Modèles de programmation

Introduction au calcul parallèle
ANGD Choix, installation et exploitation d'un calculateur

Guy Moebs

Laboratoire de Mathématiques Jean Leray, Université de Nantes

Octobre 2009

Plan de la présentation

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 y a eu beaucoup de progrès technologique dans le matériel informatique (processeurs à 3 GHz, bus mémoire, ...) mais il ne faut pas compter sur des avancées au même rythme
- Pourtant il existe toujours des applications scientifiques qui consomment "trop" de ressources en temps de processeur ou en mémoire
- Pour celles-ci, la seule solution, pour des raisons techniques ou économiques, reste la parallélisation

But de la parallélisation

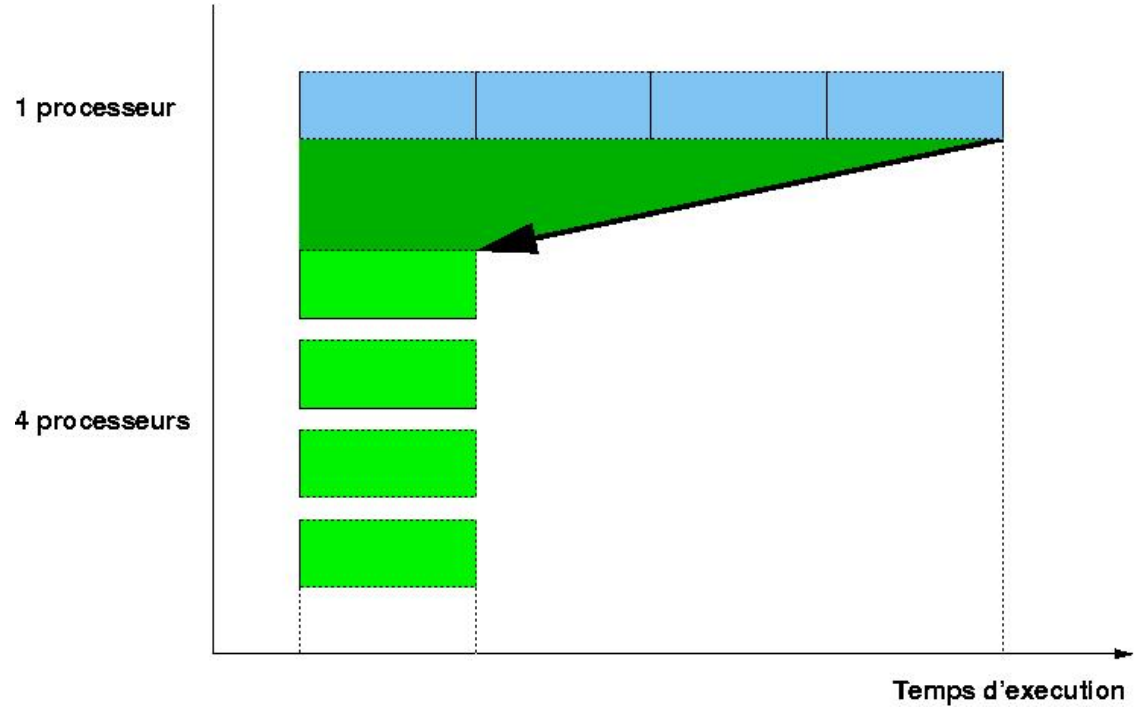
Qu'est-ce que la parallélisation ?

Ensemble de techniques logicielles et matérielles permettant l'exécution simultanée de séquences d'instructions indépendantes, sur des processeurs différents

Bénéfice de la parallélisation ?

- Exécution plus rapide du programme (gain en temps de restitution) en distribuant le travail
- Résolution de problèmes plus gros (plus de ressource matérielle accessible, notamment la mémoire)

But de la parallélisation



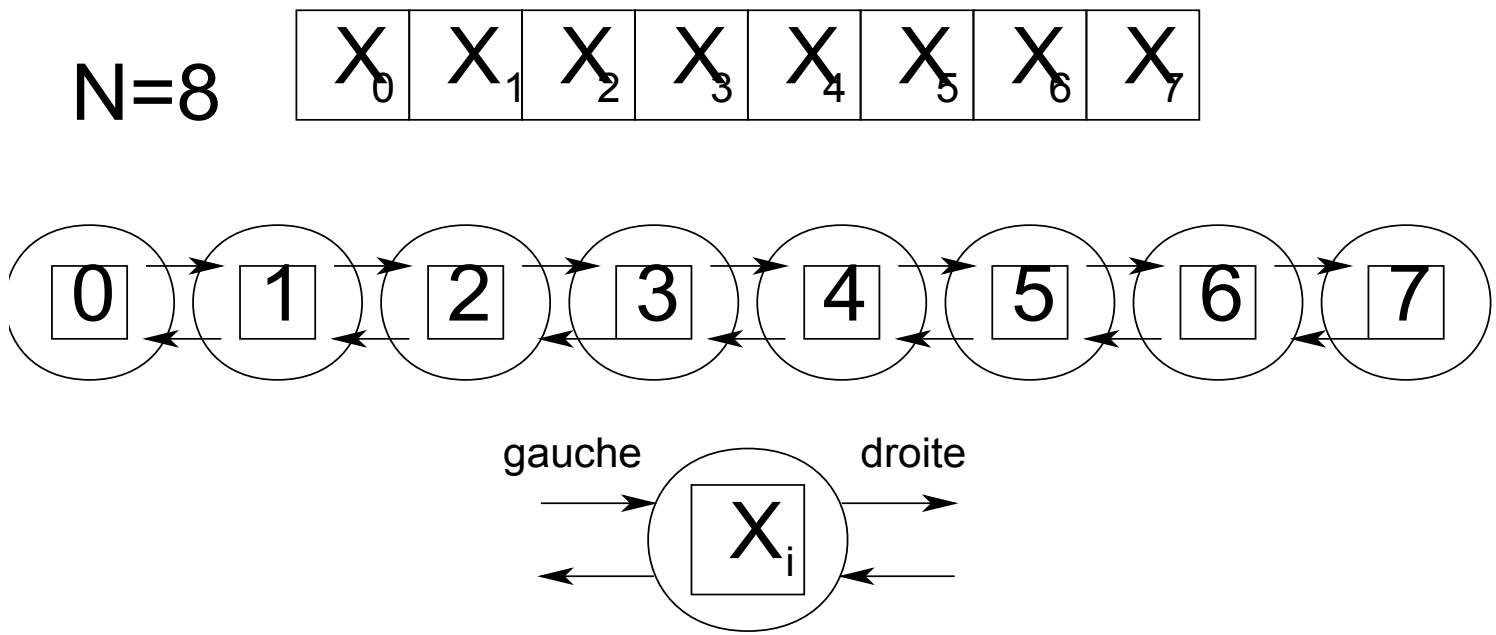
- Diminution du temps d'exécution – le but
- (Faible) augmentation du temps CPU total - effet de bord

Exemple : différences finies

- Problème de différences finies 1D

- On dispose du vecteur $X^{(0)}$ de taille N et on doit calculer $X^{(T)}$ selon :

$$0 < i < N-1, 0 \leq t < T : \quad X_i^{(t+1)} = \frac{X_{i-1}^{(t)} + 2X_i^{(t)} + X_{i+1}^{(t)}}{4}$$

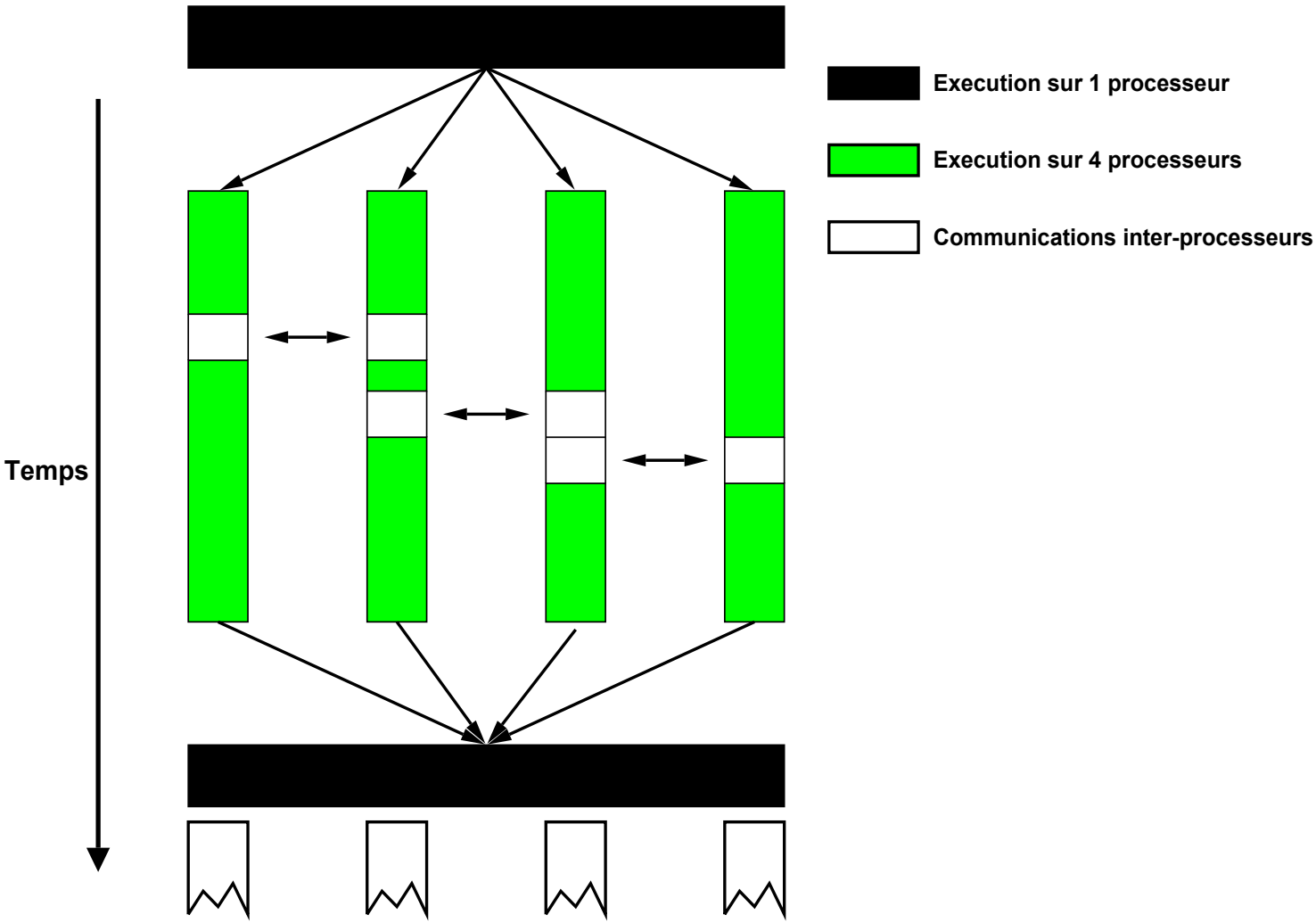


Exemple

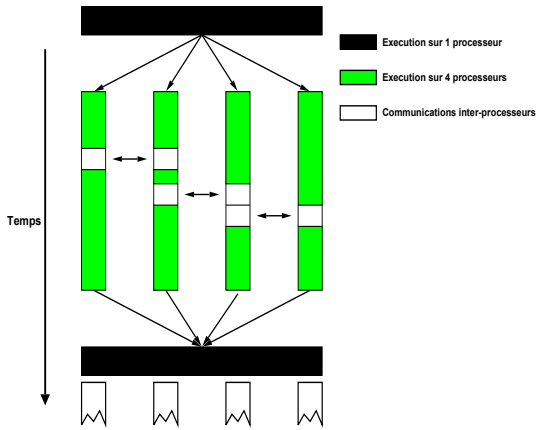
Algorithme parallèle :

