CALCUL PARALLÈLE

MCIA

Khodor.Khadra@math.u-bordeaux.fr

Du 27 juin au 01 juillet 2022

Sessions antérieures : 2017, 2018, 2019, 2020

CALCUL PARALLÈLE

Certaines planches de ce support de cours reprennent le cours de l'IDRIS www.idris.fr

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

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 coeur

CALCUL PARALLÈLE

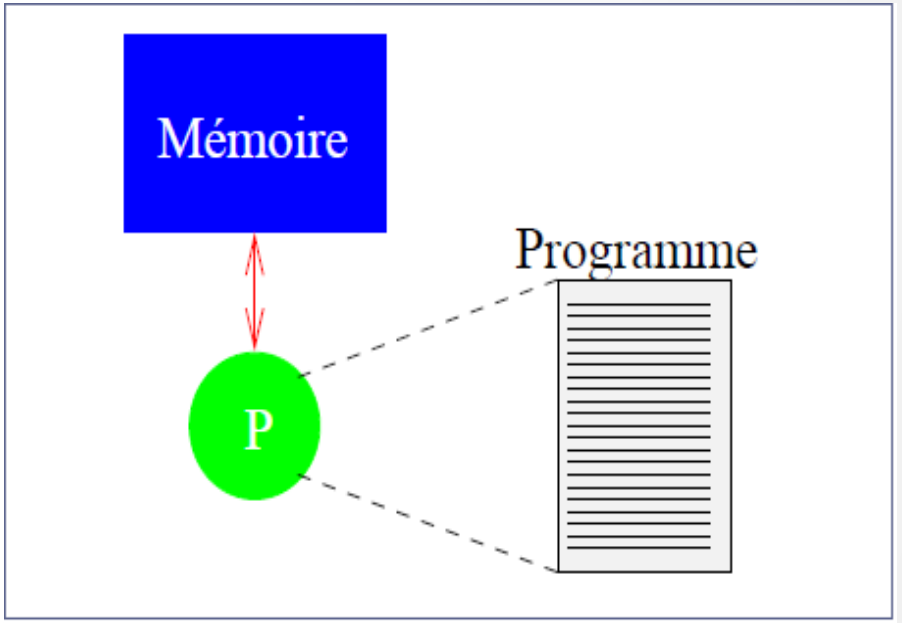
Petit rappel en mode séquentiel

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

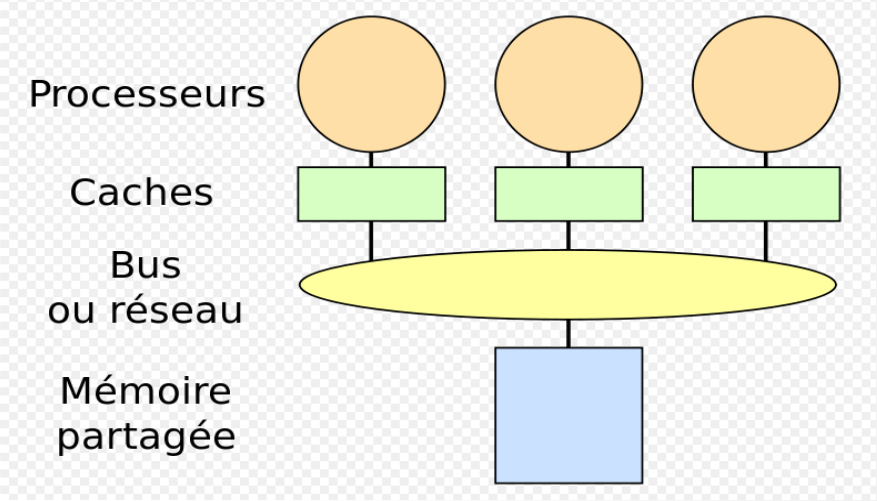
Un processus s'exécute sur un cœur d'un processeur physique de la machine



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.



