Méthodologie M2

Programmation parallèle pour le calcul scientifique

Benoît Semelin

2007

Introduction

Généralités

Bibliographie gratuite

Parallélisme en général:

- http://aramis.obspm.fr/~semelin/enseignement.html : ce cours
- http://www-unix.mcs.anl.gov/dbpp/index.html : livre en ligne sur le parallélisme

OpenMP:

- http://www.openmp.org/specs/ : spécifications officielles
- http://www.idris.fr/data/cours/parallel/openmp/OpenMP_cours.html : Cours de l'IDRIS

MPI:

- http://www.idris.fr/data/cours/parallel/mpi/mpi1_cours_couleurs.pdf : Cours de l'IDRIS
- http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html : Manuel de référence (html)
- http://www.idris.fr/data/cours/parallel/mpi/mpi1_aide_memoire_F90.html
- http://www.idris.fr/data/cours/parallel/mpi/mpi1_aide_memoire_C.html
- http://www-unix.mcs.anl.gov/mpi/tutorial/mpiexmpl/contents.html (Exemples en C)

Le calcul parallèle: qu'est ce que c'est?

« Faire coopérer plusieurs processeurs pour réaliser un calcul »

Avantages:

- **Rapidité:**

  Pour N processeurs, temps de calcul divisé par N, en théorie.

- **Taille mémoire:**

  Pour N processeurs, on dispose de N fois plus de mémoire (en général)

Difficultés:

- Il faut gérer le **partage des tâches.**
- Il faut gérer l'**échange d'information.** (tâches non-indépendantes)

Qu'est ce qui n'est pas du calcul parallèle.

Quand les processeurs ne coopèrent pas:

- Calculs monoprocesseurs séquentiels
- Calculs multiprocesseurs:
  - Exécution pour une série de conditions initiales différentes.
  - Problème divisible en sous-problèmes indépendants:

    Exemple: Mouvement de N particules test dans un champ extérieur.

Une architecture parallèle efficace coûte cher, il faut l'utiliser à bon escient.

Modèles de parallélisme

Architecture matérielle:

SISD	SIMD	MIMD
Single Instruction **S**ingle **D**ata	Single **I**nstruction Multiple **D**ata	Multiple **I**nstruction Multiple **D**ata
PC monoprocesseur	Architecture **vectorielle** monoprocesseur	Architecture **parallèle** multiprocesseur

Modèle de programmation:

Le plus utilisé →

SPMD

Single **P**rogram Multiple **D**ata

Un seul programme (n° processeur = variable)

MPMD

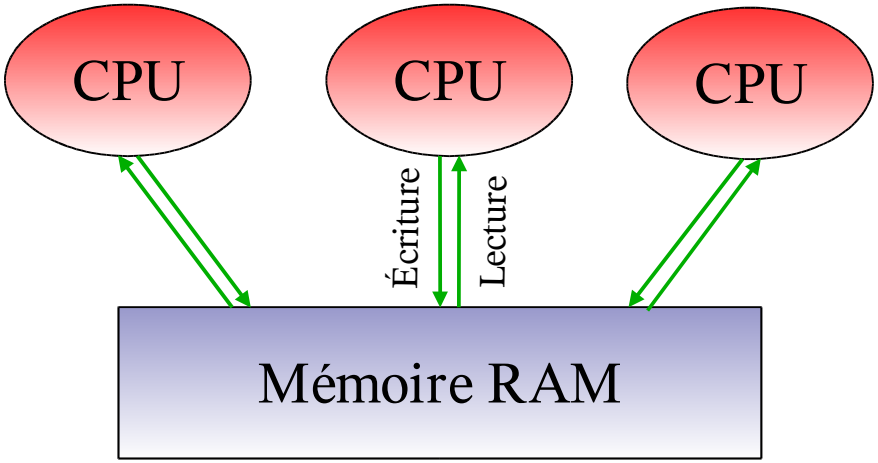
Multiple **P**rogram Multiple **D**ata

Création dynamique de process
Modèle maître-esclave.

Outils de parallélisation:

OpenMP	MPI
ordinateur à **mémoire partagée**	ordinateur à **mémoire distribuée**

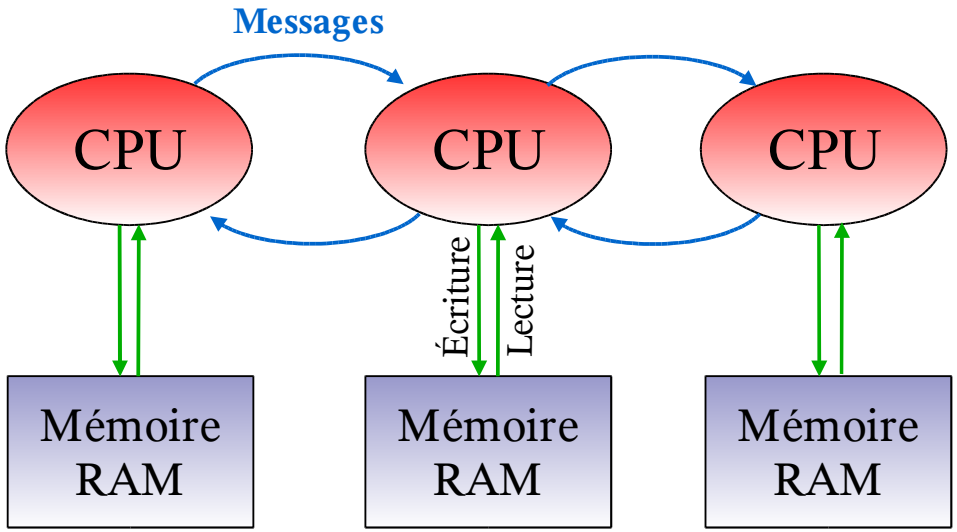
Mémoire partagée / distribuée



Mémoire partagée (SMP)

Tous les processeurs ont accès à l'ensemble de la mémoire.

- Attention aux conflits.
- Très peu de surcoût de parallélisation.
- Le plus souvent nb proc < 32.
- Architecture coûteuse.



Mémoire distribuée

Chaque processeur possède sa propre mémoire. Il n'a pas accès à celle des autres.

- Il faut gérer l'échange de messages (surcoût)
- Architecture bon-marché.
- Il faut ajouter un réseau de com performant.
- Nb de proc ~ illimité.

Revue d'effectif 1: dans le monde

Tous les ans, une liste des 500 plus gros ordinateur est publiée sur:

http://www.top500.org

Extrait de la liste de Novembre 2006:

Rang	Nom	Architecture	Pays/Année	Nb processeur	Puissance de calcul
1	DOE/NNSA/LLNL	BlueGene IBM	USA / 2005	131072	280.6 Tflops
2	NNSA/ Sandia nationnal lab	Cray Red Storm Opteron	USA/2006	26544	101.4 Tflops
3	Watson Research center	BlueGene IBM	USA / 2005	40960	91.2 Tflops
4	DOE/NNSA/LLNL	ASC Purple IBM	USA/2005	12208	75.7 Tflops
5	Barcelonna	Marenostrum IBM	Espagne/2006	10240	62.6 Tflops
6	NNSA Sandia national Lab	Thunderbird Dell	USA/2006	9024	53 Tflops
7	CEA	Tera 10 Bull	France/2006	9968	52.8 Tflops
8	Nasa	Columbia SGI	USA / 2004	10160	51.8 Tflops
9	GSIC Center Tokyo intitute of Tech	TSUBAME Grid Opteron	Japon /2006	11088	47.3 Tflops
10	Oak ridge National Lab	Cray XT3	USA/2006	10424	43.4 Tflops
14	Earth Simulator	NEC	Japon / 2002	5120	35.8 Tflops
...					
+∞	PC Maison		Partout / 2005	1	~ 1-2 Gflops

Revue d'effectif 2: l'astrophysique en France

Voici les moyens de calcul parallèle disponibles pour la recherche en France (non-exhaustif):

Nom	Type d'architecture	Nb processeur	Condition d'accés
CEA DAM	Nova 10 Noeuds de 8 bicoeurs	9968	Secret Défence
IDRIS	IBM Power4 Noeuds de 32 (SMP)	1024	Sur projet détaillé
CINES	IBM Power4 Noeuds de 32 (SMP) IBM Power3 Noeuds de 16 (SMP) Nouvelle machine en 2006 ????	288 464 4000 ?	Sur projet détaillé
MPOPM	HP Marvel EV7 SMP	16	Sur projet pour les membres de l'observatoire
Machines dédiées MAGIC,HORIZON...	-	-	Etre membre du projet.

Outils de parallélisation

La parallélisation peut-être effectuées à divers niveaux:

- *Langages:*
  - Compositional C++ (CC++)
  - Fortran M
  - High performance Fortran (HPF)
  - etc...
- *Bibliothèques:*
  - **Message Passing Interface (MPI)**
  - Parallel Virtual Machine (PVM): ~ obsolète.
  - Pthreads (langage C)
- *Directives de compilation:*
  - **OpenMP**
  - Instructions spécifiques constructeur.
- *Compilateurs:* efficacité très faible.
  - Intel Fortran/C compiler: gratuit.