- création de N tâches, une par valeur de X
- la tâche i dispose de $X_i^{(0)}$ et doit calculer $X_i^{(1)}$, $X_i^{(2)}$, ..., $X_i^{(T)}$
- A chaque étape elle doit disposer de $X_{i-1}^{(t)}$ et $X_{i+1}^{(t)}$ en provenance des tâches $i-1$ et $i+1$ (attention à $i=0$ et $i=N$)
- Travail en 3 étapes pour la tâche i :
  1. "envoie" sa donnée $X_i^{(t)}$ à ces voisines, gauche et droite (si elles existent)
  2. "reçoit" $X_{i-1}^{(t)}$ et $X_{i+1}^{(t)}$ en provenance des tâches $i-1$ et $i+1$ (attention à $i=0$ et $i=N$)
  3. utilise ces données pour calculer $X_i^{(t+1)}$
- Les N tâches travaillent indépendamment
- Seule contrainte : synchronisation lors des réceptions pour effectuer l'étape 3

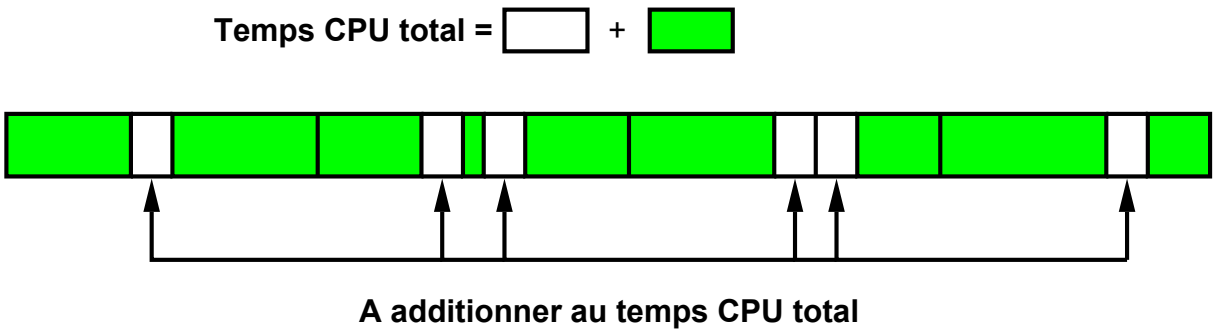
Optimisation de la parallélisation



Optimisation de la parallélisation



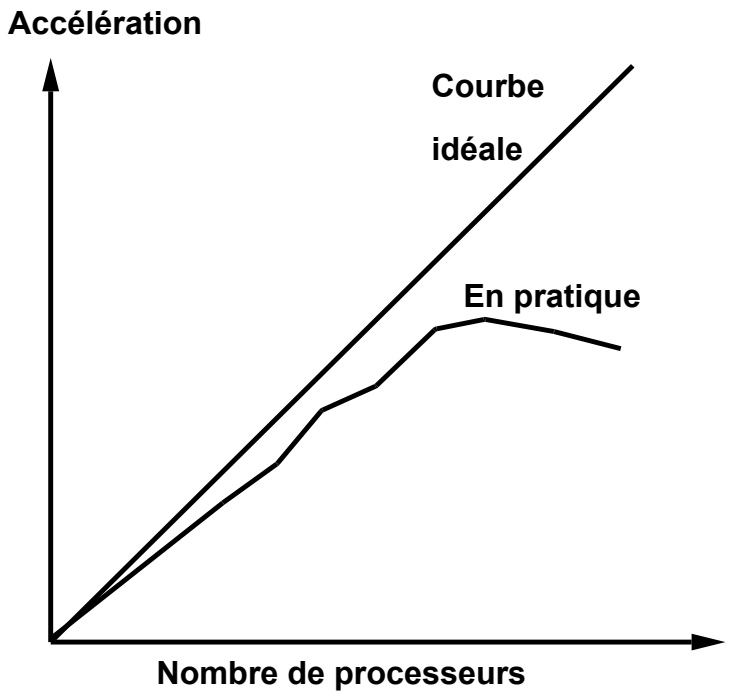
Une parallélisation efficace
minimise les communications

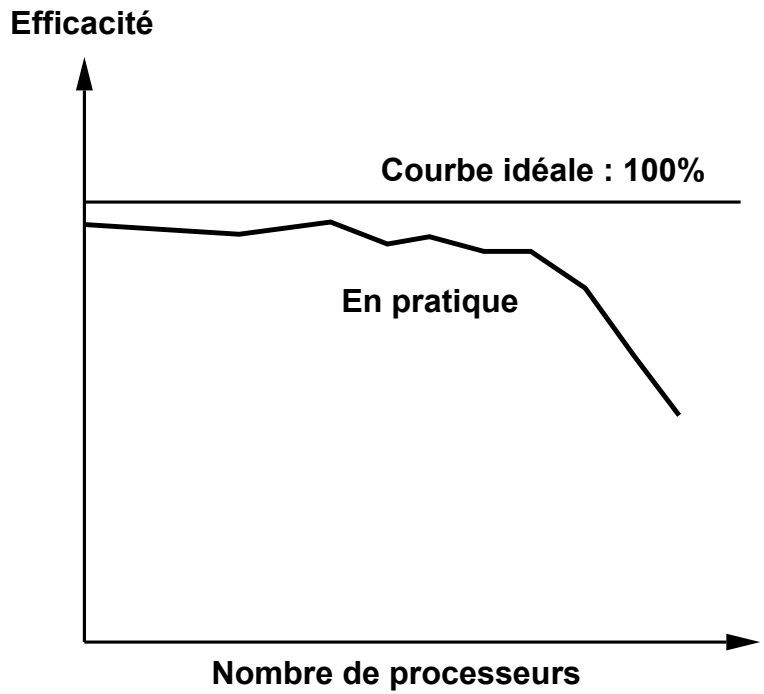


Accélération et efficacité

- Les deux sont une mesure de la qualité de la parallélisation
- Soit `T(p)` le temps d'exécution sur `p` processeurs
- *L'Accélération* `A(p)` et *l'Efficacité* `E(p)` sont définies comme étant :
  A(p) = T(1) / T(p) (p=1, 2, 3, ...)
  E(p) = A(p) / p
- Pour une accélération parallèle parfaite on obtient :
  T(p) = T(1) / p
  A(p) = T(1) / T(p) = T(1) / (T(1) / p) = p
  E(p) = A(p) / p = p / p = 100%

Accélération et efficacité





- Les programmes scalables demeurent efficaces pour un grand nombre de processeurs
  (scalables = passage à l'échelle)

Scalabilité

⇒ Propriété d'une application à être exécutée efficacement sur un très grand nombre de processeurs

⇒ Raisons possibles d'une faible scalabilité :

- La "machine" parallèle employée : architecture inadaptée, charge, ...
- Le programme parallélisé : analyser le comportement, améliorer les performances
- un peu des deux ...

Loi d'Amdahl

- Plus une règle qu'une loi
- Principe :

La partie non parallèle d'un programme limite les performances et fixe une borne supérieure à la scalabilité

Loi d'Amdahl

- Prenons un exemple avec une parallélisation de l'ordre de 80% du code :

  `T(1) = T(parallele) + T(séquentiel) = 80 + 20`

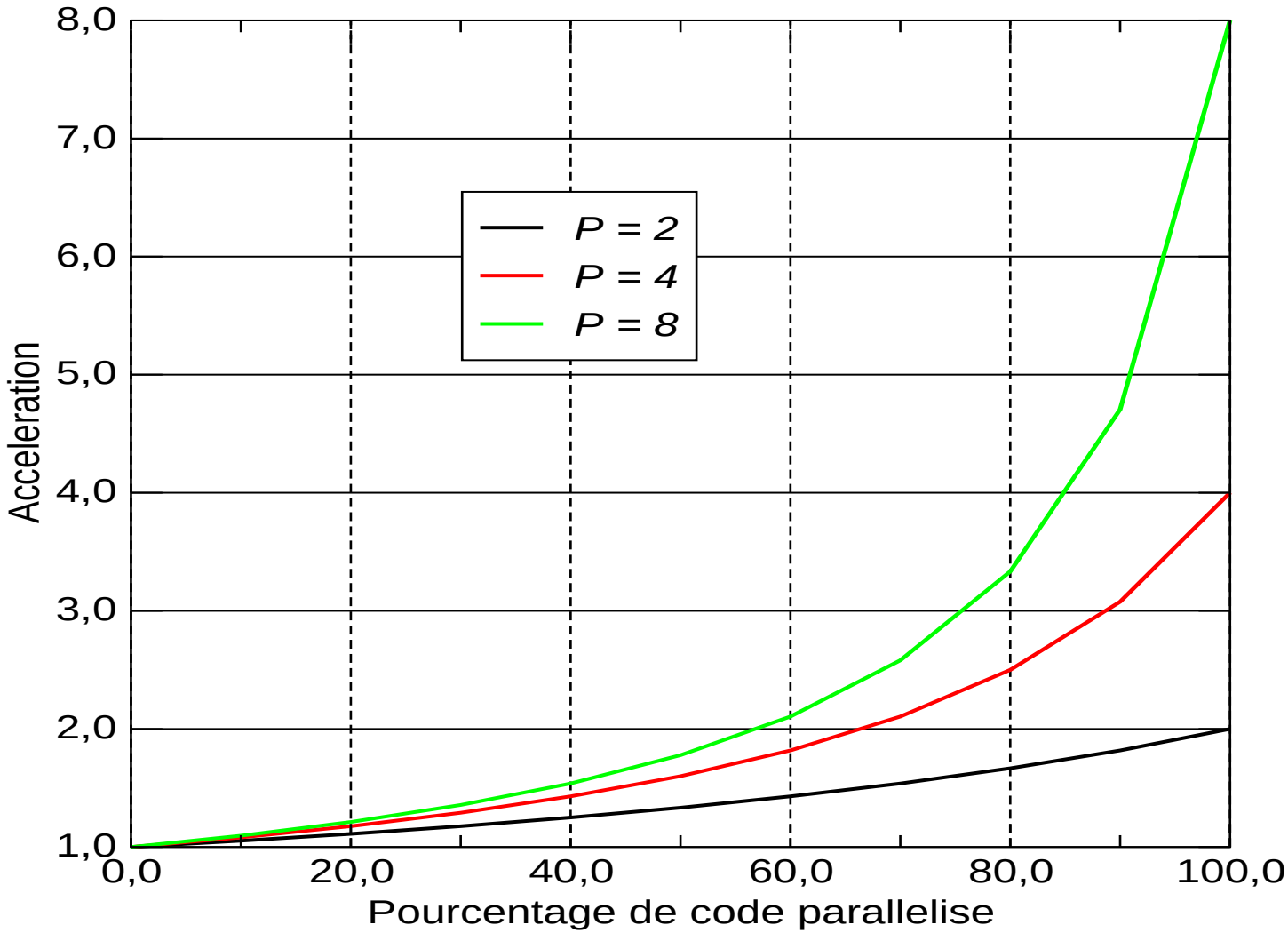
- Sur p processeurs, on obtient : `T(p) = 80 / p + 20`