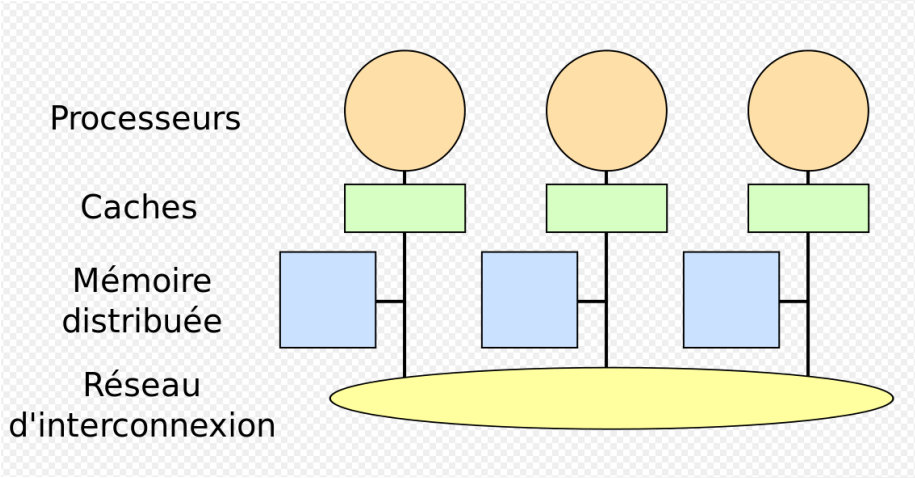
Symmetric shared memory multiprocessor (SMP) : architecture parallèle qui consiste à multiplier les processeurs identiques au sein d'un noeud, de manière à augmenter la puissance de calcul avec une mémoire unique partagée

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

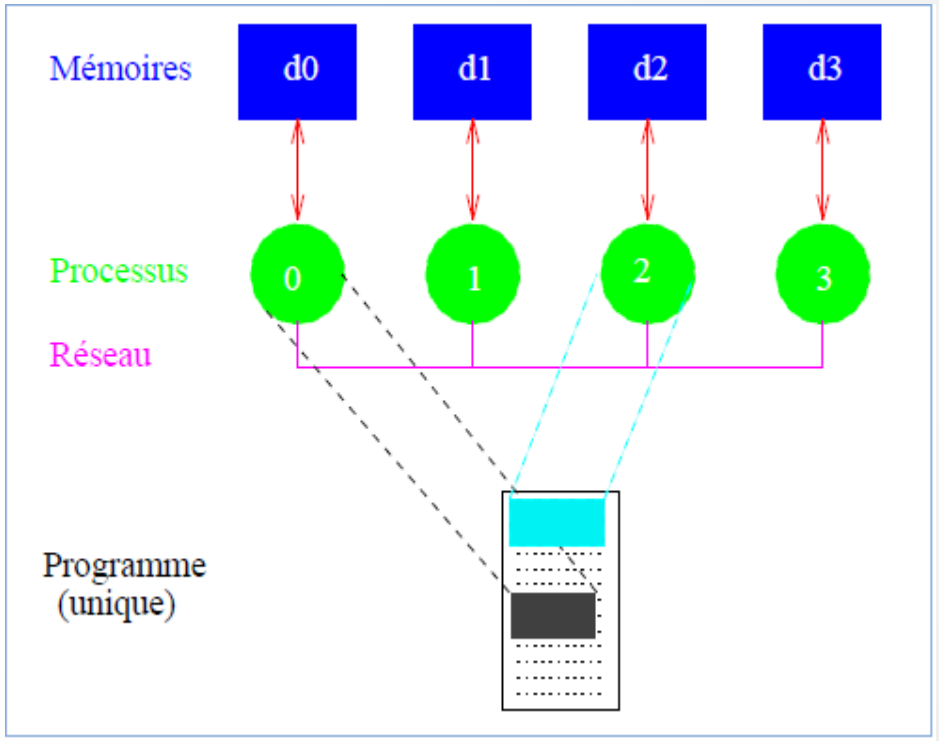
Programme écrit dans un langage classique (Fortran, C, C++, ...)