OpenMP

OpenMP:

modèle de programmation

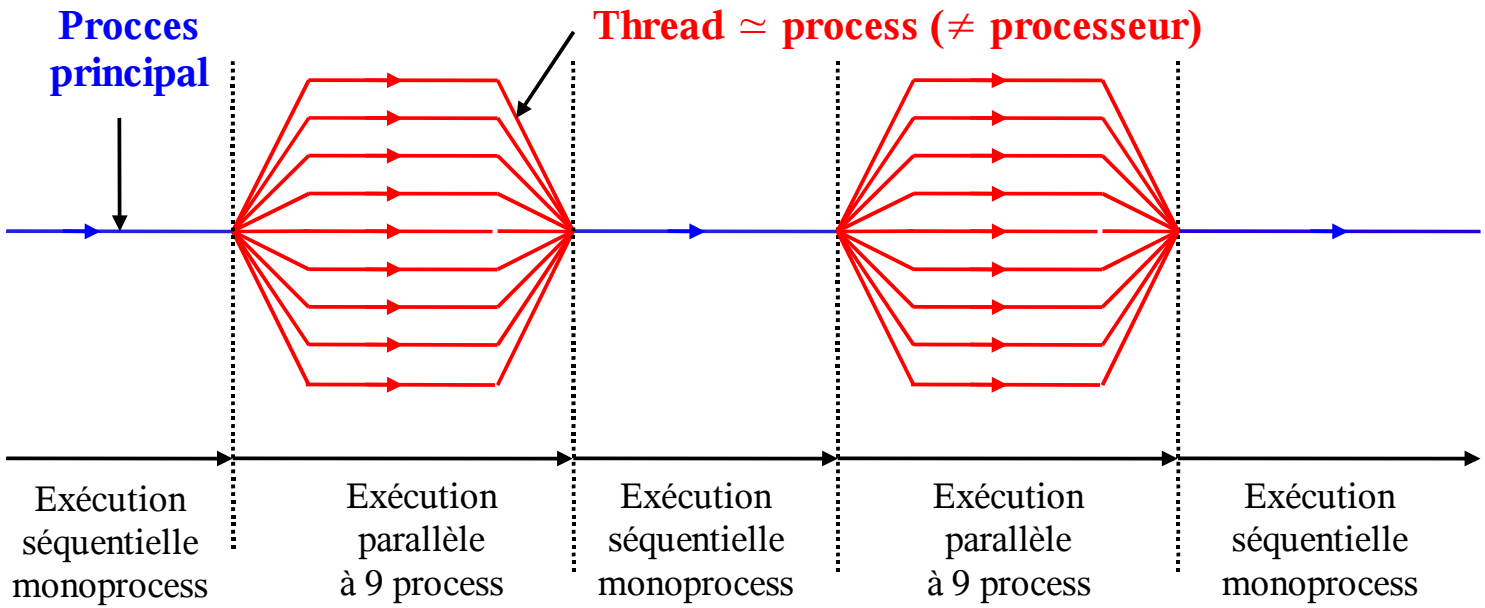
La parallélisation facile: OpenMP

OpenMP est un ensemble de directives de compilation pour paralléliser un code sur une architecture SMP (interfaces Fortran, C et C++)

Le compilateur interprète les directive OpenMP (si il en est capable!)

Les standards d'OpenMP datent de 1997, ceux d'OpenMP-2 de 2000. Les développeurs de compilateurs les implémentent.

Modèles d'exécution OpenMP:



Comment « *faire tourner* » un code OpenMP

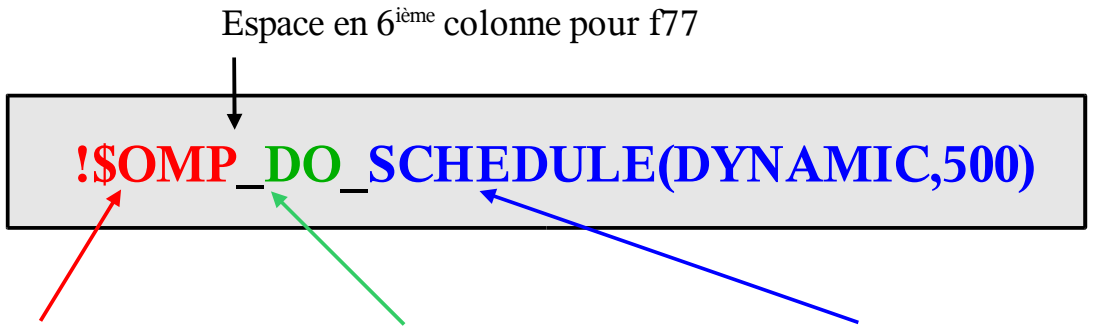
Voici quelques points de repère:

- Écrire le code séquentiel (tout ou partie)
- Débuguer le mieux possible
- Se placer sur une machine SMP multiprocesseur
- Insérer les directives OpenMP
- Compiler avec l'option appropriée: **ifort -openmp toto.f90 -o toto.out**
- Définir le nombre de CPU: **export OMP_NUM_THREADS =4**
- Lancer normalement: **./toto.out**
- Débuguer...
- Évaluer les performances, en particulier le « speedup »:

  Temps exécution N procs / Temps exécution 1 proc

Syntaxe d'une directive OpenMP

Voici un exemple de directive pour f90:



Sentinelle:
Doit être le premier caractère non-blanc de la ligne. « !$ » peut servir de sentinelle de compilation conditionnelle pour une ligne de code normale.

Directive:
Une directive par ligne.

Clause:
Optionnelle. Modifie le comportement de la directive. Il peut y en avoir plusieurs, elle peut être répétée, l'ordre est indifférent

Syntaxe en C/C++: **#pragma_omp_for_schedule(dynamic,500)**

OpenMP:

régions parallèles, comportements des variables

Définir une région parallèle

Comment amorcer une zone d'exécution multiprocesseur:

- Début de zone:

  !$OMP PARALLEL

  #pragma omp parallel {code}

- Fin de zone:

  !$OMP END PARALLEL

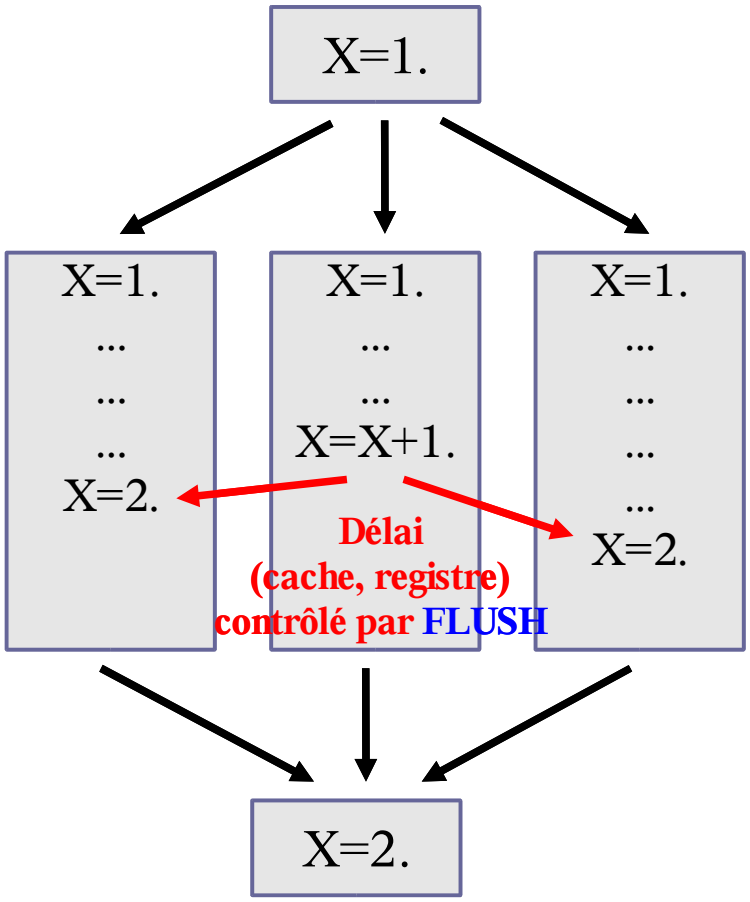
Dans la zone correspondante, N *threads* sont exécutés en parallèle. N est fixé par la variable d'environnement OMP_NUM_THREADS.

Que devient la valeur d'une variable à l'entrée et à la sortie d'une zone parallèle?? Que vaut-elle sur 2 processeurs différents ?

