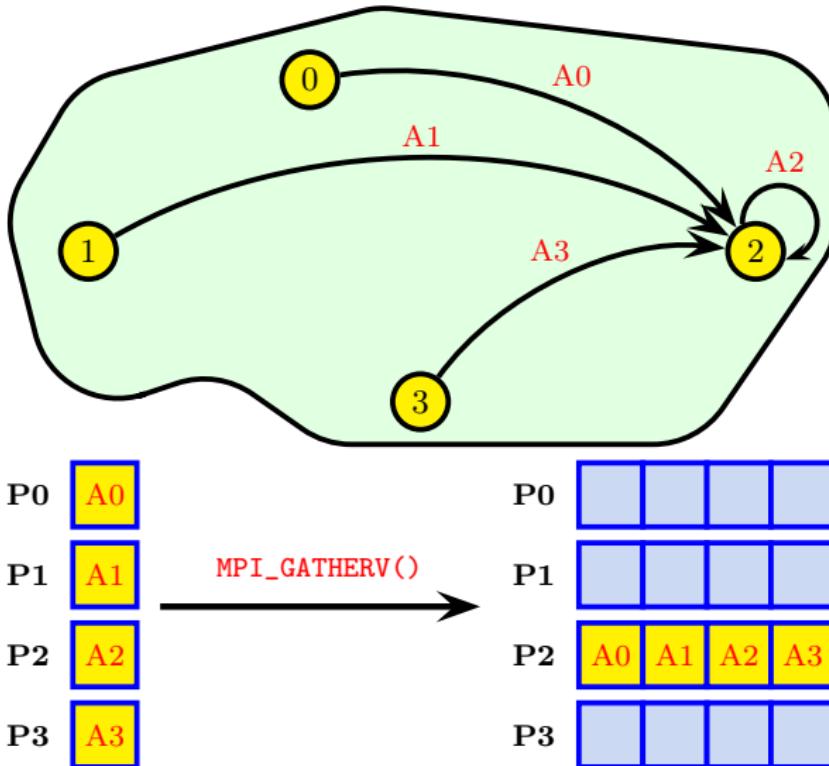


FIGURE 17 – Collecte : MPI_GATHERV()

MPI – Version 4.8.0 – Septembre 2016

I. Dupays, M. Flé, J. Gaidamour, D. Girou, P.-F. Lavallée, D. Lecas, P. Wautelet

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()