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

Bonne règle : un seul processus par cœur

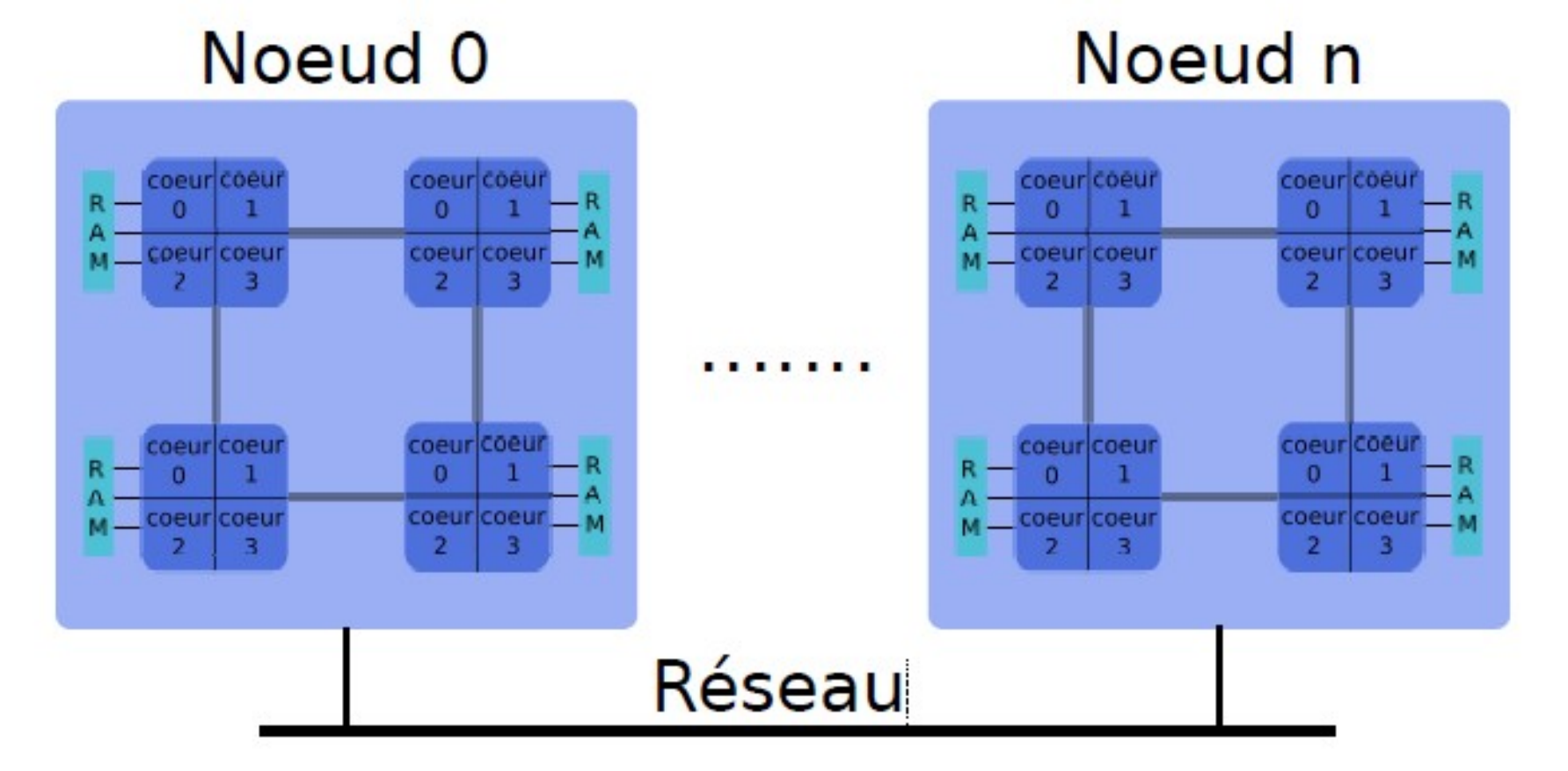
Toutes les variables du programme sont privées et résident dans la mémoire locale allouée à chaque processus