- Quel que soit p, `T(p) > 20`

- `A(p) = T(1) / T(p) = 100 / ( 80 / p + 20 ) < 5`

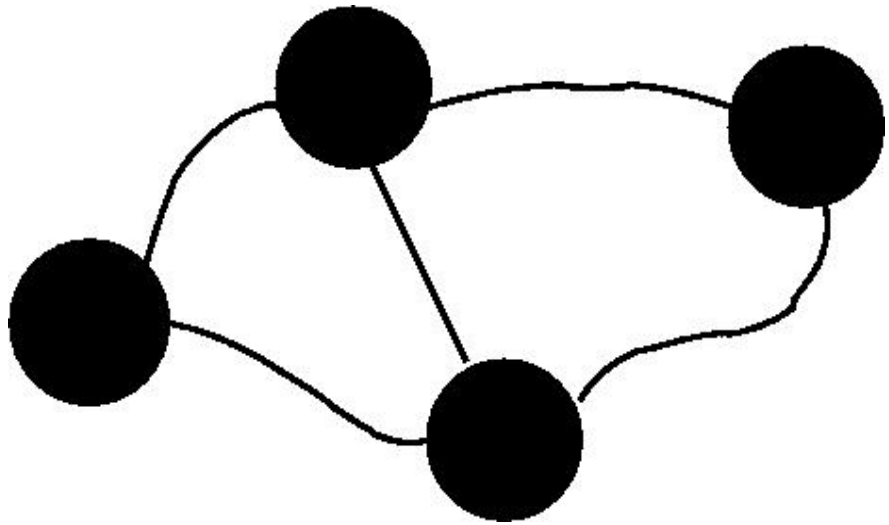
  `E(p) = ( A(p) / p ) < ( 5 / p )` $\xrightarrow{p \to +\infty}$ `0`!

Loi d'Amdahl


8,0
7,0
6,0
5,0
4,0
3,0
2,0
1,0
Acceleration
P = 2
P = 4
P = 8
0,0
20,0
40,0
60,0
80,0
100,0
Pourcentage de code parallelise

Interconnexion

- Pour un programmeur, un point essentiel réside dans le type de mémoire de l'architecture cible : mémoire partagée ou mémoire distribuée

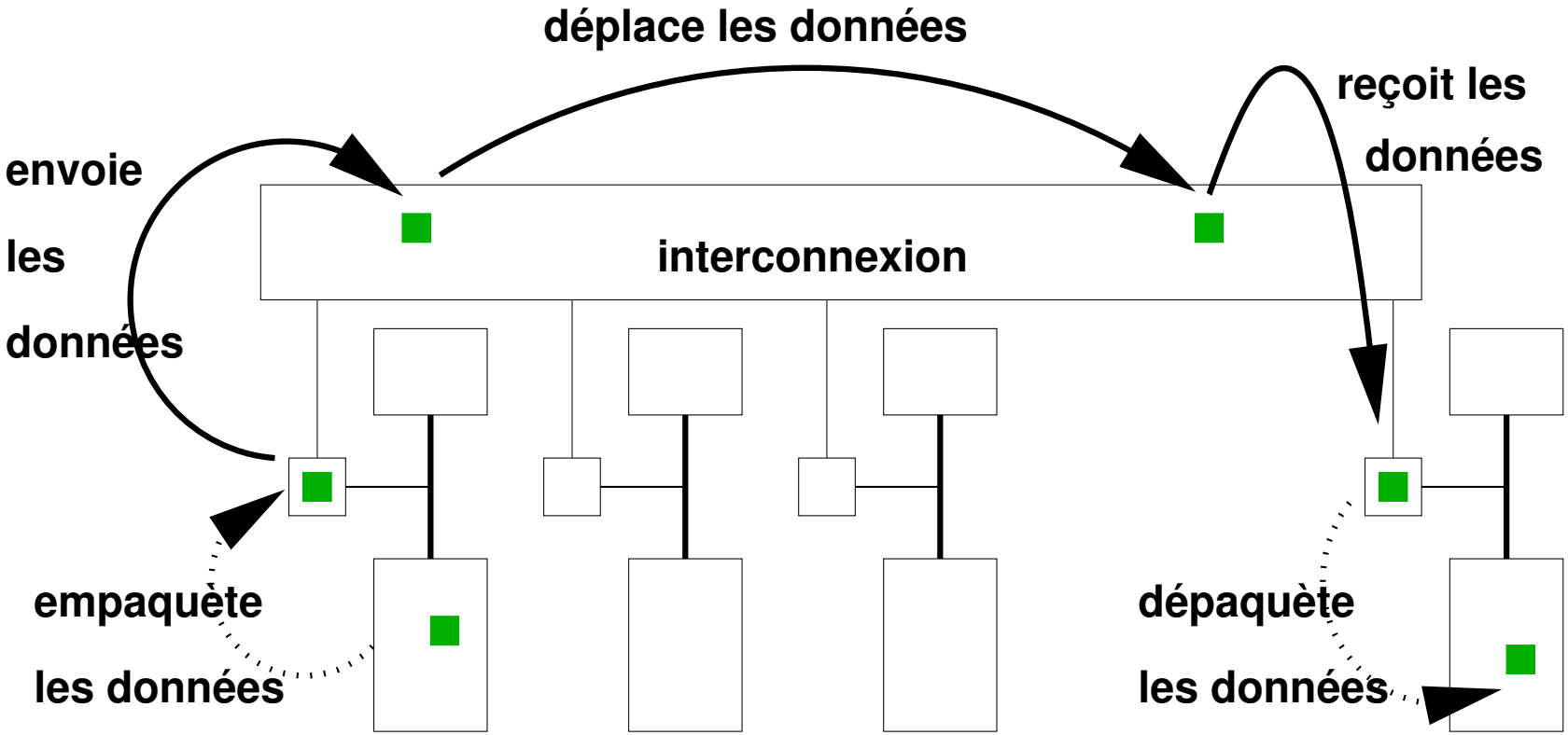


- L'interconnexion a un grand impact sur :
  - le choix de l'implémentation de la parallélisation
  - la performance parallèle
  - la facilité d'utilisation

- L'application elle-même, va influer sur la relative importance de ces facteurs

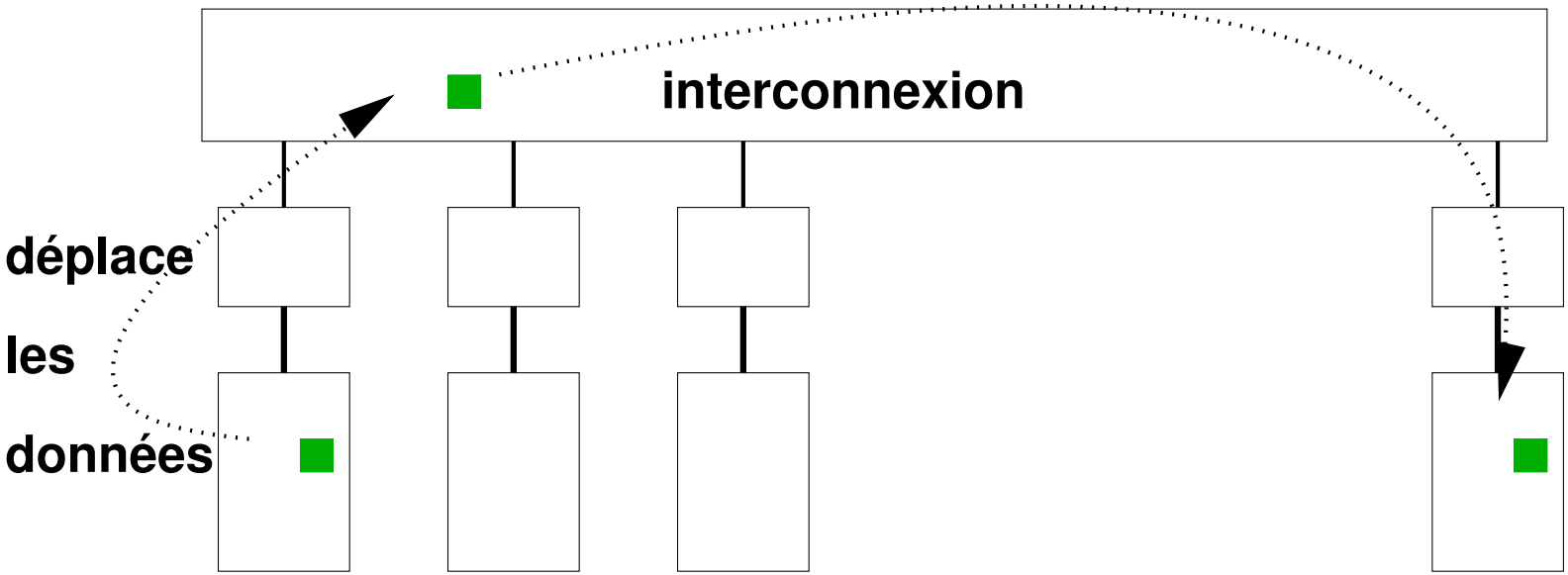
Ordinateur à mémoire distribuée

- Le système est programmé en utilisant l'échange de messages
- protocole réseau



Ordinateur à mémoire partagée

- Le mouvement des données est transparent pour l'utilisateur
- transfert mémoire



S.M.P. = Symmetric Multi-Processor

(tous les processeurs sont *égaux*)

Types de parallélisme

- Parallélisme de tâches
  Plusieurs traitements
  Traitements indépendants
  $\Rightarrow$ Echanges de données, directives de contrôle
  Le parallélisme découle de la réalisation simultanée de différents traitements sur les données

- Parallélisme de données
  Données régulières et en grand nombre
  Traitement identique sur chaque donnée
  $\Rightarrow$ Programmation Data-parallèle
  Le parallélisme découle de l'application simultanée du même traitement sur des données différentes