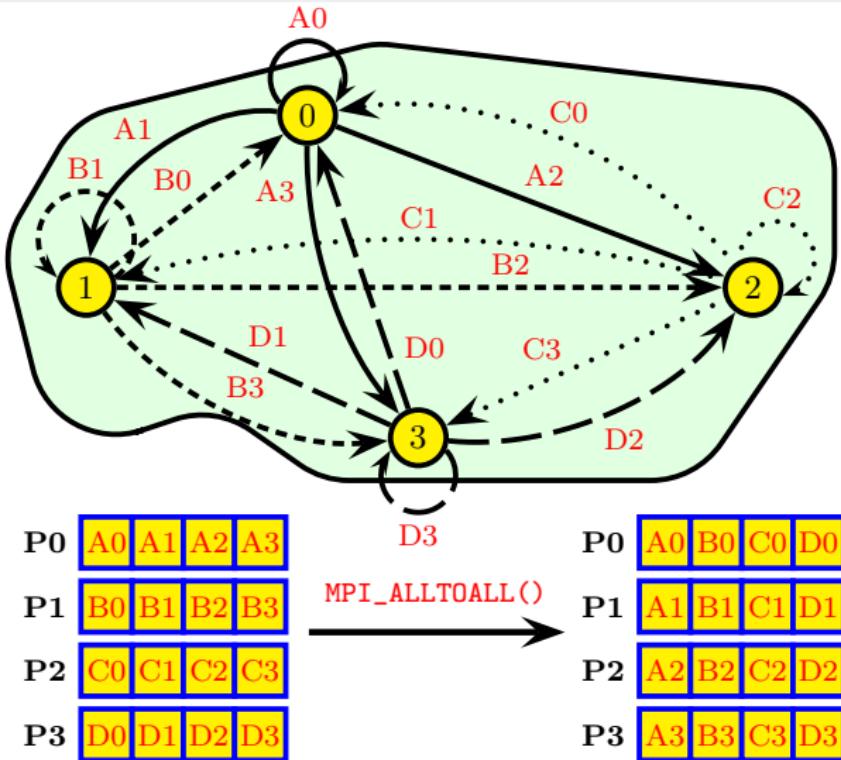


FIGURE 18 – Collecte et diffusion sélectives : MPI_ALLTOALL()

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

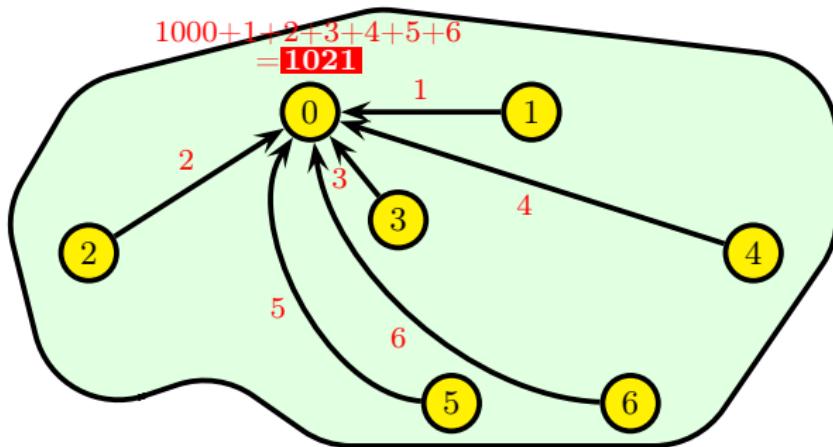


FIGURE 19 – Réduction répartie : MPI_REDUCE() avec l'opérateur somme

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

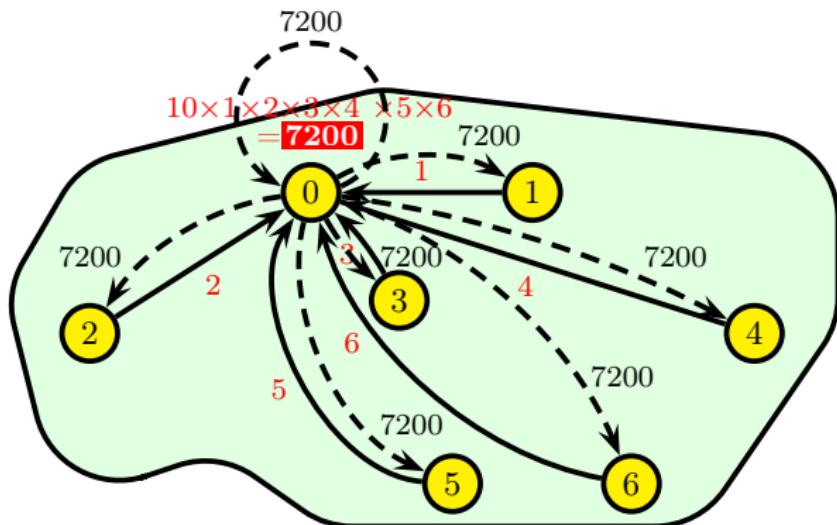


FIGURE 20 – Réduction répartie avec diffusion du résultat : MPI_ALLREDUCE (utilisation de l'opérateur produit)