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



CALCUL PARALLÈLE

Cluster de calcul à mémoire distribuée



2.
Programmation OpenMP
Généralités

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

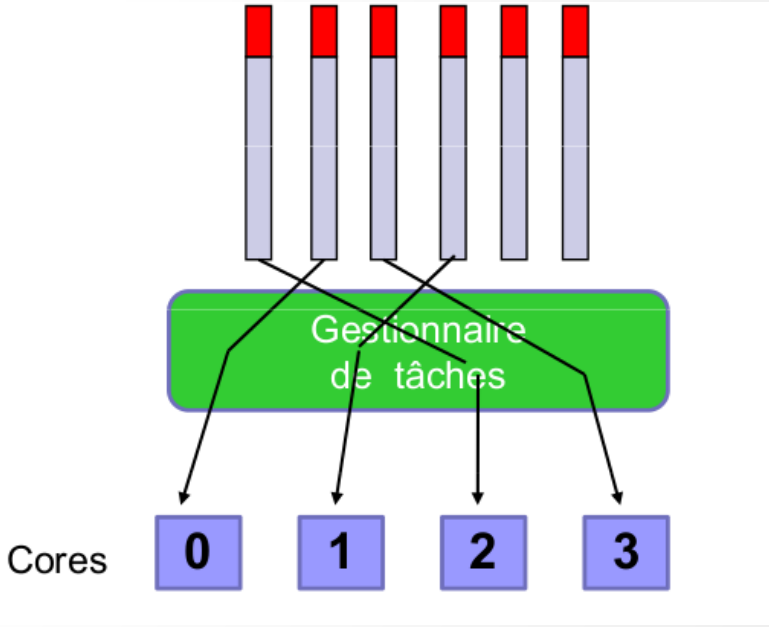
Les tâches peuvent être attribuées à des coeurs distincts

Exécution d'un programme OpenMP sur un nœud multicoeur

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

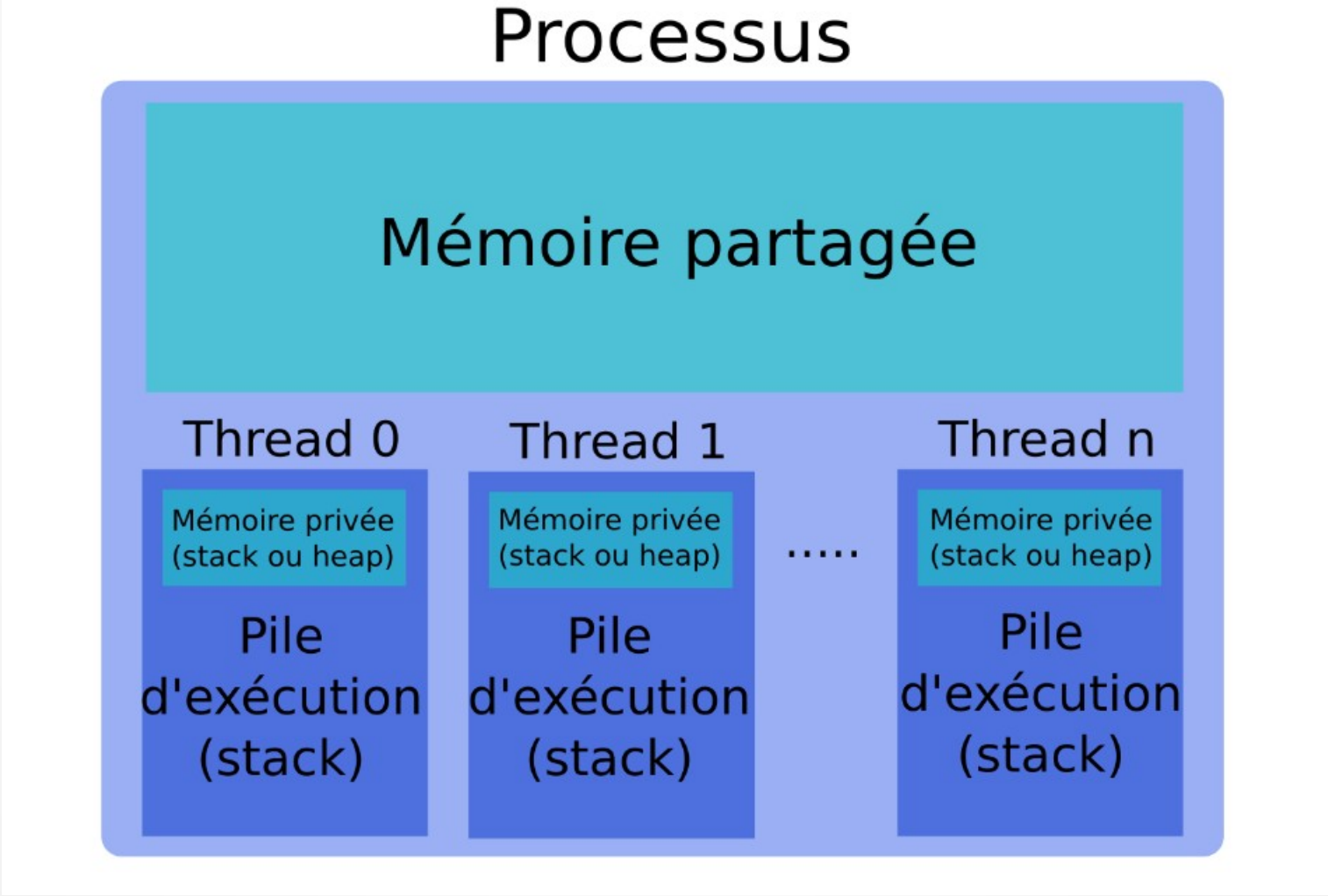
Les threads accèdent aux mêmes ressources que le processus.