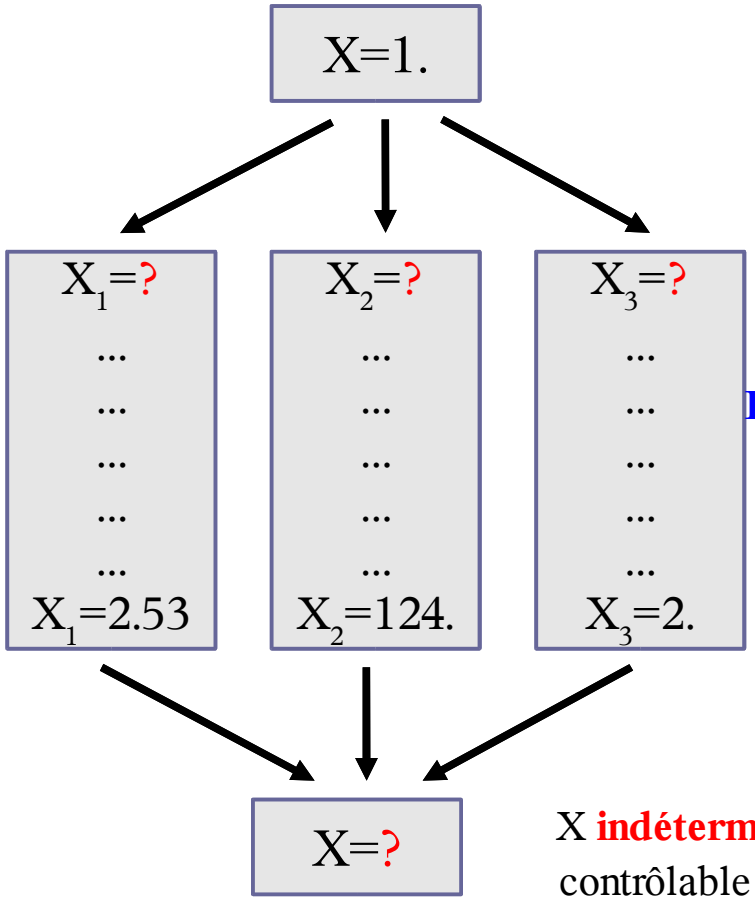
```
SUBROUTINE compute_grad(field,grad,n,size,TYPEMIN)
USE VARIABLES
INTEGER, INTENT(IN) :: n
REAL(KIND=8), INTENT(IN) :: size
REAL(KIND=8), DIMENSION(n), INTENT(IN) :: field
REAL(KIND=8), DIMENSION(n), INTENT(OUT) :: grad
INTEGER, INTENT(IN) :: typemin
REAL(KIND=8) :: val,g1,g2,fx
INTEGER :: i,ip,im

!$OMP PARALLEL
! Autre directive OMP
do i=1,NCELLNOW
  ip=i+1
  im=i-1
  if(ip == NCELLNOW+1) ip=1
  if(im == 0) im=ncellnow
  if(celltype(i) >= typemin) then
    g1=(field(i)-field(im))/(cellsize(i)+cellsize(im))*2.
    g2=(field(ip)-field(i))/(cellsize(i)+cellsize(ip))*2.
    fx=(cellsize(im)+cellsize(i))/(cellsize(im)+2.*cellsize(i)+cellsize(ip))
    grad(i)=(1.-fx)*g1+fx*g2
  endif
enddo
!$OMP END PARALLEL

END SUBROUTINE compute_grad
```

Comportements possibles d'une variable

Variable **SHARED**



Variable **PRIVATE**



X_i **indéterminé**.
contrôlable par
FIRSTPRIVATE

X **indéterminé**.
contrôlable par
LASTPRIVATE

Comportement SHARED/PRIVATE d'une variable

SHARED et PRIVATE sont deux clauses qui spécifient le comportement des variables dans une zone parallèle:

- **SHARED**: la variable a la même valeur sur tout les processeurs. C'est le **statut par défaut**. Attention à la synchro si on la modifie sur un processeur (directive FLUSH).
- **PRIVATE**: la variable a une valeur différente sur chaque processeur (espace mémoire correspondant dupliqué) et indéterminée en entrée de zone. C'est le statut de toute variable déclarée à l'intérieur d'une zone parallèle.

```
SUBROUTINE compute_grad(field,grad,n,cellsize,TYPEMIN)
USE VARIABLES

INTEGER, INTENT(IN) :: n
REAL(KIND=8), INTENT(IN) :: cellsize
REAL(KIND=8), DIMENSION(n), INTENT(IN) :: field
REAL(KIND=8), DIMENSION(n), INTENT(OUT) :: grad
INTEGER, INTENT(IN) :: typemin
REAL(KIND=8) :: val,g1,g2,fx
INTEGER :: i,ip,im

!$OMP PARALLEL PRIVATE(ip,im,g1,g2,fx)
! Autre directive OMP
do i=1,NCELLNOW
  ip=i+1
  im=i-1
  if(ip == NCELLNOW+1) ip=1
  if(im == 0) im=ncellnow
  if(celltype(i) >= typemin) then
    g1=(field(i)-field(im))/(cellsize(i)+cellsize(im))*2.
    g2=(field(ip)-field(i))/(cellsize(i)+cellsize(ip))*2.
    fx=(cellsize(im)+cellsize(i))/(cellsize(im)+2.*cellsize(i)+cellsize(ip))
    grad(i)=(1.-fx)*g1+fx*g2
  endif
enddo
!$OMP END PARALLEL

END SUBROUTINE compute_grad
```

Autres clauses de comportements de variables

On peut changer le comportement défaut:

!$OMP PARALLEL DEFAULT(PRIVATE | SHARED | NONE)

Si on utilise NONE, on doit préciser le comportement de toutes les variables qui apparaissent dans la zone parallèle (exceptions voir références OMP)

Clause supplémentaires:

- **FIRSPRIVATE(X):** X est PRIVATE et initialisé à sa valeur juste avant la zone parallèle.
- **REDUCTION(*,X)**: X est PRIVATE et un produit des valeurs de X sur les différents *threads* est effectué en fin de zone parallèle et stocké dans X. L'opérateur peut-être + , - , * , .OR. , .AND. , MAX, MIN, etc... Remplace les statuts PRIVATE ou SHARED.
- **LASTPRIVATE(X)**: X est PRIVATE. La valeur de X du *threads* exécutant la dernière mise a jour de X (sémantiquement) est conservée en fin de zone parallèle.
- **COPYPRIVATE(X)**: Pour diffuser une variable privée (Directive SINGLE uniquement).

Variable PRIVATE mais globale... dans le zone paralèlle

- Une variable PRIVATE est locale.
- Il est intéressant de pouvoir appeler une ou plusieurs procédure/fonction dans une zone parallèle.
- Certaines procédures/fonctions utilisent et modifient des variables globales.

!$OMP THREADPRIVATE(X,/COORD/) **(Dans le .h)**

La variable X et le bloc commun /COORD/ seront PRIVATE mais globaux dans chaque thead des régions parallèles. Certains compilateurs Fortran n'acceptent que des blocs commun en argument de THREADPRIVATE.

!$OMP PARALLEL COPYIN(/COORD/) **(Dans le .f90)**

Les valeurs de /COORD/ sont copiées dans les répliques privée de chaque thread en entrée dans les régions parallèles.

OpenMP:

partage du travail entre *threads*

Partage du travail: distribuer une boucle

La directive **DO** se place juste avant le début d'une boucle, elle **répartit les itérations** entre les processeurs.

- **Pas de DO WHILE** !
- Le mode de répartition dépend de la clause optionnelle **SCHEDULE**
- Le mode de répartition par défaut dépend de l'implémentation d'OpenMP.

Les itérations doivent être **indépendantes** (ordre indifférent).