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
```

CALCUL PARALLÈLE

3.6.

Compilation et
exécution d'un
programme MPI

CALCUL PARALLÈLE

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

CALCUL PARALLÈLE

4.

OpenMP versus MPI

CALCUL PARALLÈLE

MPI vs OpenMP

MPI utilise un schéma à mémoire distribuée

OpenMP utilise un schéma à mémoire partagée

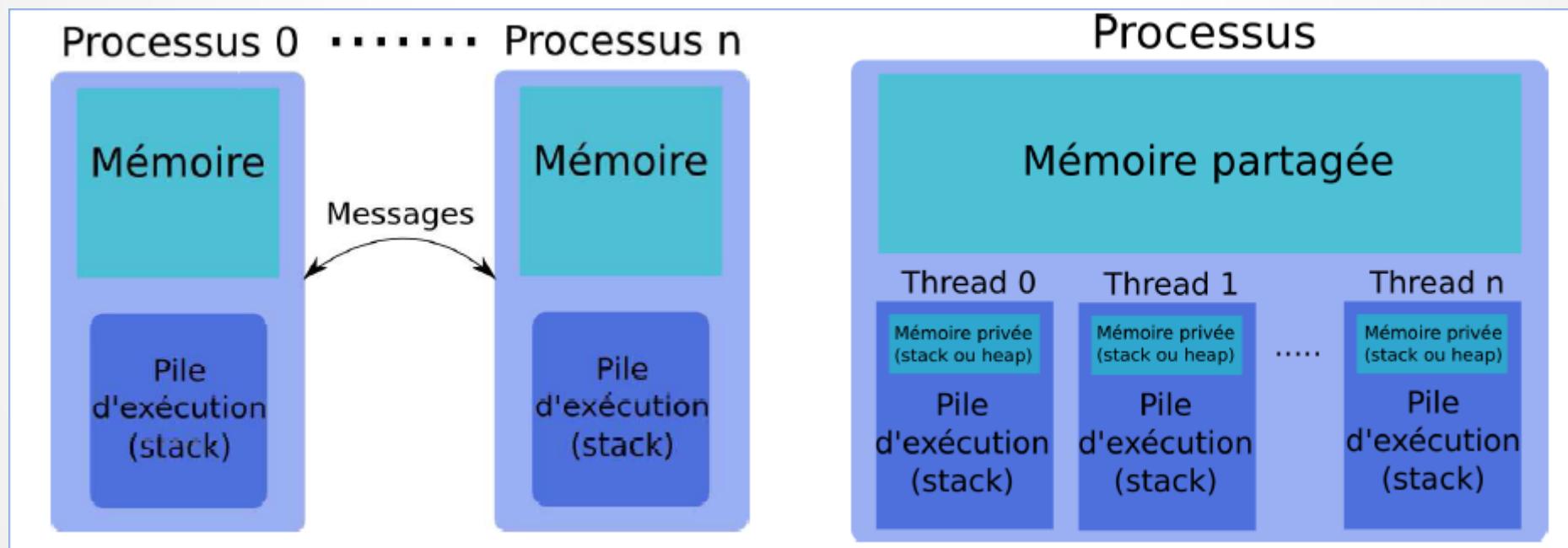


Schéma MPI

Schéma OpenMP

CALCUL PARALLÈLE

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 coût de développement
- selon les codes, ca peut être plus compliqué à développer que MPI
- limitations des performances sur un nombre de threads

CALCUL PARALLÈLE

5.

Programmation hybride
MPI/OpenMP

CALCUL PARALLÈLE

Principe de la programmation hybride MPI/OpenMP

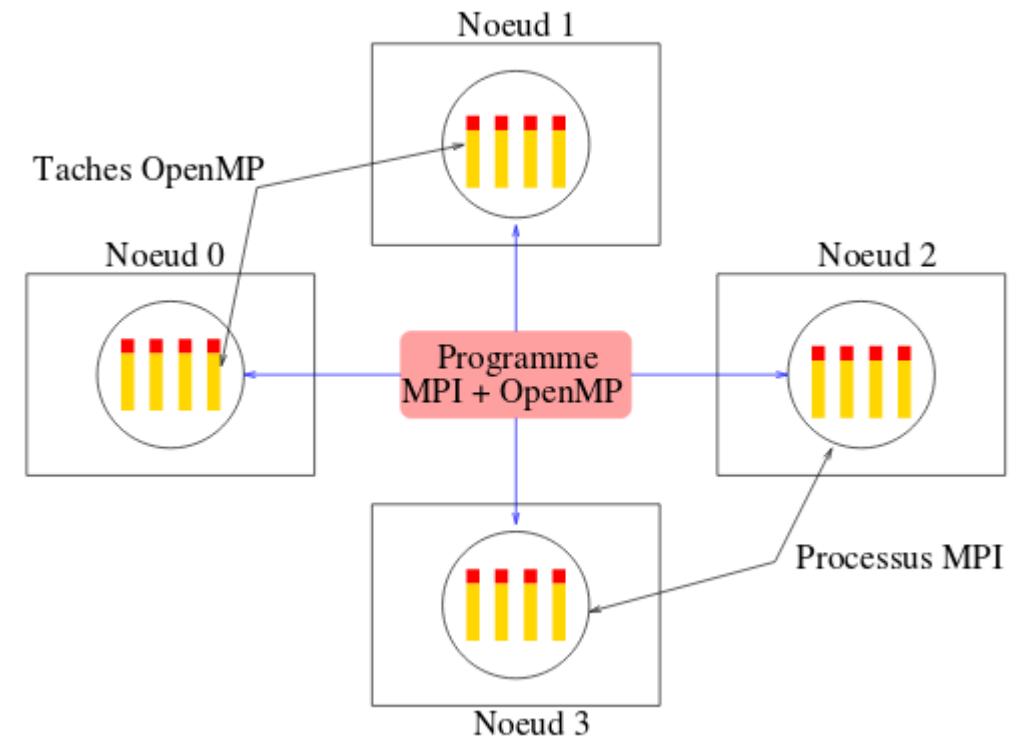
La programmation hybride parallèle consiste à mélanger plusieurs paradigmes de programmation parallèle dans le but de tirer parti des avantages des différentes approches.