Parallélisme de tâches : passage de messages avec MPI

- Il repose sur l'échange de messages entre les processus pour le transfert de données, les synchronisations, les opérations globales
- La gestion de ces échanges est réalisée par MPI (Message Passing Interface)
- Cet ensemble repose sur le principe du SPMD (*Single Program Multiple Data*)
- Chaque processus dispose de ses propres données, sans accès direct à celles des autres
- Explicite, cette technique est entièrement à la charge du développeur
- Ces échanges qui impliquent deux ou plusieurs processus se font dans un communicateur
- Chaque processus est identifié par son rang, au sein du groupe

Environnement

- Initialisation en début (MPI_INIT)
- Finalisation en fin de programmme (MPI_FINALIZE)

```
INTEGER :: nbprocs, myrank
INTEGER :: ierr = 0
CALL MPI_INIT ( ierr )
CALL MPI_COMM_SIZE ( MPI_COMM_WORLD, nbprocs, ierr )
CALL MPI_COMM_RANK ( MPI_COMM_WORLD, myrank, ierr )
...
CALL MPI_FINALIZE ( ierr )
```

- Pour réaliser des opérations impliquant des données d'autres processus, il est nécessaire d'échanger ces informations aux travers de messages
- Ces messages se font sous la forme de communications impliquant au moins deux processus
- On peut faire une analogie avec le courrier électronique

Structures de données

- Les données transmises sont typées
- Types prédéfinis : `MPI_INTEGER`, `MPI_REAL`, ...
- Type homogène :
  - données contigües : `MPI_TYPE_CONTIGUOUS`
    ⇒ colonne de matrice en Fortran
  - données distantes d'un pas constant : `MPI_TYPE_VECTOR` ou `MPI_TYPE_HVECTOR`
    ⇒ ligne ou bloc d'une matrice
  - données distantes d'un pas variable : `MPI_TYPE_INDEXED` ou `MPI_TYPE_HINDEXED`
    ⇒ triangle dans une matrice
  - portion de tableau multi-dimensionnel : `MPI_TYPE_CREATE_SUBARRAY`
- Type hétérogène :
  ⇒ Construction d'une structure : `MPI_TYPE_STRUCT`
- Validation d'un type : `MPI_TYPE_COMMIT`
- Destruction d'un type : `MPI_TYPE_FREE`

Communications point à point

- La communication point à point est une communication entre deux processus :
  $\Rightarrow$ expéditeur et destinataire
- Composition d'un message :
  le communicateur (comm)
  les deux identifiants (src et dest)
  la donnée (buf), son type (datatype) et sa taille (count)
  une étiquette (tag) qui permet au programme de distinguer différents messages

Communications synchrones (il existe des variantes ...) :

```
CALL MPI_SEND (buf, count, datatype, dest, tag,
               comm, ierr)

CALL MPI_RECV (buf, count, datatype, src , tag,
               comm, status, ierr)
```

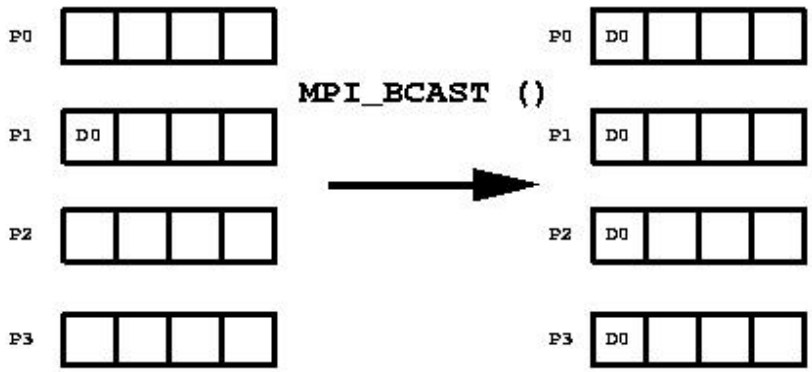
Communications collectives

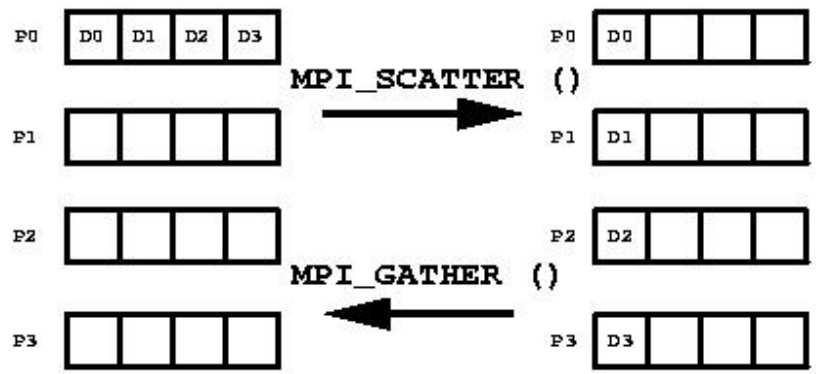
- La communication collective est une communication qui implique un ensemble de processus qui l'effectuent tous
- **les synchronisations globales** : une barrière de synchronisation qui agit sur l'ensemble des membres d'un communicateur.
  `CALL MPI_BARRIER (comm, ierr)`

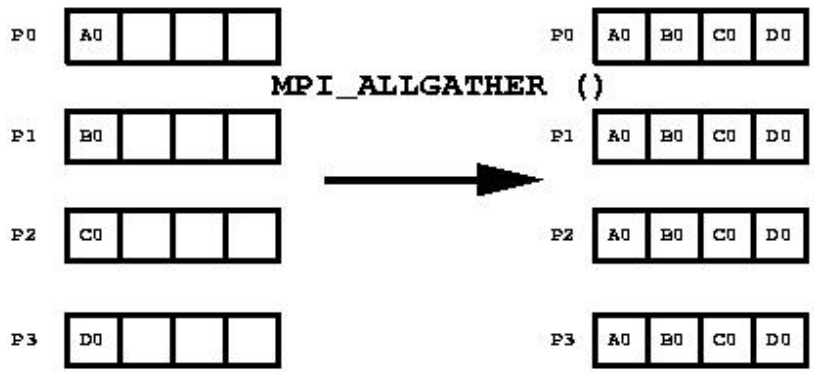
- **les opérations de réduction sur des données réparties** : somme, produit, maximum ... et le résultat peut être ensuite redistribué (`MPI_ALLREDUCE`) ou pas (`MPI_REDUCE`).

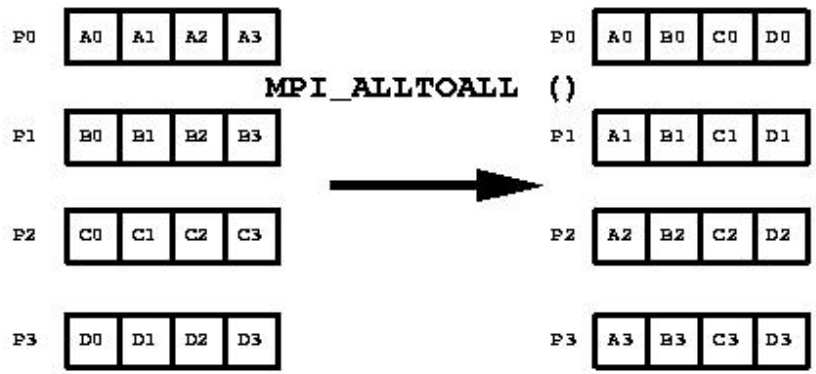
```
CALL MPI_REDUCE (sbuf, rbuf, count, datatype,
                 oper, root, comm, ierr)

CALL MPI_ALLREDUCE (sbuf, rbuf, count, datatype,
                    oper, comm, ierr)
```

Diffusion / collecte



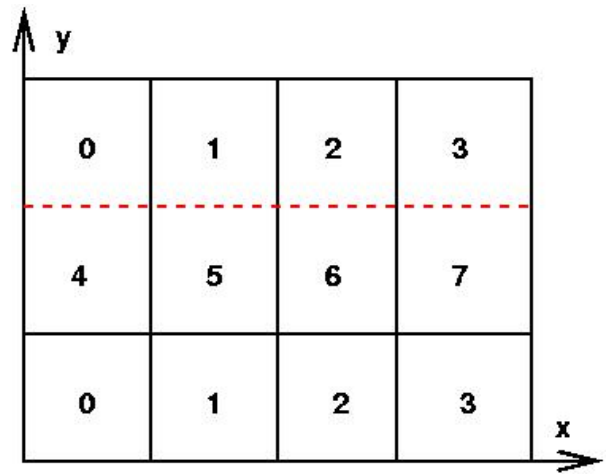






Topologie

- Topologie cartésienne : grille de processus
- Nombre de processus par dimension d'espace : `MPI_DIMS_CREATE`
- Création de la grille : `MPI_CART_CREATE`
  ⇒ périodicité ou non des conditions aux limites
  ⇒ création d'un nouveau communicateur



- Recherche des voisins dans chaque dimension : `MPI_CART_SHIFT`
- Coordonnées / rang : `MPI_CART_COORDS` et `MPI_CART_RANK`

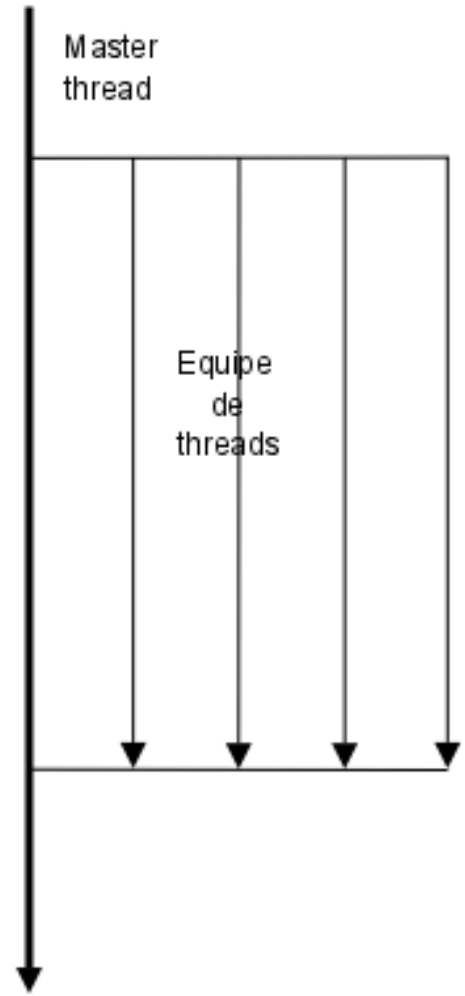
Evolution de MPI : apport de MPI-2

- Gestion dynamique des processus
- Entrées / sorties parallèles : MPI I/O
- Communications / copies de mémoire à mémoire
- Interface Fortran 95 (`mpi`), C++
- Module MPI

Parallélisme à mémoire partagée avec OpenMP

OpenMP est un ensemble de constructions parallèles basées sur des directives de compilation pour architecture à mémoire partagée.