A la fin de la boucle on insère la directive **END DO**. Les threads se synchronisent (on attend la dernière).

```
SUBROUTINE compute_grad(field,grad,n,cellsize,TYPEMIN)
USE VARIABLES

INTEGER, INTENT(IN) :: n
REAL(KIND=8), INTENT(IN) :: cellsize
REAL(KIND=8), DIMENSION(n), INTENT(IN) :: field
REAL(KIND=8), DIMENSION(n), INTENT(OUT) :: grad
INTEGER, INTENT(IN) :: typemin
REAL(KIND=8) :: val,g1,g2,fx
INTEGER :: i,ip,im

!$OMP PARALLEL PRIVATE(ip,im,g1,g2,fx)
!$OMP DO SCHEDULE(DYNAMIC,20)
do i=1,NCELLNOW
  ip=i+1
  im=i-1
  if(ip == NCELLNOW+1) ip=1
  if(im == 0) im=ncellnow
  if(celltype(i) >= typemin) then
    g1=(field(i)-field(im))/(cellsize(i)+cellsize(im))*2.
    g2=(field(ip)-field(i))/(cellsize(i)+cellsize(ip))*2.
    fx=(cellsize(im)+cellsize(i))/(cellsize(im)+2.*cellsize(i)+cellsize(ip))
    grad(i)=(1.-fx)*g1+fx*g2
  endif
enddo
!$OMP END DO
!$OMP END PARALLEL

END SUBROUTINE compute_grad
```

Stratégies de répartition des itérations

La clause **SCHEDULE** permet de contrôler la stratégie de répartition des itérations d'une boucle entre les processeurs, elle admet 2 arguments: **SCHEDULE(stratégie,N).**

Stratégies possibles:

- **STATIC**: Chaque thread reçoit à tour de rôle **N** itérations à traiter. La distribution s'effectuent dans un ordre fixé, éventuellement en plusieurs tours. N est optionnel, si il n'est pas spécifié **N**~ nb itération / nb threads.
- **DYNAMIC**: Chaque thread reçoit N itérations à traiter. Dès qu'un thread a fini, il en reçoit N autres, jusqu'à épuisement du travail. **N** est optionnel, par défaut **N**=1.
- **GUIDED**: Les itérations sont divisées en paquets de taille exponentiellement décroissante. Les paquets sont distribués dynamiquement. La taille du plus petit paquet est **N**.
- **RUNTIME**: Le choix de la stratégie est reporté au moment de l'exécution Il sera alors déterminé par le contenu de la variable d'environnement OMP_SCHEDULE.

DYNAMIC et GUIDED assurent un meilleur équilibrage de charge que STATIC.

Un exemple: calcul de potentiel périodique

Programme pour calculer une interaction gravitationnelle avec des conditions de bord périodiques.

Remarques:

- Il existe un algorithme plus efficace (décomposition de la somme en deux parties).
- On pourrait paralléliser la première boucle!
- Remarquer l'utilisation de la clause REDUCTION.
- Zones parallèles longues => bon speedup

```
PROGRAM POTPER

IMPLICIT NONE

REAL(KIND=8),PARAMETER :: BOXSIZE=1.
INTEGER, PARAMETER :: NREP=200
INTEGER, PARAMETER :: NDEF=40
INTEGER :: i,j,k,l
REAL(KIND=8) :: x1,y1,z1,x2,y2,z2,pot,dist

y1=0.
z1=0.
do i=1,ndef
  x1=dble(i)/ndef*BOXSIZE
  pot=0.
!$OMP PARALLEL PRIVATE(k,l,x2,y2,z2,dist) REDUCTION(+:pot)
!$OMP DO SCHEDULE(DYNAMIC)
  do j=-nrep,nrep
    x2=dble(j)*BOXSIZE
    do k=-nrep,nrep
      y2=dble(k)*BOXSIZE
      do l=-nrep,nrep
        z2=dble(l)*BOXSIZE
        dist=sqrt((x2-x1)**2+(y2-y1)**2+(z2-z1)**2)
        if(dist < BOXSIZE*NREP*0.95) then
          pot=pot+(x2-x1)/(sqrt((x2-x1)**2+(y2-y1)**2+(z2-z1)**2))**3
        endif
      enddo
    enddo
  enddo
!$OMP END DO
!$OMP END PARALLEL
  print*,x1,pot
enddo

END PROGRAM POTPER
```

Partage du travail en l'absence de boucle

- La directive **SECTIONS** amorce une zone où le codes est découpé en morceaux. **END SECTION** clôt cette zone
- Ces morceaux sont séparés par des directives **SECTION**.
- Chaque morceau (section) sera exécuté une fois unique par un des thread.

- **L'ordre d'exécution** des sections doit être indifférent**!**

- **SECTIONS** admet **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE,** et **REDUCTION** comme clauses.

```
...

!$OMP PARALLEL
!$OMP SECTIONS

!$OMP SECTION
 CALL UPDATE_XPOS()

!$OMP SECTION
 CALL UPDATE_YPOS()

!$OMP SECTION
CALL UPDATE_ZPOS()

!$OMP END SECTIONS
!$OMP END PARALLEL

...
```

Partage du travail avec WORKSHARE

WORKSHARE, une directive trop générale/ambitieuse?

D'après les spécifications d'OpenMP WORKSHARE doit diviser le travail de telle manière que chaque instruction de la zone soit exécutée exactement 1 fois dans un thread, en respectant la sémantique du code...Si les instructions ne sont pas indépendantes, pas de speedup.

Aux fabricants de compilateurs de l'implémenter.

En pratique, utilisable en f90 pour:

- Opérations sur les tableaux.
- Instruction WHERE
- Instruction FORALL

WORKSHARE n'admet **AUCUNE** clause.

```
!$OMP PARALLEL
!$OMP WORKSHARE

X(1:N) = sin(THETA(1:N))
Y(1:N) = cos(THETA(1:N))
WHERE (Y .ne. 0) P = X / Y

FORALL( i = 1 : N , j = 1 : N , i /= j )
 pot(i)=1./(x(i)-x(j))
END FORALL

!$OMP END WORKSHARE
!$OMP END PARALLEL
```

Exécution exclusive: directives SINGLE et MASTER

Si, dans une zone parallèle, on souhaite qu'une partie du code soit exécutée par un seul thread, on a le choix entre 2 directives:

- **!$OMP SINGLE / !$OMP END SINGLE :**

  Le premier thread qui aborde la zone l'exécute. Les autres sautent à la fin et attendent que celui qui exécute ait fini. **SINGLE** admet les clauses **PRIVATE** et **FIRSTPRIVATE**. **END SINGLE** admet **NOWAIT** et **COPYPRIVATE**.

- **!$OMP MASTER / !$OMP END MASTER :**

  Le thread « master » (numéro 1) exécute la zone. Les autres sautent la zone, et **n'attendent pas! MASTER** n'admet aucune clause.

Ces directives peuvent par exemple servir à faire des entrées/sorties.

OpenMP:

synchronisation des *threads*

Problèmes de synchronisation.

Dans une zone parallèle, l'utilisation de variables SHARED peut créer des problèmes de synchronisation. Exemples:

- 3 threads calculent X(1),X(2) et X(3), puis chacun utilise les 3 valeurs. Il faut s'assurer que les autres ont finit leur calcul.
- N threads incrémentent la **même** variable. Il faut s'assurer qu'ils ne tentent pas d'écrire en même temps dans l'emplacement mémoire de la variable.
- Problème de vidage de registre et de tampon.

Synchronisation simple: BARRIER

La directive BARRIER synchronise les threads: tous s'arrête au niveau de la directive jusqu'à ce que le dernier soit arrivé. Puis il continuent tous. Syntaxe:

!$OMP BARRIER

N'admet aucune clause.

Une **BARRIER** est **implicitement incluse** dans les directives suivantes:

- END PARALLEL
- END DO (sauf clause NOWAIT)
- END SECTIONS (sauf clause NOWAIT)
- END SINGLE (sauf clause NOWAIT)
- END WORKSHARE (sauf clause NOWAIT)

Pas de **BARRIER** implicite de END MASTER !

Mettre à jour la mémoire: directive FLUSH

!$OMP FLUSH(x,y)

Lorsqu'un thread rencontre une directive FLUSH, il met à jour les variables spécifiées entre parenthèses, c'est à dire:

- Il vide les tampons d'écriture, les caches, les registres.
- Il vérifie si un autre thread n'a pas modifié la variable.

FLUSH n'a d'utilité que pour les variables visibles des autres threads.
Si aucune variable n'est spécifiée, toutes les variable visibles sont mises à jour.

La directive **FLUSH** est implicitement appelée par les directives suivantes: BARRIER, CRITICAL, END CRITICAL, END DO , END SECTIONS , END SINGLE, END WORKSHARE, ORDERED, END ORDERED,PARALLEL et END PARALLEL.

On a rarement besoin d'utiliser FLUSH directement. Il est plus simple (et plus lent) d'utiliser BARRIER.

Éviter les conflits: directives ATOMIC et CRITICAL

Comment éviter que 2 threads tentent de changer en même temps la même variable.

Directive **ATOMIC**:

La ligne de code qui suit la directive ATOMIC et qui modifie une variable X, est exécutée atomiquement, c'est à dire jamais simultanément pas deux threads. (Voir référence OpenMP pour la forme de la ligne de code concernée).
ATOMIC peut être plus performant que CRITICAL.

```
!$OMP PARALLEL PRIVATE(XLOCAL)
!$OMP DO
do I=1,N
call compute(xlocal(i),i)

!$OMP ATOMIC
xglobal = xglobal + xlocal(i)

enddo
!$OMP END DO
!$OMP END PARALLEL
```

Directive **CRITICAL**:

Même fonction que ATOMIC, mais concerne une zone de code. Un seul thread peu accéder à cette zone simultanément La zone se termine par END CRITICAL. Si on a plusieurs zones CRITICAL, il faut les nommer pour les rendre indépendante.

```
!$OMP PARALLEL PRIVATE(XLOC,YLOC)

call compute_xloc()
call compute_yloc()

!$OMP CRITICAL(ZONE1)
xglob=xglob+xloc
yglob=yglob+yloc
!$OMP END CRITICAL(ZONE1)

!$OMP END PARALLEL
```

Dans l'ordre: directive ORDERED

La directive **ORDERED** permet, à l'intérieur d'une boucle parallélisée, d'exécuter une zone séquentiellement, c'est à dire thread par thread, dans l'ordre des indices croissant.