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



CALCUL PARALLÈLE

Schéma de principe OpenMP



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

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

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

Compilation et exécution OpenMP

Option de compilation : -openmp

Définir le nombre de threads avant exécution

```
export OMP_NUM_THREADS=2
```

Lancer l'exécutable :

```
./monexec
```

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

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 phénomème s'accentue.

3.

Programmation MPI

3.1.
Généralités sur MPI

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

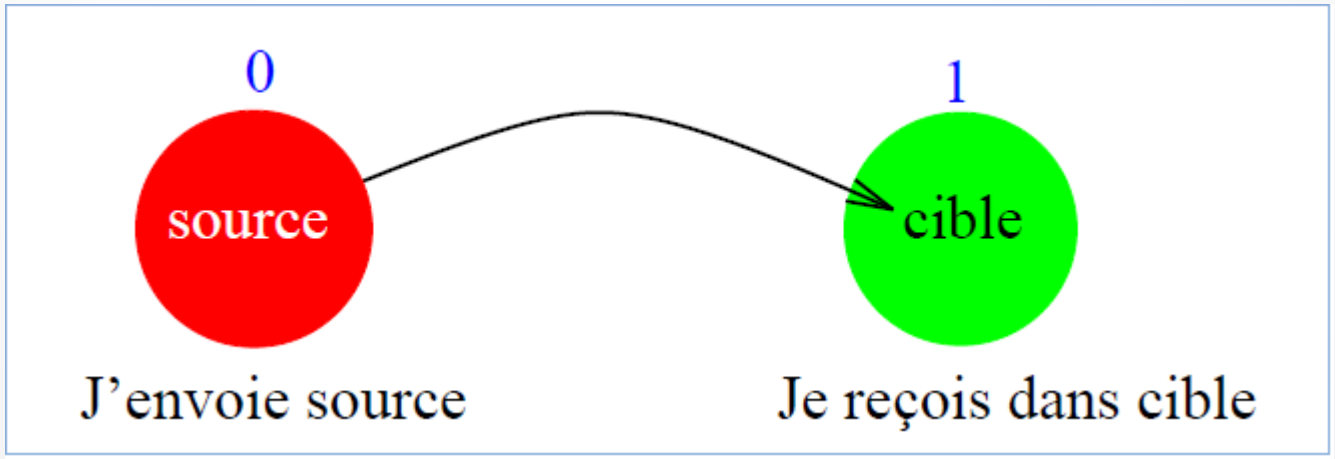
L'environnement en question est MPI (Message Passing Interface). Une application MPI est un ensemble de processus autonomes exécutant chacun leur propre code et communiquant via des appels à des sous-programmes de la bibliothèque MPI

Une parallélisation efficace doit minimiser les communications par rapport aux calculs