- Il est basé sur le principe du fork and join
- Une équipe de threads est créée à l'entrée d'une région parallèle
- Son effectif est contrôlé par variable d'environnement ou appel librairie
- Avant et après, l'exécution est séquentielle
- Les threads sont identifiées par leur rang et celui de la *thread master* est 0



Directives

La parallélisation se fait en insérant des directives dans le code séquentiel pour construire des régions parallèles :

```
!$OMP PARALLEL [clause [,clause] ...]
nbth = 1
!$     nbth = OMP_GET_NUM_THREADS()
```

Où :

`!$OMP` ou `C$OMP`	est une sentinelle, obligatoire
`#pragma` en C/C++	comme préfixe des directives
`!$` ou `C$`	permet d'effectuer une compilation conditionnelle si OpenMP est reconnu
`PARALLEL`	est une directive
`[clause [,clause] ...]`	sont les clauses, optionnelles, qui permettent de décrire la visibilité des variables

Variables privées

Les variables `PRIVATE` et `FIRSTPRIVATE` ont une adresse unique dans chaque thread parallèle : elles sont dupliquées.

Elles ne sont pas accessibles en dehors de leur région parallèle.

Seules, celles déclarées comme `FIRSTPRIVATE` sont initialisées au début de la région.

Les variables privées qui conservent leur valeur d'une région parallèle à une autre sont appelées `THREADPRIVATE`.

Les variables `THREADPRIVATE` peuvent être initialisées par la thread master au travers de la clause `COPYIN`.

Exemples : indices de boucle (par défaut), variables locales, tableaux automatiques, ...

Variables partagées

Par défaut dans les régions parallèles, toutes les variables sont partagées (ou publiques).

La déclaration se fait avec la clause `SHARED`.

Toutes les threads accèdent à la même instance de la variable (attention aux conflits !).

Certaines variables doivent être partagées, comme les variables des opérations de réduction, i.e. opérations associatives usuelles (produit scalaire, maximum, somme, ET logique) sur des données partagées.

Exemples : variables locales rémanentes (`DATA`, `SAVE`), celles déclarées dans des commons ou modules, ou encore passées en argument sont partagées.

Autres objets OpenMP

D'autres objets existent :

- clauses : FIRSTPRIVATE, LASTPRIVATE, COPYPRIVATE, COPYIN,...

- directives : THREADPRIVATE, ...

- sous-programmes : OMP_SET_NUM_THREADS, OMP_GET_WTIME, OMP_GET_NUM_THREADS, OMP_GET_THREAD_NUM, ...

- variables d'environnement : OMP_NUM_THREADS, OMP_SCHEDULE, OMP_DYNAMIC, ...

Un module destination du Fortran OMP_LIB, un fichier d'en-tête pour le C/C++ omp.h, permettent de définir les prototypes des fonctions OpenMP

Partage du travail 1/3

Les constructions parallèles permettent de partager le travail entre les threads.

Des directives contrôlent la répartition du travail et des données.

La directive `DO [clause[,clause...]]` ... `END DO [nowait]` répartit les itérations de la boucle "finie" qui la suit immédiatement.

Elle n'a pas de synchronisation en entrée et la clause `NOWAIT` permet de lever celle en sortie.

La clause `SCHEDULE` décide du mode de répartition de ses itérations.

Une région parallèle peut se limiter à une boucle parallèle.

Boucle dans une région parallèle :

```
!$OMP DO
      DO i = 1, n
         CALL mywork(i)
      END DO
!$OMP END DO
```

Construction parallèle restreinte à une triple boucle :

```
!$OMP PARALLEL DO DEFAULT(NONE) &
!$OMP SHARED (x, y, z, nx, ny, nz) &
!$OMP PRIVATE (i, j, k)
      DO k = 1, nz
         DO j = 1, ny
            DO i = 1, nx
               x(i,j,k) = y(i,j,k) * z(i,j,k)
            END DO
         END DO
      END DO
!$OMP END PARALLEL DO
```

Partage du travail 2/3

Une section est une portion de code exécutée par une seule thread.

On peut définir un ensemble de sections avec la directive

```
SECTIONS [clause[,clause...]]
SECTION
...
SECTION
...
END SECTIONS[nowait]
```

On répartit ainsi plusieurs portions indépendantes entre les threads.

Faire attention à avoir une bonne correspondance entre nombre de threads, nombre de sections et tenir compte de leur poids respectif pour l'équilibrage de charge.

Une région parallèle peut se limiter à un groupe de sections.

Région parallèle limitée à une clause SECTIONS :

```
!$OMP PARALLEL SECTIONS DEFAULT(NONE) &
!$OMP SHARED (x, y, z, nx, ny, nz) &
!$OMP SECTION
      CALL WORK_X (x, nx)
!$OMP SECTION
      CALL WORK_Y (y, ny)
!$OMP SECTION
      CALL WORK_Z (z, nz)
!$OMP END PARALLEL SECTIONS
```

Partage du travail 3/3

La directive `WORKSHARE` est un apport d'OpenMP 2.

Elle est utile pour répartir le travail des contructions Fortran95 comme :

- opérations sur les variables tableaux
- instruction `FORALL`, `WHERE`

L'unité de partage du travail est l'élément d'un tableau.

Le partage peut être sous-optimal et dégrader les performances.

Une région parallèle peut se limiter à une construction `WORKSHARE`.

Construction WORKSHARE

```
!$OMP PARALLEL DEFAULT(NONE) &
!$OMP SHARED (x, y, nu, nv, epsilon) &
!$OMP PRIVATE (i, j)
!$OMP DO
      DO j = 1, nv
         DO i = 1, nu
            y(i,j) = y(i,j) + 1.0
         END DO
      END DO
!$OMP END DO

!$OMP WORKSHARE
      WHERE ( abs(y(:,:) ) > epsilon) &
          x(:,:) = 1.0 / y(:,:)
!$OMP END WORKSHARE NOWAIT
!$OMP END PARALLEL
```

Sérialisation et synchronisations

Il y a plusieurs constructions qui permettent de restreindre ou spécifier l'ordre d'accès à des données partagées

- directive `MASTER ... END MASTER`
  accès pour la thread de rang 0 uniquement
- directive `SINGLE ... END SINGLE [nowait]`
  accès pour une seule thread, non déterminée à l'avance
- directive `ATOMIC`
  section critique formée d'une seule instruction
- section `CRITICAL ... END CRITICAL`
  accès pour une seule thread à la fois
- directive `BARRIER`
  barrière de synchronisation globale

Passage de messages ou mémoire partagée ?

$\Rightarrow$ quelques observations ...

Passage de Messages	Mémoire partagée
Demande plus de temps à implémenter	Plus facile à implémenter
Plus de détails à prendre en compte	Le système gère pas mal de détails
Augmente la taille du code	Faible accroissement de la taille du code
Plus de maintenance	Peu de maintenance additionnelle
Complexe à déboguer et à optimiser	Plus facile à déboguer et à optimiser
Augmente le volume mémoire	Usage efficace de la mémoire
Meilleur pour une scalabilité optimale	Scalable, mais ....
Parallélisme visible	Parallélisme traité par le compilateur

Exemple récapitulatif : définition du problème

- Les équations ...

$$\begin{cases} \frac{\partial u}{\partial t} - \lambda \Delta u = f & \text{dans } \Omega, \qquad \text{avec } \Omega = [0,1]^3 \\ u = 0 & \text{sur } \Gamma = \partial\Omega \end{cases}$$

- La parallélisation est faite :
  par décomposition de domaine avec MPI,
  par partage du travail avec OpenMP.

- La discrétisation spatiale est faite par un schéma aux différences finies d'ordre deux à trois points
  $\Rightarrow$ maillage uniforme noté x_i, j, k, de pas d'espace h_x, h_y, h_z

- La discrétisation temporelle est faite par le schéma de Crank-Nicholson, ordre 2 semi-implicite et inconditionnellement stable dans notre cas, avec dt le pas de la discrétisation.

$$\{I - 0.5\lambda dt\Delta\}\, u^{n+1} = \{I + 0.5\lambda dt\Delta\}\, u^n + 0.5dt \left\{ f^{n+1} + f^n \right\}$$

Exemple récapitulatif :définition du problème

- Le laplacien est ainsi discrétisé :

$$\Delta_h u(i,j,k) = \frac{u_{i-1,j,k} - 2.u_{i,j,k} + u_{i+1,j,k}}{{h_x}^2} + \frac{u_{i,j-1,k} - 2.u_{i,j,k} + u_{i,j+1,k}}{{h_y}^2} + \frac{u_{i,j,k-1} - 2.u_{i,j,k} + u_{i,j,k+1}}{{h_z}^2} + (\text{termes d'erreurs en } h^2).$$

- Le système global s'écrit alors sous la forme :

$${u_{i,j,k}}^{n+1} - 0.5\lambda dt \left\{ \frac{{u_{i-1,j,k}}^{n+1} - 2{u_{i,j,k}}^{n+1} + {u_{i+1,j,k}}^{n+1}}{{h_x}^2} + \frac{{u_{i,j-1,k}}^{n+1} - 2{u_{i,j,k}}^{n+1} + {u_{i,j+1,k}}^{n+1}}{{h_y}^2} + \frac{{u_{i,j,k-1}}^{n+1} - 2{u_{i,j,k}}^{n+1} + {u_{i,j,k+1}}^{n+1}}{{h_z}^2} \right\}$$

$$= {u_{i,j,k}}^{n} + 0.5\lambda dt \left\{ \frac{{u_{i-1,j,k}}^{n} - 2{u_{i,j,k}}^{n} + {u_{i+1,j,k}}^{n}}{{h_x}^2} + \frac{{u_{i,j-1,k}}^{n} - 2{u_{i,j,k}}^{n} + {u_{i,j+1,k}}^{n}}{{h_y}^2} + \frac{{u_{i,j,k-1}}^{n} - 2{u_{i,j,k}}^{n} + {u_{i,j,k+1}}^{n}}{{h_z}^2} \right\} + 0.5dt \left\{ {f_{i,j,k}}^{n+1} + {f_{i,j,k}}^{n} \right\}$$

Algorithme séquentiel

- Initialisations :
  $\longrightarrow$ initialisations des vecteurs

- Boucle en temps
  1. Construction du second membre
  2. Résolution du système linéaire : **gradient conjugué**
     Boucle de convergence :
     $\longrightarrow$ Produit matrice-vecteur (**pmv**)
     $\longrightarrow$ Produits scalaires (**prodscal**)
     $\longrightarrow$ Combinaisons linéaires de vecteurs (**saxpy**)
  3. Avancement en temps

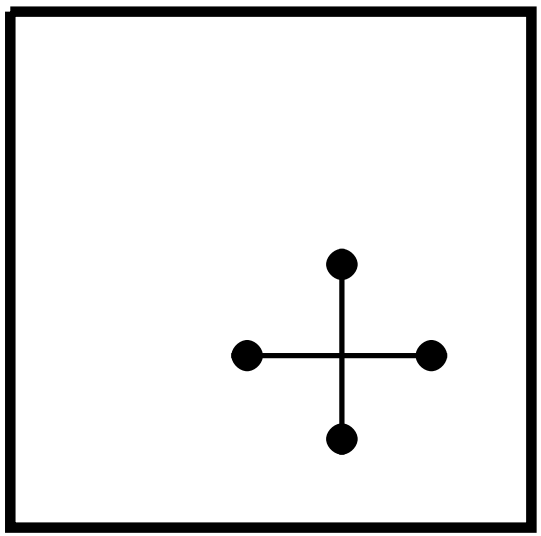
- Finalisations

Décomposition de domaine

- Partage du domaine de calcul en plusieurs morceaux, appelés sous-domaines : la somme directe forme le domaine initial.
- Parallélisation explicite (travail à la charge du programmeur).
- Pour maillages structurés ou non structurés, avec des nœuds ou des cellules de calcul.
- Grille de processus sur la grille des sous-domaines avec une topologie MPI.
- Peut se faire quel que soit le nombre de processus.
- Gestion du stockage et des synchronisations pour éviter les écrasements intempestifs.