- Cela permet de faire une entrée-sortie ordonnée dans une zone parallèle.

- C'est un outil de débugage. Cela permet de vérifier l'indépendance des itérations d'une boucle.

- Si la zone ORDERED représente une fraction du temps de calcul d'une itération, supérieure à 1./OMP_NUM_THREAD, cela ralentit l'exécution.

```
!$OMP PARALLEL PRIVATE(XLOCAL)

!$OMP DO ORDERED SCHEDULE(DYNAMIC)
do I=1,N
call compute(xlocal(i),i)
!$OMP ORDERED
write(*,*) i, xlocal(i)
!$OMP END ORDERED
enddo
!$OMP END DO

!$OMP END PARALLEL
```

OpenMP:

variables d'environnement et bibliothèque standard

Variables d'environnement

Il existe 4 variables définissent l'environnement OpenMP:

- **OMP_DYNAMIC**: booléen. Permet l'ajustement dynamique du nombre de threads. Si TRUE, l'utilisateur déclare le nombre maximal de thread, et le système en donne un nombre inférieur ou égal, en fonction des disponibilités.
- **OMP_NUM_THREADS**: entier. Fixe le nombre (maximal) de threads.
- **OMP_NESTED**: booléen. Permet d'imbriqué les régions parallèles les une dans les autres: chaque threads peut se subdiviser. Délicat à utiliser. Inutile pour le calcul scientifique?
- **OMP_SCHEDULE**: chaîne de caractères. Spécifie la stratégie de distribution des itérations de la directive DO associée à la clause RUNTIME.

Exemples:

```
export OMP_DYNAMIC TRUE
export OMP_NUM_THREADS 8        (souvent indispensable)
export OMP_NESTED FALSE
export OMP_SCHEDULE "GUIDED 4 "
```

Les valeurs des variables peuvent être modifiées pendant l'exécution par des fonctions de la bibliothèque standard.

Bibliothèque standard: contrôle de l'environnement

On peut souvent se passer complètement de ces fonctions.

Modèles d'exécution: (appel depuis une zone séquentielle)

- **OMP_SET_DYNAMIC(boolean)**: Subroutine. Active/désactive l'ajustement dynamique du nombre de thread: fixe OMP_DYNAMIC.
- **OMP_GET_DYNAMIC()**: Function. Retourne la valeur actuelle de OMP_DYNAMIC.

- **OMP_SET_NESTED(boolean)**: Subroutine. Fixe OMP_NESTED
- **OMP_GET_NESTED()**: Function. Retourne la valeur actuelle de OMP_NESTED .

Contrôle du nb de thread/processeur:

- **OMP_SET_NUM_THREADS(entier):** Subroutine. Fixe le nombre de thread (maximum) pour les prochaines zones parallèle (valeur de OMP_NUM_THREADS). Appel depuis une zone séquentielle.
- **OMP_GET_NUM_THREADS():** Function. Retourne **le nb réel de threads utilisé à l'instant t.**
- **OMP_GET_MAX_THREADS():** Function. Retourne la valeur de OMP_NUM_THREADS (nb maximal de threads).
- **OMP_GET_NUM_PROCS():** Function. Retourne le **nombre de processeurs** utilisés.

Bibliothèque standard: contrôle *manuel* de la parallélisation

Exécution conditionnelle/ partage des tâches manuel:

- **OMP_IN_PARALLEL()**: Boolean function. Détermine si on est dans une région parallèle. La directive PARALLEL peut comporter une clause de condition (cf spécifications OpenMP). Utile dans ce cas/
- **OMP_GET_THREAD_NUM()**: Fonction entière. Retourne le numéro du thread. Permet, par exemple, de faire le partage du travail "à la main", sans utiliser de directive OpenMP (Masochiste sur une architecture SMP).

Utilisation de verrous:

Un verrou est libre ou possédé par un thread. Une série de fonction OpenMP permet de les manipuler, par exemple d'attendre à un point du code que le verrou soit libéré par un autre thread.

Permet des comportements similaires à la directive CRITICAL, mais avec un contrôle plus fin. Voir spécifications OpenMP.

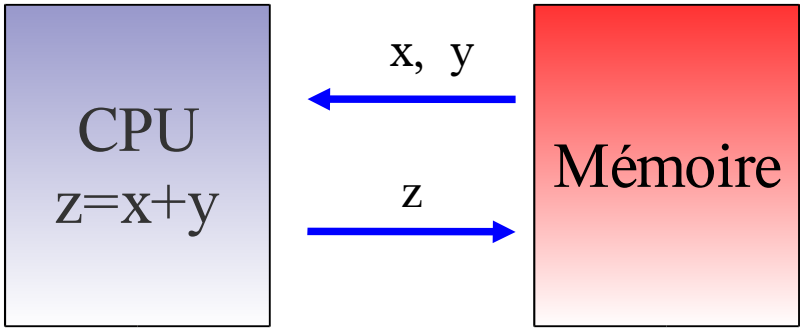
Parallélisation sur architectures à mémoire distribuée:

décomposition en domaines

Décomposition en domaine / fonction

Schéma basique d'un ordinateur:

- Une unité de calcul opère sur des données quelle lit et écrit mémoire.



Si on a *N* CPU et *N* mémoires, il faut:

- Répartir les calculs:
  Chaque unité de calcul va assurer un type de calcul différent sur l'ensemble des données. C'est le principe de **décomposition par fonction**.
- Répartir les données:
  Toutes les unités de calcul effectuent les mêmes calculs sur la partie de données qui leur est attribuée. Il accès aux données des autres unités decalcul se fait par échage de messages. C'est le principede **décomposition en domaines**.

La séparation est arbitraire, en général on combine les deux. Mais, pour le calcul scientifique, c'est la décomposition en domaines qui est cruciale.

Décomposition en domaine: stratégie.

Les contraintes:

- Les communications (échange de données entre domaines) ont un coût en temps CPU: il faut **minimiser la quantité de communication** (nombre et taille).

- Il faut que chaque domaine représente la même quantité de travail. Il faut **minimiser les fluctuations** de charge et **assurer l'équilibrage** dynamique des charges.

Une décomposition idéale:

- **Aucun** besoin de communication entre domaines.
- Domaines statiques: charge constante, équilibré.

Ce n'est **pas du parallélisme**!

Une décomposition satisfaisante:

- Temps de communication << temps de calcul.
- Equilibrage avec variation relative de charge **< 10%.**
- Temps de redéfinition dynamique des domaines << temps de calcul.

Décomposabilité

La possibilité/efficacité d'une décomposition en domaines dépend du problème-algorithme. Exemples:

Pb-algo **non décomposable** (non parallélisable...):

- Intégration d'une ODE par méthode Euler, Runge-Kutta...
- Dynamique un système à trois corps (cf ODE).

Pb-algo à **décomposition délicate** ou inefficace:

- Dynamique d'un système N corps.
- Résolution d'une EPD sur grille adaptative et pas de temps adaptatif (équilibage).
- Réseau de réactions chimiques (vitesses de réaction).

Pb-algo à **décomposition simple** et efficace:

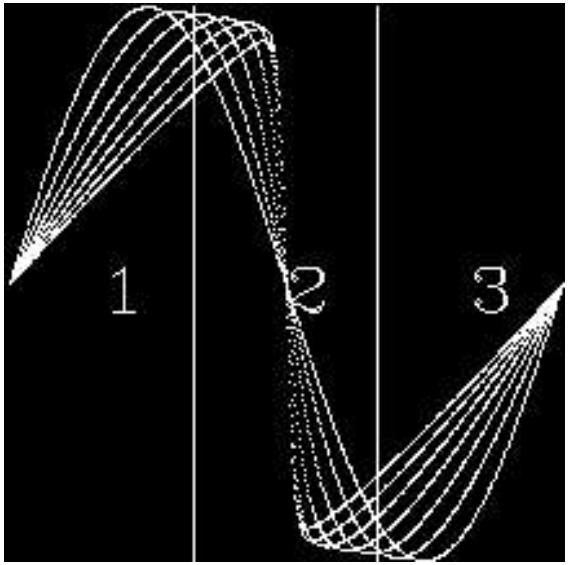
- Résolution d'une EDP sur grille non adaptative.
- Simulation Monte-Carlo pour certains pbs (équilibrage de charge simple)

Exemple de décomposition 2: résolution d'EDP sur grille.