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

CALCUL PARALLÈLE

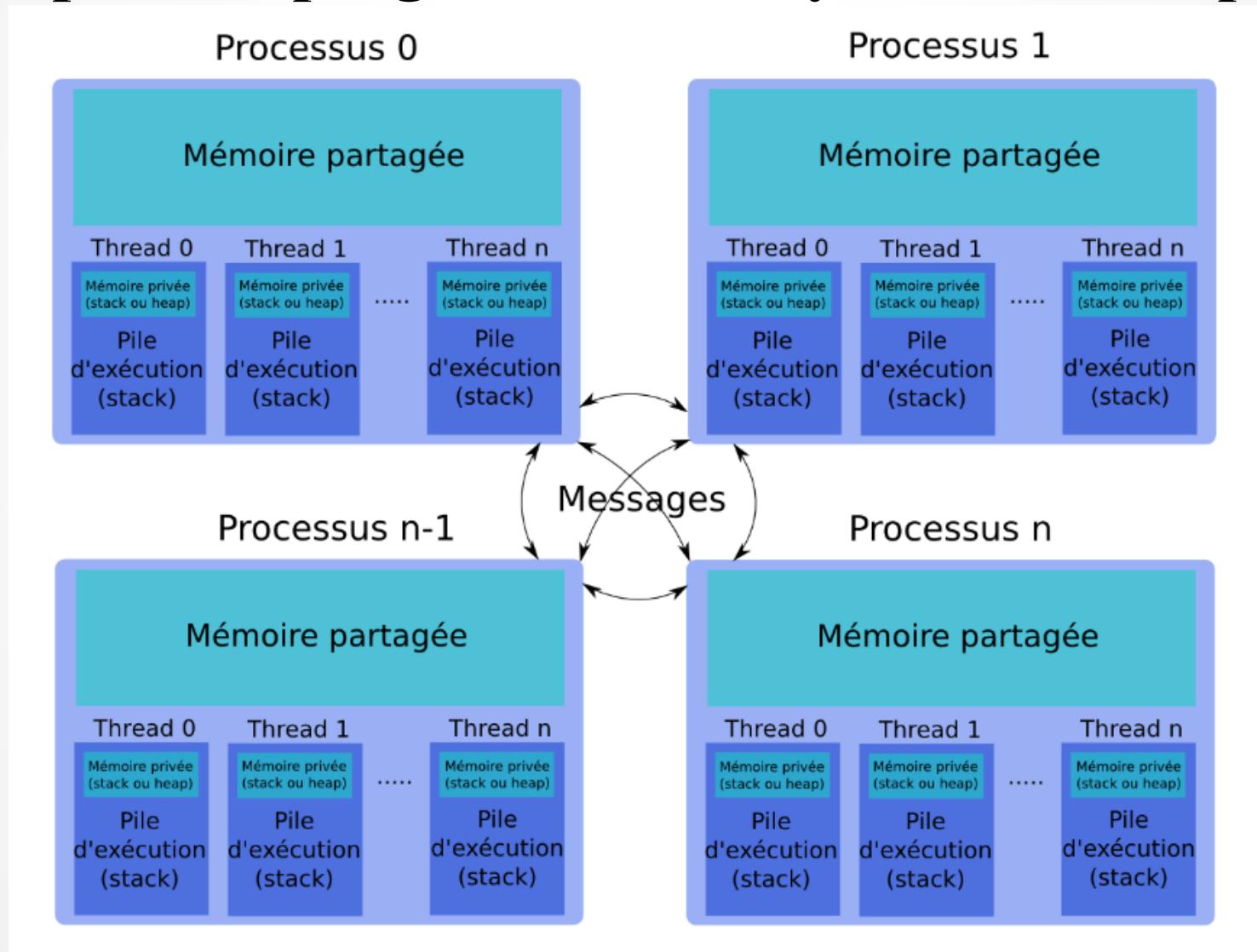
Principe de la programmation hybride MPI/OpenMP

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



CALCUL PARALLÈLE

6.

Mesures de performances

CALCUL PARALLÈLE

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, ...

CALCUL PARALLÈLE

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 processus (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

CALCUL PARALLÈLE

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 :

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

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

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

CALCUL PARALLÈLE

Accélération

Appréciation de l'accélération :

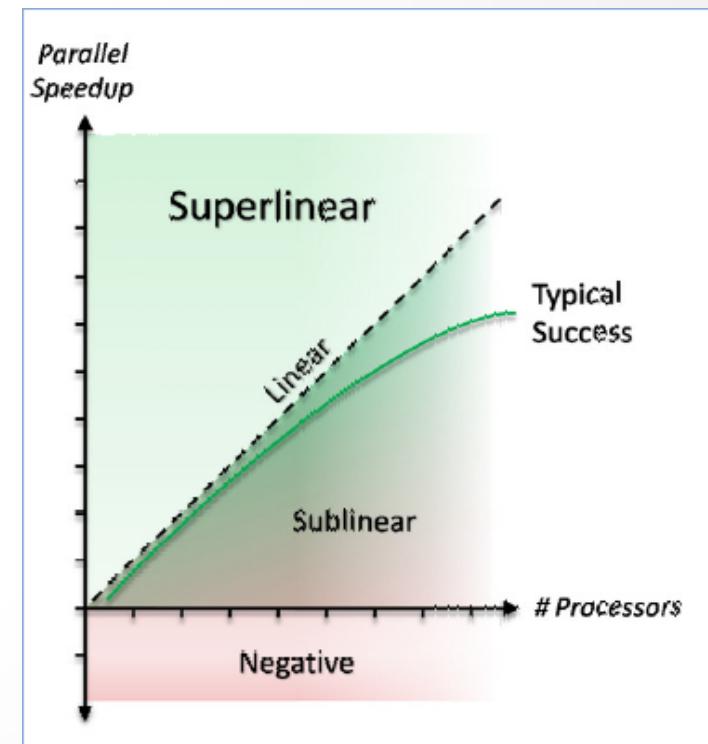
- accélération linéaire : parallélisation optimale