Analyse des dépendances

- Comment les algorithmes accèdent aux données ?
  $\longrightarrow$ Schéma aux différences finies

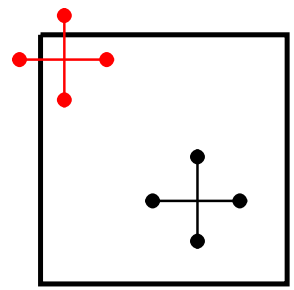


- Quelles données sont nécessaires ?
  $\longrightarrow$ Pour calculer une nouvelle valeur de quoi ai-je besoin ?

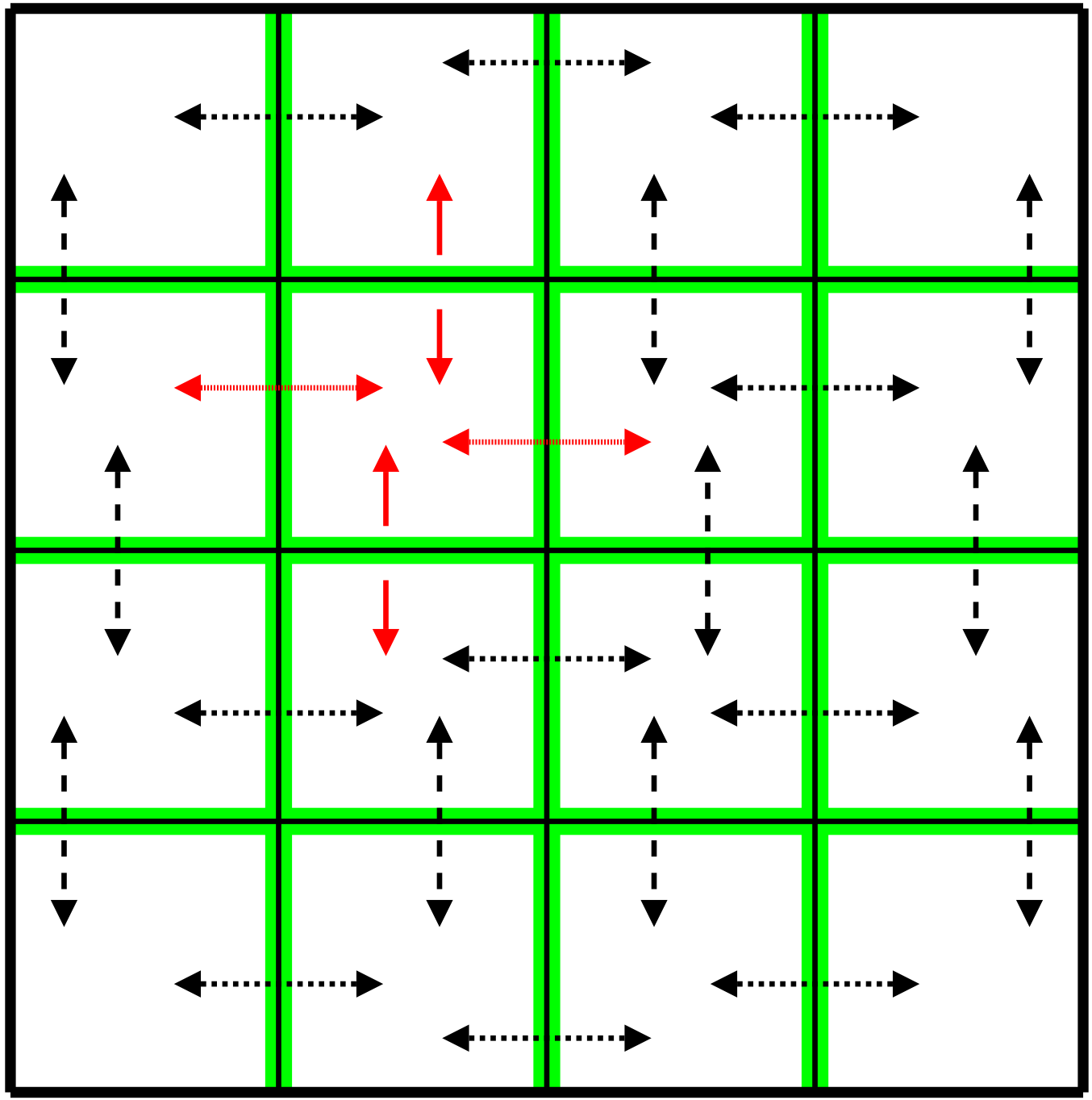
- Quels algorithmes ont des dépendances avec les voisins ?
  $\longrightarrow$ Est-ce que tout ce dont j'ai besoin est chez moi ?

Produit matrice vecteur ? (**pmv**)

```
c1 = ... ; c2 = ...
c3 = ... ; c4 = ....
DO k = sz, ez
   DO j = sy, ey
      DO i = sx, ex
         a(i,j,k) = c4 * b(i,j,k)
          + c1 * ( b(i+1,j  ,k  ) + b(i-1,j  ,k  ) )
          + c2 * ( b(i  ,j+1,k  ) + b(i  ,j-1,k  ) )
          + c3 * ( b(i  ,j  ,k+1) + b(i  ,j  ,k-1) )
      END DO
   END DO
END DO
```



→ Dépendance d'un rang tout autour



Combinaison linéaire de vecteurs ? (**saxpy**)

```
DO k = sz, ez
   DO j = sy, ey
      DO i = sx, ex
         a(i,j,k) = scala * a(i,j,k) + scalb * b(i,j,k)
      END DO
   END DO
END DO
```

→ Pas de dépendance

Produit scalaire (**prodscal**)

```
DO k = sz, ez
   DO j = sy, ey
      DO i = sx, ex
         rnorm = rnorm + a(i,j,k) * b(i,j,k)
      END DO
   END DO
END DO
```

→ Dépendance globale pour le cumul (réduction)

```
CALL MPI_ALLREDUCE (rnorm, gnorm, 1, ITYPE_REAL,
                    MPI_SUM, comm3d, ierr)
```

N.B. : ITYPE_REAL = MPI_REAL ou MPI_DOUBLE_PRECISION

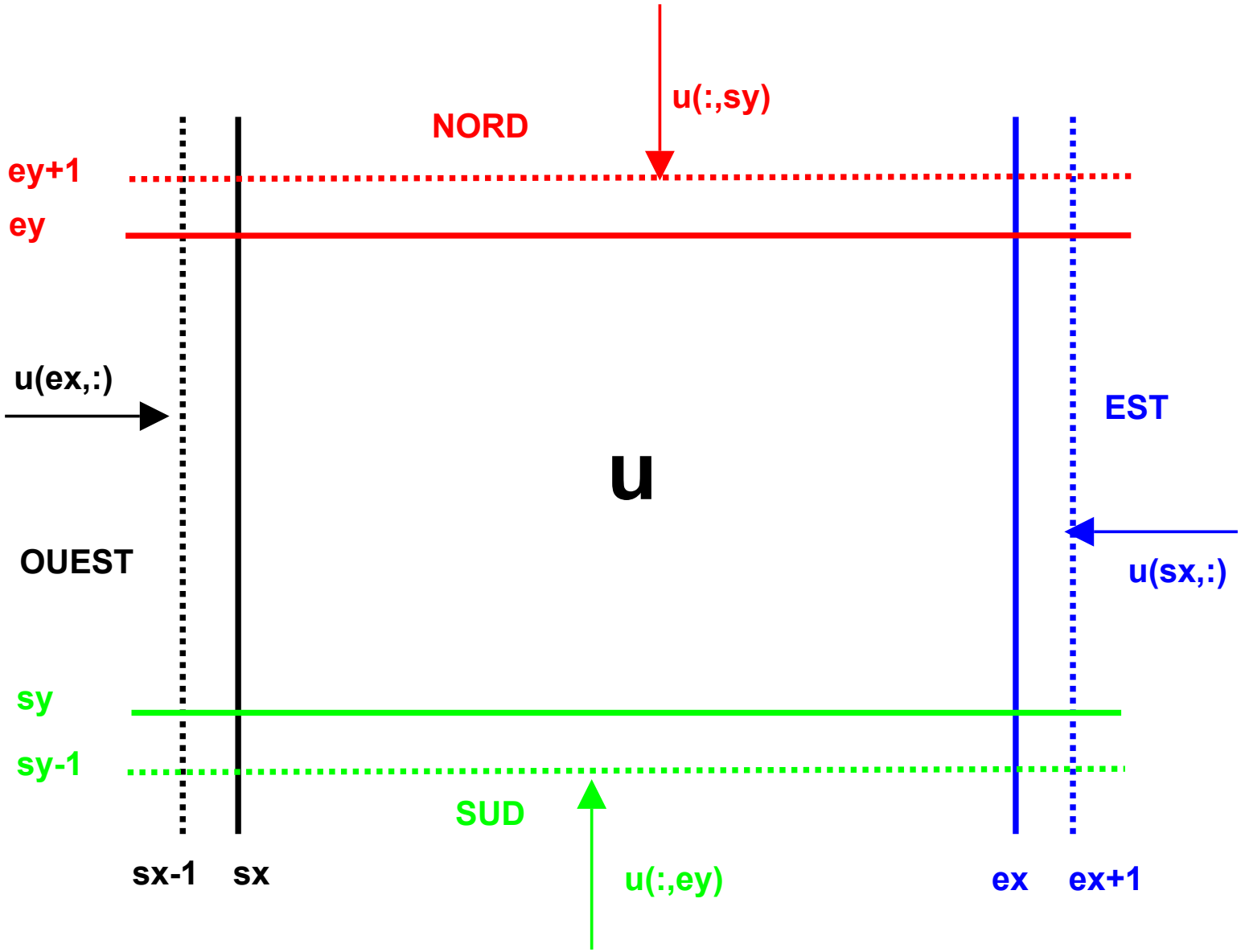
Sous-domaine par processus (1/2)

- Nœuds ou cellules fantômes nécessaires pour calculer les valeurs aux interfaces entre sous-domaines (conditions aux limites pour les bords extérieurs)

- Mise à jour de ces cellules à chaque pas de temps avec les valeurs calculées par le processus qui traite le sous-domaine propriétaire de ces cellules.