Résolution de l'équation de Burger: $$\frac{\partial v}{\partial t} + v\frac{\partial v}{\partial x} = \frac{1}{Re}\frac{\partial^2 v}{\partial x^2}$$

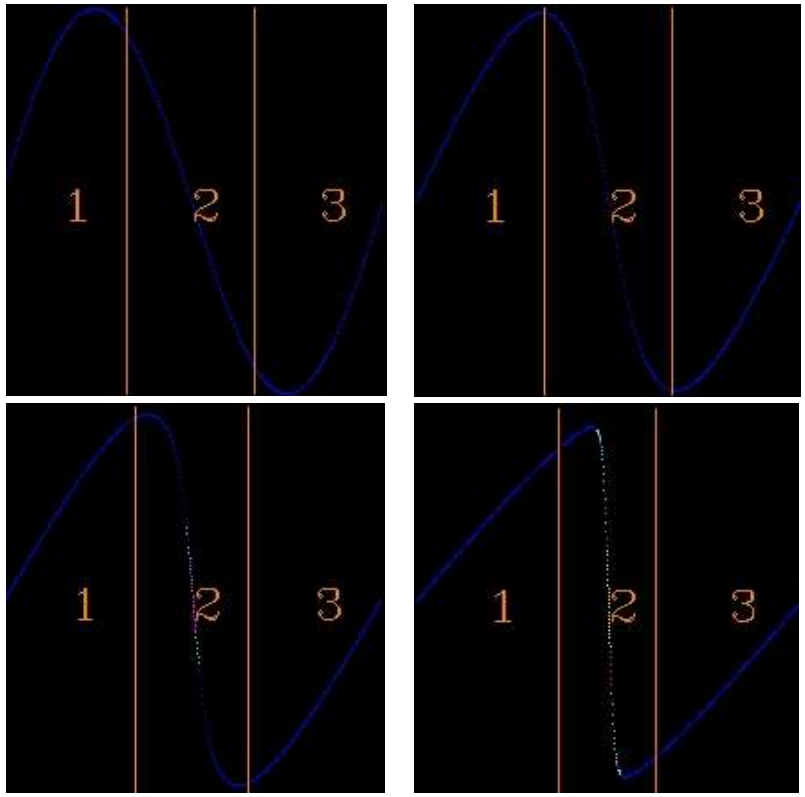
On résout l'équation sur une grille 1-D, par une méthode de différences finies, et avec un schéma explicite (!) à **pas de temps constant**.

Taille de cellule constante: domaines statiques



Grille adaptative : domaines dynamique.

Le domaine 2 se restreint spatialement mais conserve autant de cellule que 1 et 3.



Exemple de décomposition 2: L'algorithme Treecode.

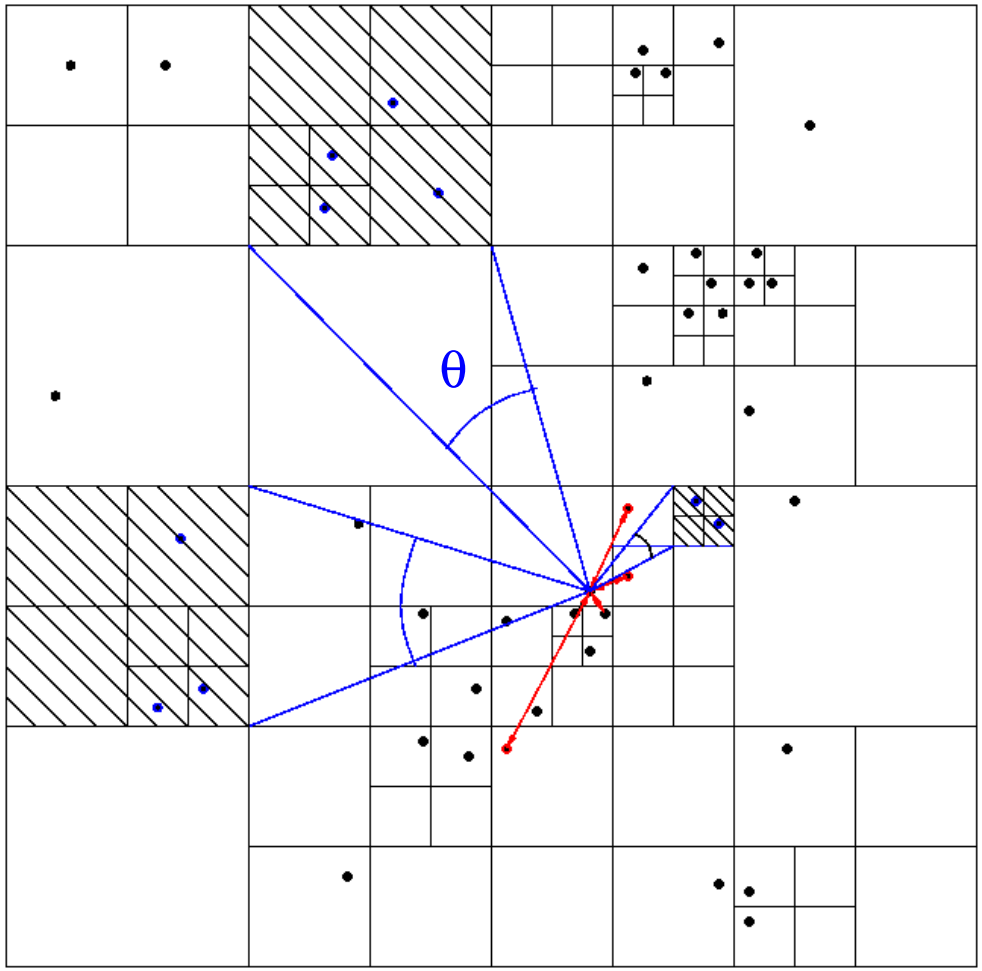
Dynamique d'un système N-corps:

Calcul des forces en N ln(N), par utilisation du développement multipolaire. Contrôle de la précision par θ, l'angle d'ouverture. C'est un algorithme dit **« en arbre ».**

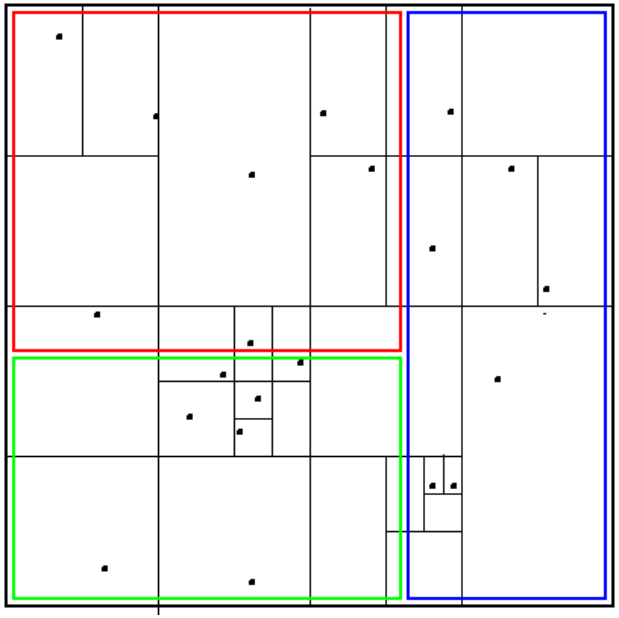
Conséquences pour la parallélisation:

Moins d'interactions entre zones distantes.

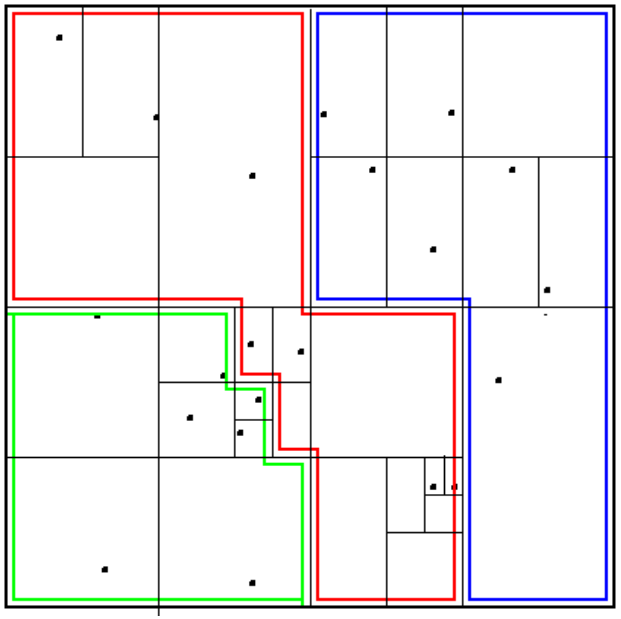
=> Des domaines "compacts" dans l'espace (x,y,z) minimisent les communications.