Echanges de messages

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

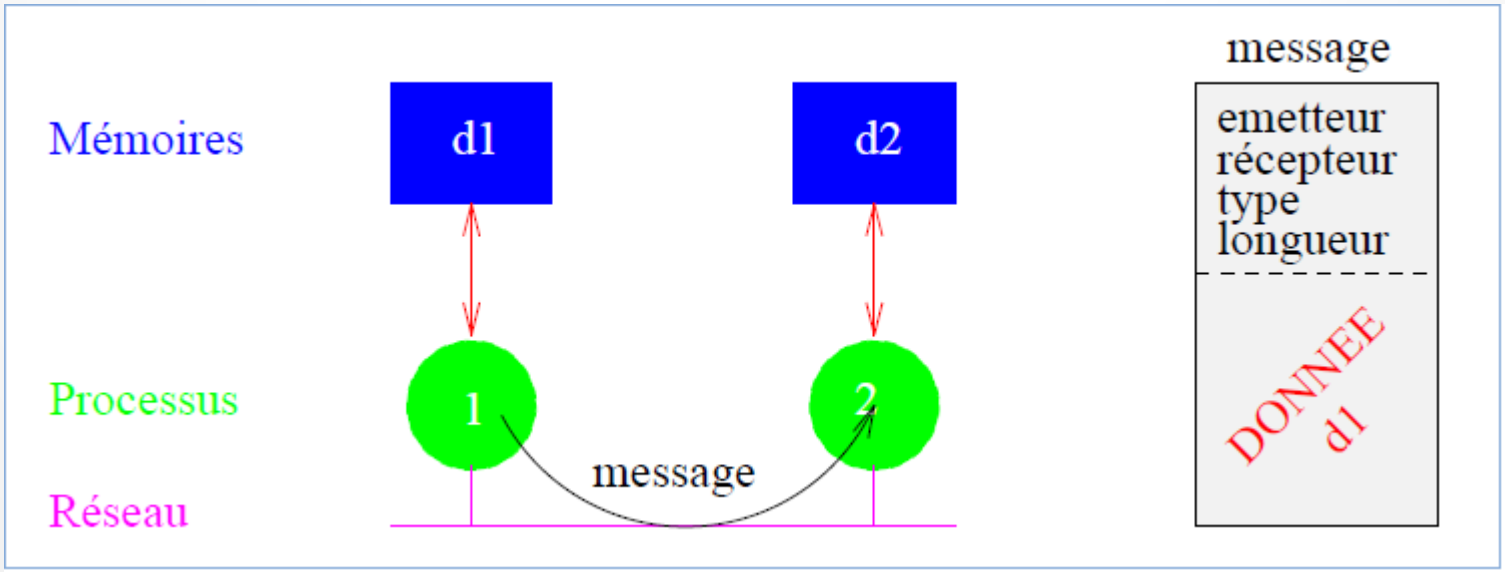
Un message est constitué de paquets de données transitant du processus émetteur au processus récepteur



Echanges de messages

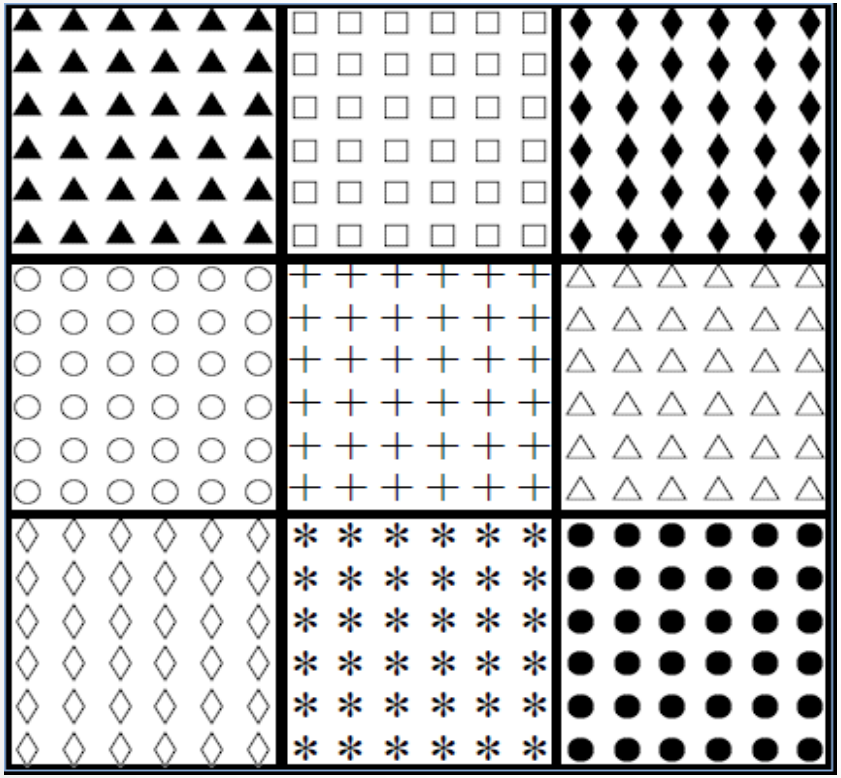
En plus des données (variables scalaires, tableaux, etc.) à transmettre, un message doit contenir les informations suivantes :