- Calcul automatique des dimensions locales, des espaces pour les nœuds fantômes, ...
  indices locaux de calcul : de sx à ex ;
  indices locaux de stockage : de (sx-1) à (ex+1)
  (NB : longueur = ex-sx+3)

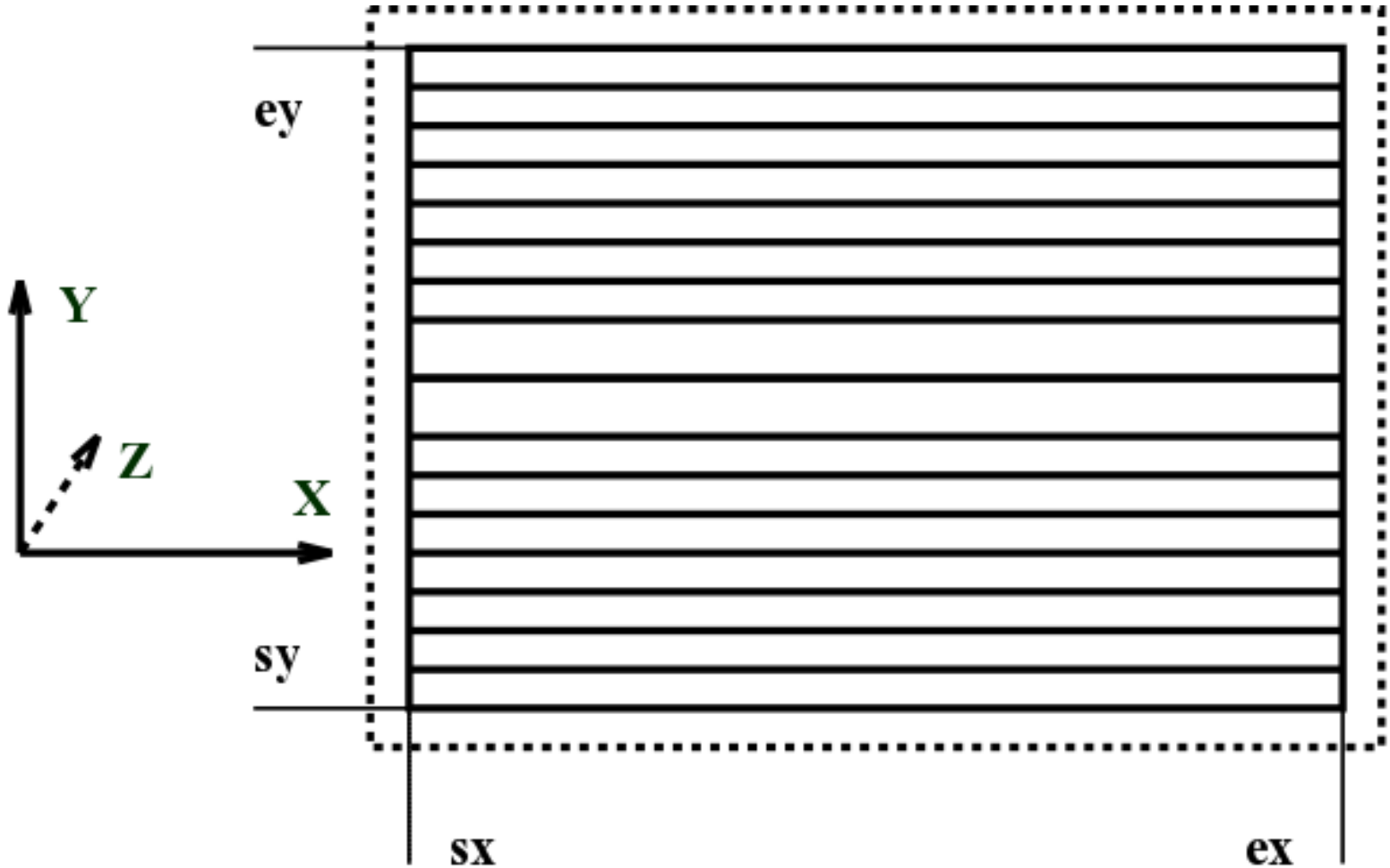
Sous-domaine par processus (2/2)



Types de données des messages

- Les sous-domaines sont des cubes
- Les zones d'interface des surfaces sont des sections des plans xOy, xOz et yOz
- On construit des types MPI pour envoyer facilement ces rectangles de données

Plan xOy (1/2)



Il s'agit de (**ey-sy+1**) vecteurs régulièrement espacés

$\Rightarrow$ longueur d'un vecteur : **ex-sx+1**,

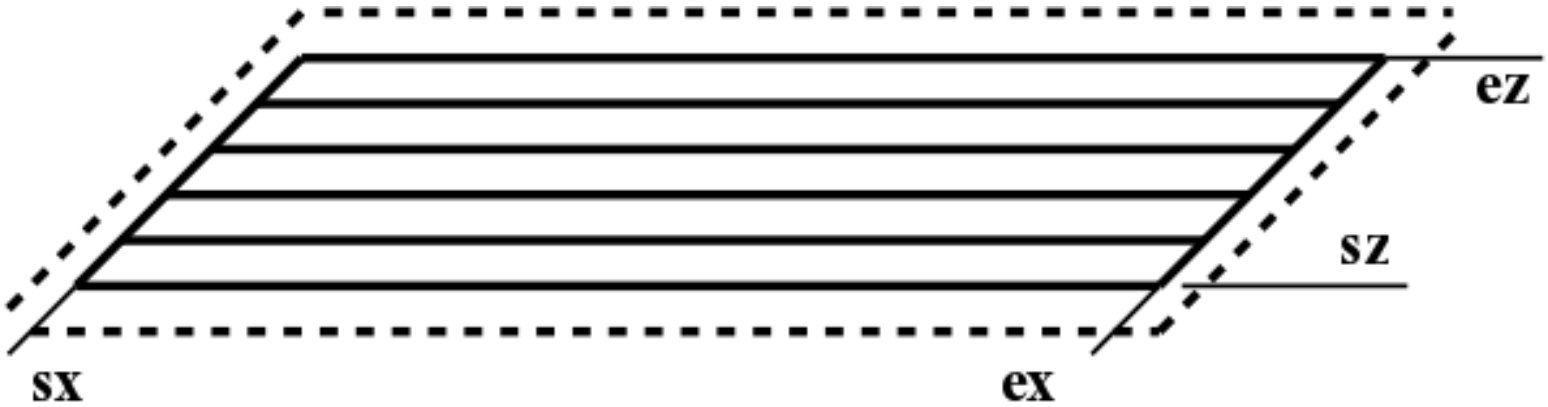
$\Rightarrow$ pas entre deux débuts consécutifs : **ex-sx+3**

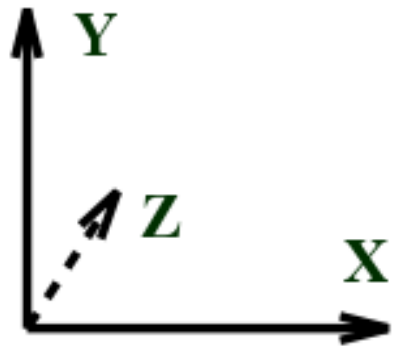
Plan xOy (2/2)

- On crée le type **ip_xy** avec la fonction `MPI_TYPE_VECTOR` :

```
CALL MPI_TYPE_VECTOR(
ey-sy+1,          nombre de blocs
ex-sx+1,          longueur d'un bloc
ex-sx+3,          pas entre le début de
ITYPE_REAL,       deux blocs consecutifs
ip_xy, ierr )
CALL MPI_TYPE_COMMIT(ip_xy,ierr)
```

Plan xOz (1/2)





Il s'agit de (**ez-sz+1**) vecteurs
régulièrement espacés :

⇒ longueur d'un vecteur : **ex-sx+1**,

⇒ pas entre deux débuts consécutifs : **(ex-sx+3)*(ey-sy+3)**

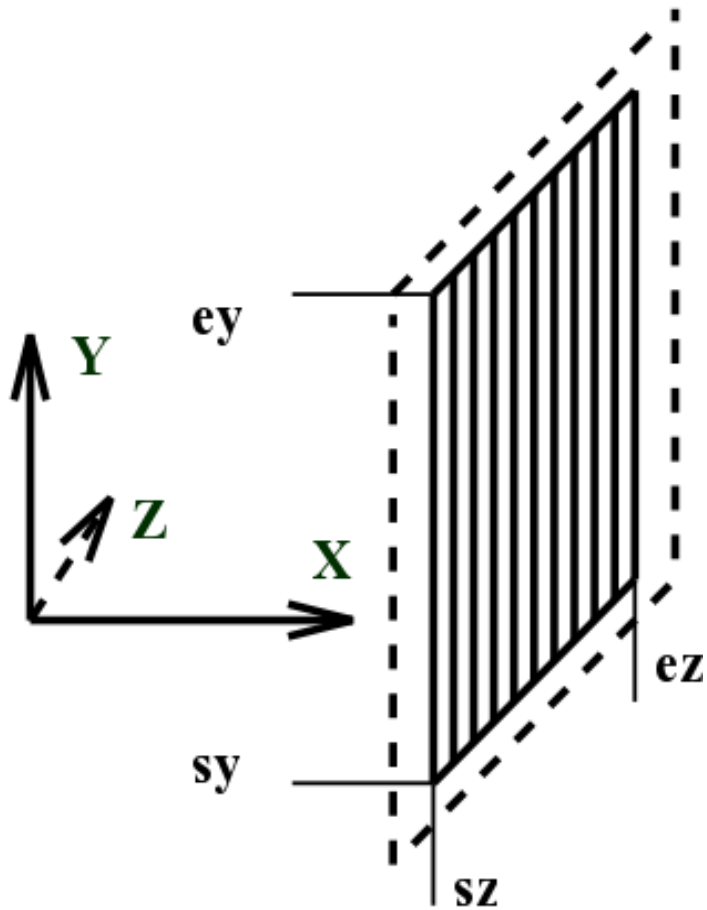
⇒ on crée le type **ip_xz** avec la fonction `MPI_TYPE_VECTOR`

Plan xOz (2/2)

- On crée le type **ip_xz** avec la fonction `MPI_TYPE_VECTOR` :

```
CALL MPI_TYPE_VECTOR(
ez-sz+1,                    nombre de blocs
ex-sx+1,                    longueur d'un bloc
(ex-sx+3)*(ey-sy+3),        pas entre le début de
ITYPE_REAL,                 deux blocs consécutifs
ip_xz, ierr )
CALL MPI_TYPE_COMMIT(ip_xz,ierr)
```

Plan yOz (1/2)



Pour le plan **yOz**, il s'agit d'éléments isolés irrégulièrement espacés
$\Rightarrow$ on utilise le constructeur de sous-tableau, apport de MPI-2.