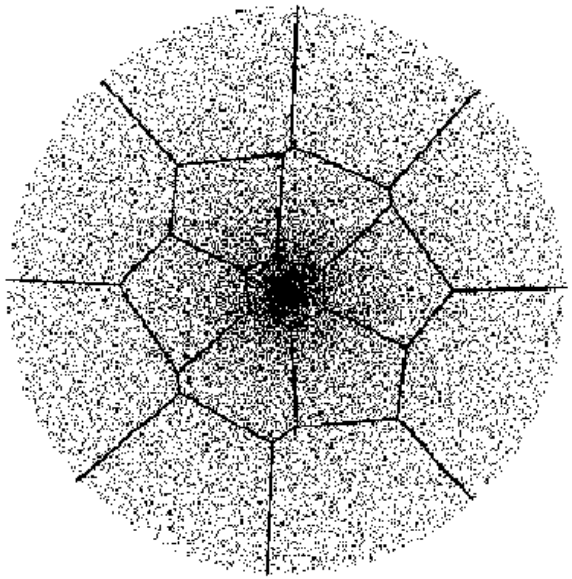
Algorithme Treecode: exemple de décomposition.



Méthode par dichotomie.
Simple et efficace.
Gadget (code publique)



Décomposition de l'arbre.
Pour les algo oct-tree.



Tesselation Voronoï.
Communications mimales.
Construction des domaines coûteuse.

Plusieurs solutions possibles. Ca dépend des détails de l'implémentation de l'algorithme commun.

Parallélisation sur architectures à mémoire distribuée:

communications.

Temps de transmission d'un message.

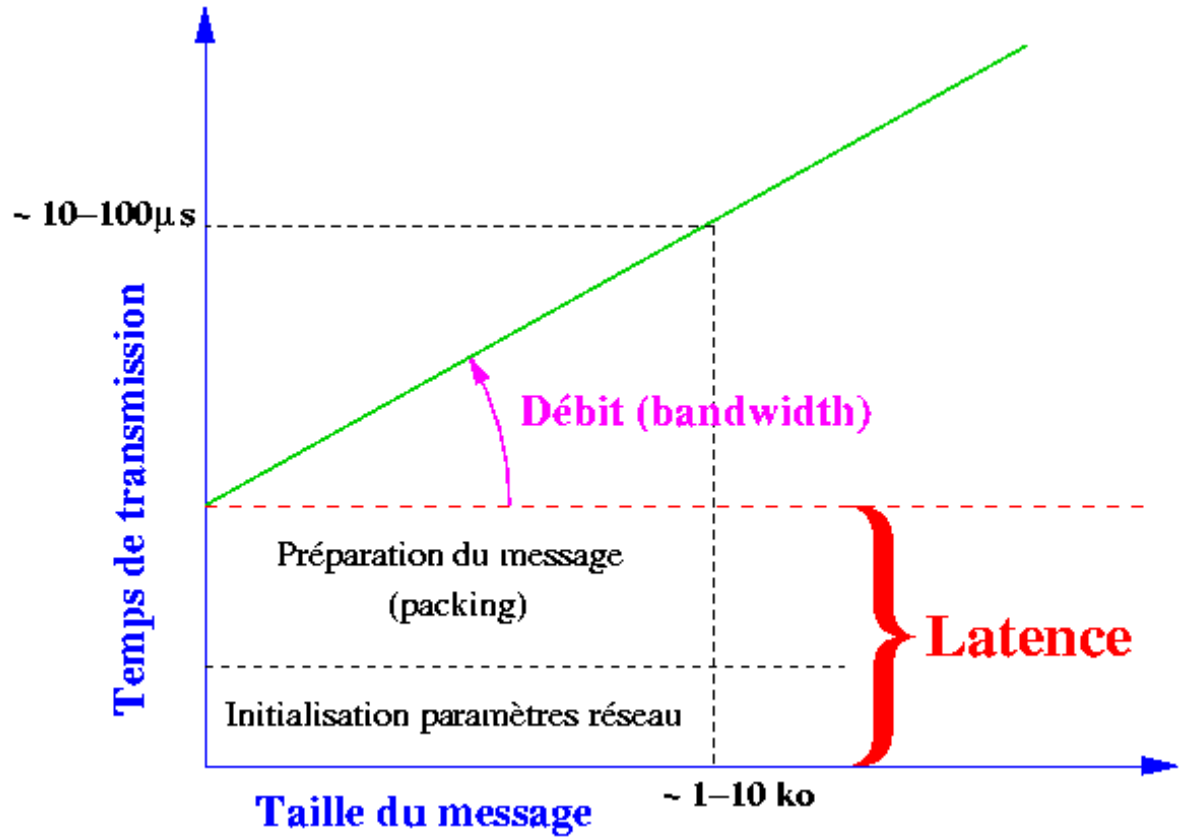
Sur une architecture à mémoire distribuée, les unités de calculs doivent échanger des messages. Ces communications ne sont pas instantanée.

Le coût dépend du réseau de communiations et de l'implémentation MPI.

Détails des contributions au coût de communication. voir graphique.

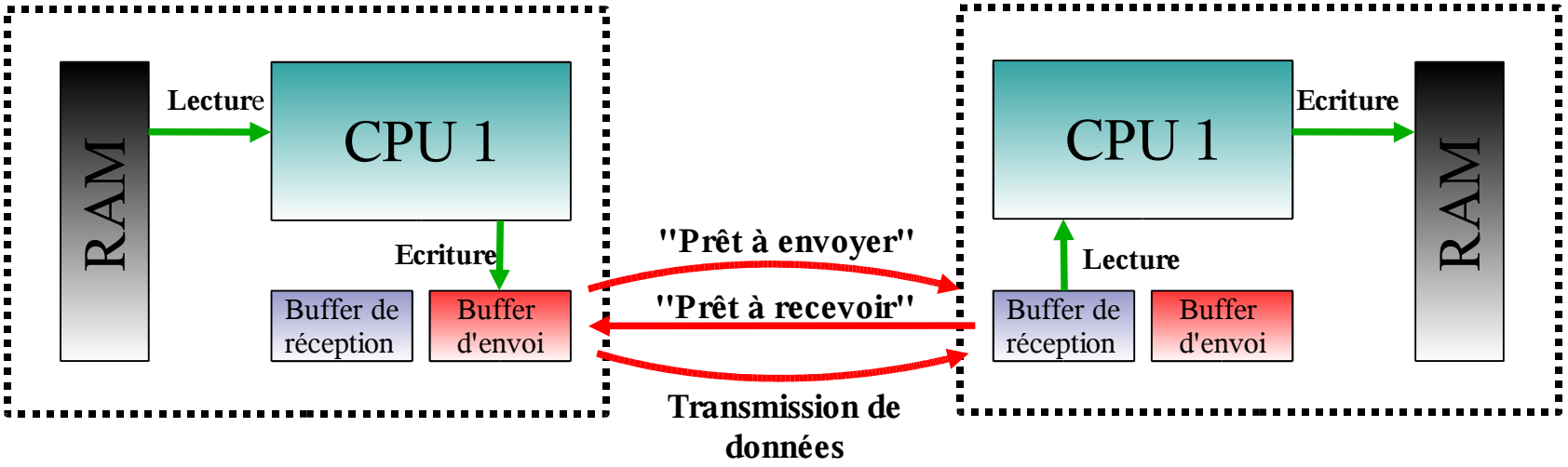
Conséquences:

Il vaut mieux envoyer 1 gros message que plein de petits. Il faut grouper les communication!



Modes de communications entre process.

Exemple de mécanisme de communication: (two-sided, buffered, synchrone)



Mode de commnications: définitions

- **One-sided / Two-sided**:

  Un seul processeur gère tout les éléments de la com / les deux processeurs intervienent, il gère chacun une partie des éléments de la com.

- **Synchrone / Asynchrone**:

  Le processeur qui envoit (1) attend / n'attend pas que le proccesseur qui recoit (2) ait signalé qu'il était prêt. En mode asynchrone, le processeur (1) transmet les données qui sont écrites dans le buffer de réception du processeur (2). Il viendra les lire plus tard, quand le code lui dira de faire.

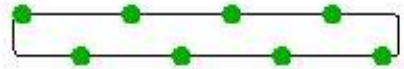
Communication globales: les contraintes matérielles

Dans un ordinateur multi-processeur, chaque processeur n'a pas un lien avec tout les autres pour les communications (trop cher). Il faut pourtant tenter de l'optimiser.