- accélération sur-linéaire : attention !
- accélération sub-linéaire : ralentissement dû au parallélisation

Pour une accélération parallèle parfaite on obtient :

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

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

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



CALCUL PARALLÈLE

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

$T(P)$: 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 $C \rightarrow \infty$, 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 $T(PS) \rightarrow 0$, 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 :

$$T(1) = (T(\text{partie parallèle}) + T(\text{partie non parallèle})) = 80$$

$$+ T(\text{partie non parallèle}) = 20$$

Sur C coeurs, on obtient pour une parallélisation optimale :

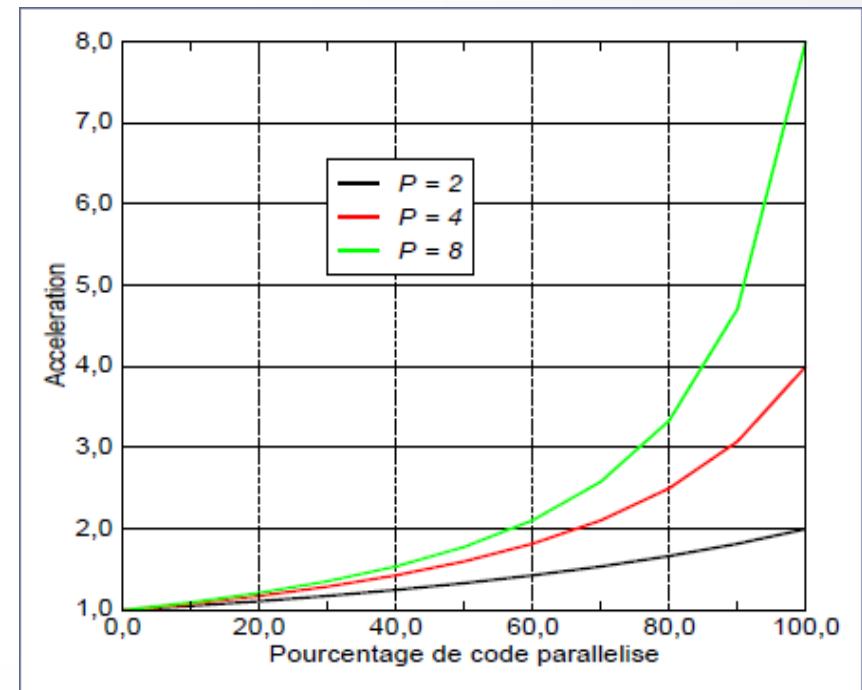
$$T(C) = (80 / C) + 20$$

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

$$\begin{aligned} A(C) &= T(1) / T(C) \\ &= 100 / ((80 / C) + 20) < 100/20 = 5 \end{aligned}$$

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

quand $C \rightarrow +\infty$



Dans le schéma, $p = C$

CALCUL PARALLÈLE

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)

CALCUL PARALLÈLE

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 cœur
- ◆ Si le temps de calcul est constant : scalabilité faible parfaite

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

CALCUL PARALLÈLE

Speedup et scalabilité en mesure relative

Les mesures de speedup et scalabilité vues précédemment sont **absolues** car rapportées par rapport à l'exécution séquentielle sur 1 cœur 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 cœur de calcul, par conséquent on définit 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 $C \geq ncmin$

CALCUL PARALLÈLE

MERCI POUR VOTRE ATTENTION