Plan yOz (2/2)

```
!profil du tableau u a partir duquel on va extraire un
sous-tableau
   profil_tab = (/ex-sx+3, ey-sy+3, ez-sz+3 /)
!profil du sous-tableau extrait
   profil_sous_tab = (/ 1, ey-sy+1, ez-sz+1 /)
!coordonnees envoi a W / reception de E
   coord_envoie_W = (/ 1       , 1, 1 /)    ! (/ sx   , sy, sz /)
   coord_recoit_E = (/ ex-sx+2, 1, 1 /)     ! (/ ex+1, sy, sz /)
!coordonnees envoi a E / reception de W
   coord_envoie_E = (/ ex-sx+1, 1, 1 /)     ! (/ ex   , sy, sz /)
   coord_recoit_W = (/ 0       , 1, 1 /)    ! (/ sx-1, sy, sz /)
!
!creation des types
   CALL MPI_TYPE_CREATE_SUBARRAY(3, profil_tab, profil_sous_tab,&
                  coord_envoie_W, MPI_ORDER_FORTRAN, ITYPE_REAL,&
                  type_envoie_W, ierr)

   CALL MPI_TYPE_COMMIT (type_envoie_W, ierr)

   ...
```

Communications

- Synchronisation des processus :
  ⇒ phase de calcul
  ⇒ phase de communication
- Échange entre les processus 2 à 2 : `MPI_SENDRECV`
  ⇒ `MPI_SEND` et `MPI_RECV` en une seule communication

```
CALL MPI_SENDRECV(
a(sx,sy,sz  ), 1, ip_xy, voisin(AV), tag1,   envoi au voisin AV
a(sx,sy,ez+1), 1, ip_xy, voisin(AR), tag1,   réception du voisin AR
comm3d, status, ierr )                       type prédéfini ip_xy, plan xOy

CALL MPI_SENDRECV(
a(sx,sy,ez  ), 1, ip_xy, voisin(AR), tag2,   envoi au voisin AR
a(sx,sy,sz-1), 1, ip_xy, voisin(AV), tag2,   réception du voisin AV
comm3d, status, ierr )                       type prédéfini ip_xy, plan xOy
```

- De même pour les deux autres paires de faces

Algorithme MPI

- Initialisations :
  - $\longrightarrow$ construction de la topologie
  - $\longrightarrow$ construction des types MPI
  - $\longrightarrow$ initialisations des vecteurs locaux
  - $\longrightarrow$ diffusion des constantes / paramètres $\longrightarrow$ communications
- Boucle en temps
  1. remplissage des cellules fantômes $\longrightarrow$ communications
  2. construction du second membre
  3. résolution du système linéaire :
     - $\Rightarrow$ communications / synchronisations
     - $\longrightarrow$ produit matrice-vecteur
     - $\longrightarrow$ produits scalaires $\Rightarrow$ réduction
     - $\longrightarrow$ combinaisons linéaires de vecteurs (saxpy)
  4. avancement en temps
- Finalisations :
  - $\longrightarrow$ destruction des types MPI
  - $\longrightarrow$ destruction de la topologie

Partage du travail

- Boucles de calcul :
  parallélisation = distribution des itérations entre les tâches
- Parallélisation implicite (synchronisations, distribution des itérations gérées par la bibliothèque parallèle)
- Le travail de parallélisation est transmis au compilateur au travers de directives insérées dans le code source
- Après analyse des dépendances, les boucles sont parallélisées ou non
- Considérons donc les différentes boucles du programme ...

La boucle en temps

```
DO itg = 1, nbiter

   CALL scdmb (b, f1, f2, u, tps, rnudt2)

   tps = tps + dt

   CALL gradconj (u, b, p, q, r, it_max, prec,
                  rnorm_k, rnudt2)

END DO
```

- Dépendance entre les itérations,
  elle n'est pas parallélisable

La boucle du gradient conjugué

```
DO WHILE ( (.NOT. convergence) .AND. (it < it_max) )
   it = it + 1
   CALL pmv (q, p, - rnudt2)
   CALL prodscal (alpha_k, q, p)
   alpha_k = rnorm_k / alpha_k ; beta_k = rnorm_k

   CALL saxpy (u, one, p, alpha_k)
   CALL saxpy (r, one, q, - alpha_k)

   CALL prodscal (rnorm_k, r, r)
   beta_k = rnorm_k / beta_k
   CALL saxpy (p, beta_k, r, one)
   convergence = ( SQRT(rnorm_k) < prec)
END DO
```

- Dépendance entre les itérations + DO WHILE
  elle n'est pas parallélisable

Le produit matrice vecteur

```
c1 = .... ; c2 = ... ; c3 = ... ; c4 = ...

DO k = 1, ntz
   DO j = 1, nty
      DO i = 1, ntx
         a(i,j,k) = c4 * b(i,j,k) &
           + c1 * ( b(i+1,j  ,k  ) + b(i-1,j  ,k  ) ) &
           + c2 * ( b(i  ,j+1,k  ) + b(i  ,j-1,k  ) ) &
           + c3 * ( b(i  ,j  ,k+1) + b(i  ,j  ,k-1) )
      END DO
   END DO
END DO
```

- Indépendance entre les itérations (a,b partagés), **elle est parallélisable**

Le produit scalaire

```
rnorm = 0.0_8

DO k = 1, ntz
   DO j = 1, nty
      DO i = 1, ntx
         rnorm = rnorm + a(i,j,k) * b(i,j,k)
      END DO
   END DO
END DO
```

- Dépendance entre les itérations
- **Elle est tout de même parallélisable car il s'agit d'une opération de réduction**

La combinaison linéaire de vecteurs

```
DO k = 1, ntz
   DO j = 1, nty
      DO i = 1, ntx
         a(i,j,k) = scala * a(i,j,k) + scalb * b(i,j,k)
      END DO
   END DO
END DO
```

- Indépendance entre les itérations, **elle est parallélisable**

Bilan de l'analyse

- La boucle en temps : dépendance, **elle n'est pas parallélisable**
  $\longrightarrow$ Elle est exécutée par toutes les threads, en intégralité
- La boucle du gradient conjugué : dépendance, **elle n'est pas parallélisable**
  $\longrightarrow$ Elle est exécutée par toutes les threads, en intégralité
- Les produits matrice-vecteur : pas de dépendance, parallélisés
- Les produits scalaires : pas de dépendance, parallélisés
- Les combinaisons linéaires de vecteurs : pas de dépendance, parallélisées
- Création d'une région parallèle qui englobe la boucle en temps

La région parallèle

```
!$OMP PARALLEL DEFAULT( NONE )
!$OMP SHARED (u, b, f1, f2, p, q, r)
!$OMP SHARED (tps, dt, rnorm_k, prec)
!$OMP SHARED (nbiter, it_max, rnudt2)
!$OMP PRIVATE (itg)
DO itg = 1, nbiter
      CALL scdmb (b, f1, f2, u, tps, rnudt2)
!$OMP MASTER
      tps = tps + dt
      rnorm_k = zero
!$OMP END MASTER
      CALL gradconj (u, b, p, q, r, it_max, prec,
                     rnorm_k, rnudt2)
END DO
!$OMP END PARALLEL
```

Boucle de convergence du gradient conjugué (1/2)

```
REAL(8) :: alpha = 0.0_8   ← variables locales
REAL(8) :: beta  = 0.0_8      initialisées, donc partagées
INTEGER :: it      ← variable locale, donc privée
LOGICAL :: convergence      ← variable locale, donc privée

CALL pmv (r, u, - rnudt2)
CALL saxpy (r, -one, b, one)    rnorm_k, argument en sortie
CALL prodscal (rnorm_k, r, r)← donc partagé

it = 0       ← variable locale, donc privée
convergence = ( SQRT(rnorm_k) < prec )
     ↑ variable locale, donc privée

DO WHILE ( (.NOT. convergence) .AND. (it < it_max) )
      it = it + 1
      CALL pmv (q, p, - rnudt2)
      CALL prodscal (alpha_k, q, p)
```

Boucle de convergence du gradient conjugué (2/2)

```
!$OMP MASTER
      alpha_k = rnorm_k / alpha_k   ← variables locales
      beta_k = rnorm_k              initialisées, donc partagées
      rnorm_k = 0.0_rp   ← argument en sortie, donc partagé
!$OMP END MASTER
!$OMP BARRIER

      CALL saxpy (u, one, p,   alpha_k)
      CALL saxpy (r, one, q, - alpha_k)
      CALL prodscal (rnorm_k, r, r)
!$OMP MASTER
      alpha_k = 0.0_rp     ← variables locales initialisées
      beta_k = rnorm_k / beta_k ← donc partagées
!$OMP END MASTER
!$OMP BARRIER

      CALL saxpy (p, beta_k, r, one)
      convergence = ( SQRT(rnorm_k) < prec )
END DO
```

Boucles parallélisées : pmv

```
c1 = scal / hx**2                              ← variables
c2 = scal / hy**2                              ← locales,
c3 = scal / hz**2                              ← donc
c4 = one - 2.0_rp * ( c1 + c2 + c3 )           ← privées
!$OMP DO
DO k = 1, ntz
   DO j = 1, nty
      DO i = 1, ntx
         a(i,j,k) = c4 * b(i,j,k)
                  + c1 * ( b(i+1,j  ,k  ) + b(i-1,j  ,k  ) )
                  + c2 * ( b(i  ,j+1,k  ) + b(i  ,j-1,k  ) )
                  + c3 * ( b(i  ,j  ,k+1) + b(i  ,j  ,k-1) )
      END DO
   END DO
END DO
!$OMP END DO
```

Boucles parallélisées : prodscal et saxpy

```
!$OMP DO REDUCTION(+:rnorm)
DO k = 1, ntz
   DO j = 1, nty
      DO i = 1, ntx
         rnorm = rnorm + a(i,j,k) * b(i,j,k)
      END DO        ↑ argument en sortie, donc partagé,
   END DO             donc réduction
END DO
!$OMP END DO
```

```
!$OMP DO
DO k = 1, ntz
   DO j = 1, nty
      DO i = 1, ntx
         a(i,j,k) = scala * a(i,j,k) + scalb * b(i,j,k)
      END DO        ↑ arguments en entrée,↑,
   END DO             donc partagés
END DO
!$OMP END DO
```

Parallélisation à deux niveaux

- Il s'agit d'utiliser MPI et OpenMP simultanément, de manière complémentaire
- Parallélisation par MPI sur laquelle on applique une parallélisation par OpenMP : chaque processus MPI est alors une thread maîtresse OpenMP qui crée son équipe de threads
- Dans la pratique, on met un ou deux processus MPI par nœud de calcul ; on remplit le reste du nœud avec des tâches OpenMP
- On limite les communications MPI, on exploite mieux la mémoire partagée
- Dans tous les cas, les performances du matériel sous-jacent influent fortement sur les performances (notamment la bande passante mémoire)

Comparaison MPI, OpenMP, MPI-OpenMP

Dimensions du domaine : ntx = nty = ntz = 512
Volume mémoire total : 8.5 Go
Temps d'exécution pour 20 pas de temps

Nombre de tâches OpenMP ou processus MPI	1	2	4	6	8
OpenMP	1513	1085	628	665	630
MPI	///	1001	555	783	555

		Nombre de tâches OpenMP		
		2	3	4
Nombre de processus MPI	2	577	636	632
	4	561	334	290
	8	282	///	///