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



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).

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.

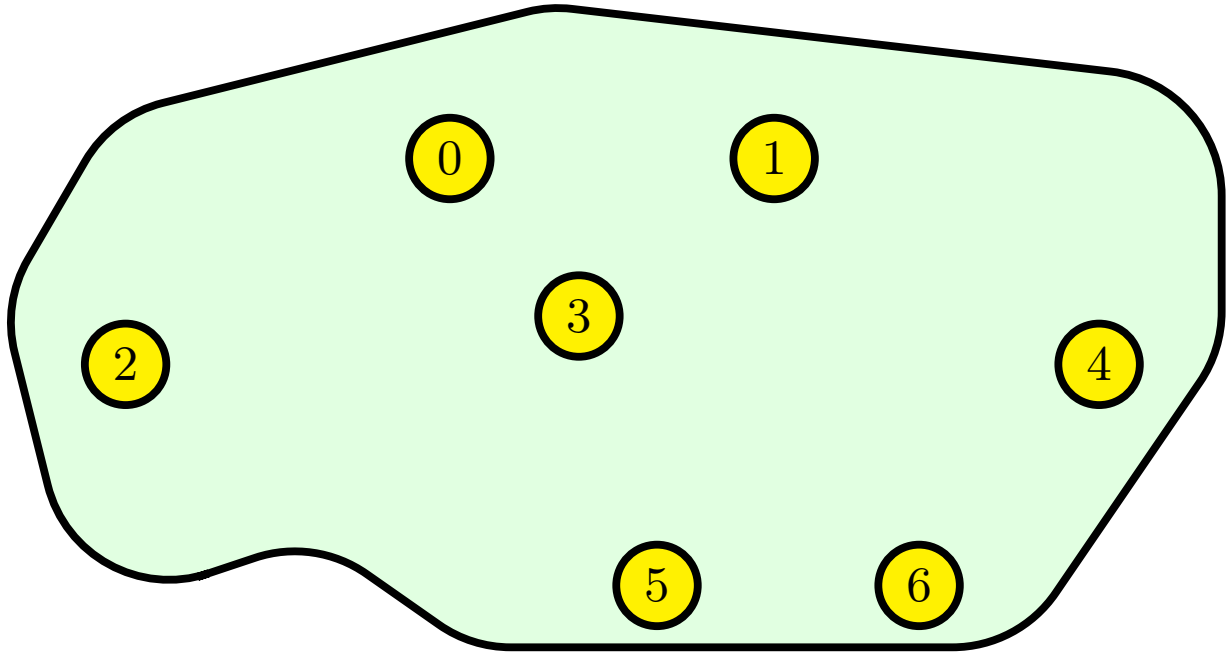


FIGURE 9 – Communicateur MPI_COMM_WORLD

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)
```

```fortran
program qui_je_suis
  use mpi
  implicit none
  integer :: nb_procs,rang,code

  call MPI_INIT(code)

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

  print *,'Je suis le processus ',rang,' parmi ',nb_procs

  call MPI_FINALIZE(code)
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
```