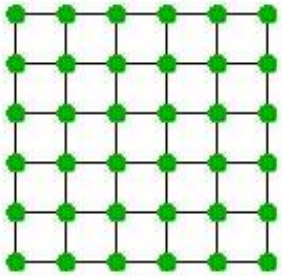
Deux **critères de performance** pour un réseau de n processeurs:

- Nombre de liens moyen pour joindre 2 processeurs n_l
- "Bande passante de bissection" B_b (bande passante entre 2 moitiés du réseau)

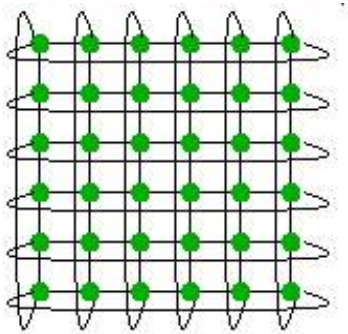
Ligne: $n_L \sim n/3$, $B_b=1$



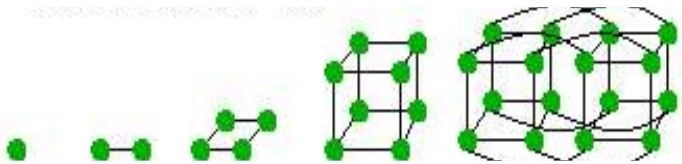
Anneau: $n_L \sim n/4$, $B_b=2$



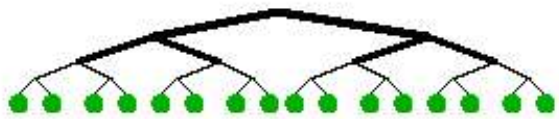
Grille: $n_L \sim ?$ n , $B_b=?$ n



Tore: $n_L \sim ?$ n/2 , $B_b=2?$ n



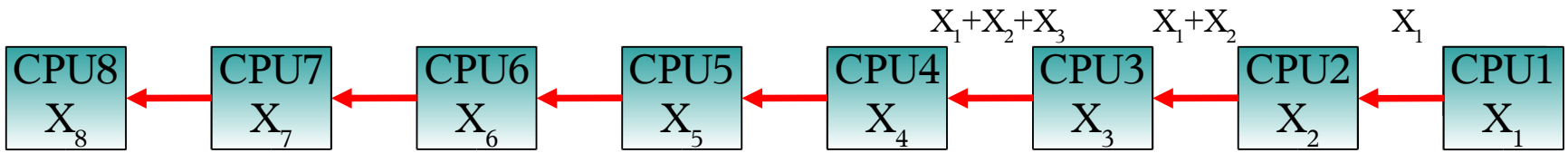
Hypercube dim D: $n_L \sim D$, $B_b=n/2$



Arbre hiérarchique: $n_L \sim \log(n)$, $B_b=n/2...?$

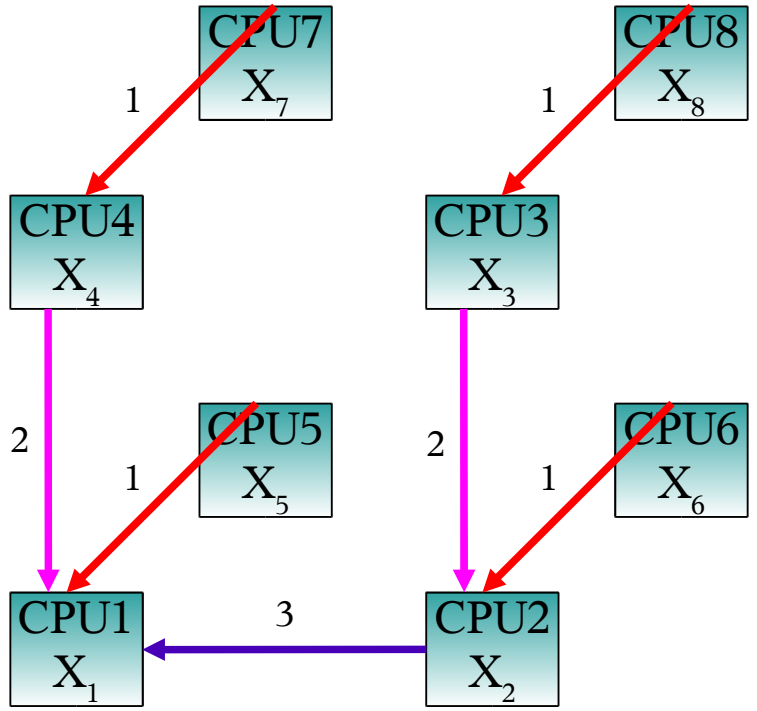
Communications globales sur l'hypercube

Une implémentation naïve de la réduction d'une variable (somme des valeurs sur les N processeurs) nécessite **N communications** successives:



Une implémentation de la réduction en utilisant le principe de l'**hypercube** nécessite **log(N) communications** successives.

Le principe de l'hypercube peut servir pour organiser des communications globales type *REDUCTION* dans un code. MPI s'en sert pour ses fonctions de communications globales génériques. Si ces routines génériques ne répondent pas aux besoins de l 'utilisation, il peut utiliser le principe de l'hypercube lui même.



Performances d'un code parallèle.

Coût des communications:

- Si le temps de communications = 10% du temps de calcul, inutile d'utiliser plus de 5-10 processeurs. Il faut **minimiser le temps de communication**.
- Pistes pour réduire les communications:
  - Recouvrement des bords de domaines (Simulation sur grille)
  - Duplication des données critiques (Treecode)

Equilibrage de charge:

- Si un processeur est 1,3 fois plus lent qu'un autre, on obtient le même temps de calcul qu'avec 0.7 fois moins de processeurs.
- **Equilibrage dynamique des charges** ("overload diffusion"):

- Applicable, par ex, quand on calcule une évolution temporelle.
- Chaque domaine a une taille $S_i(t)$ variable . A chaque pas de temps, on calcule $T_i(t)$, le temps de calcul sur chaque processeur.
- On adapte S_i, par exemple: $S_i(t+dt)=S_i(t)*T_{moy}(t)/ T_i(t)$ (stabilité?)

MPI

Introduction

MPI est une bibliothèque de communication pour le parallélisme sur architectures à mémoire partagée (Fortran, C, C++).

Les standards MPI:

- **MPI 1** (1994): Le coeur de la bibliothèque
  - Communicateurs
  - Communications locales et globale
  - Définition de type de variable
  - Topologies
- **MPI 2** (1997): extensions et support C++ et f90
  - Communications "one-sided"
  - Entrées-sorties parallèles (non traitées ici)

Implémentations gratuites:

- **LAM**: http://www.lam-mpi.org

MPI 1 et 2 (en partie) sur grille de PC ou architectures SMP.

- **MPICH2**: http://www-unix.mcs.anl.gov/mpi/mpich2/index.htm

MPI 1 et 2 (en partie) sur grille de PC ou architectures SMP.

LAM plus facile à installer?

MPI:

fonctions générales, environnement

Environnement système

Il faut inclure la bibliothèque MPI.

- mpif.h en fortran.
- mpi.h en C/C++.

```
program EX_0

implicit none
include "mpif.h"                    !(ou USE MPI avant implicit none)
integer :: nb_procs,rang,err

call MPI_INIT (err)
call MPI_COMM_SIZE ( MPI_COMM_WORLD ,nb_procs,err)
call MPI_COMM_RANK ( MPI_COMM_WORLD ,rang,err)
print *, 'Je suis le processus' ,rang, 'parmi' ,nb_procs
call MPI_FINALIZE (err)

end program EX_0
```

Exemples de commandes de compilation:

- Sur MPOPM:
  - **source /usr/opt/HPMPI200/bin/use_hp_mpi**
  - **mpif90 EX_0.f90** (ou mpif77, mpicc, mpiCC)
  - **mpirun -np 8 EX_0.out**
- Sur zelkova: (ou quadrioptéron SIO, ou grille PC-LAM)
  - **setenv PATH /usr/local/lam-mpi/bin:${PATH}**
  - **mpif77 EX_0.f** (ou mpicc ou mpic++, attention pas de f90!)
  - **lamboot -v**, puis **mpirun -np 8 EX_0.out**, puis **lamwipe -v**.

Initialisation et sortie

Avec MPI, on définit **une seule zone parallèle** dans le programme, souvant l'ensemble du code.

Les parties du code avant et après la zone parallèle sont locale sur chaque processeur. Pas de communications possibles.

```
program EX_0

implicit none
include "mpif.h"                    !(ou USE MPI avant implicit none)
integer :: nb_procs,rang,err

call MPI_INIT (err)
call MPI_COMM_SIZE ( MPI_COMM_WORLD ,nb_procs,err)
call MPI_COMM_RANK ( MPI_COMM_WORLD ,rang,err)
print *, 'Je suis le processus' ,rang, 'parmi' ,nb_procs
call MPI_FINALIZE (err)

end program EX_0
```

Début de zone parallèle

En fortran: **call MPI_INIT(err)**

En C: **MPI_Init();**

En C++: **MPI::Init();**

Fonction collective! (appelée par tous les process)

Fin de zone parallèle

En fortran: **call MPI_FINALIZE(err)**

En C: **MPI_Finalize();**

En C++: **MPI::Finalize();**

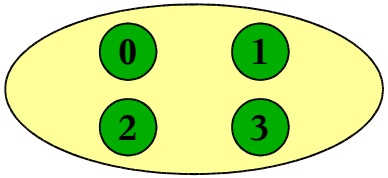
Fonction collective! (appelée par tous les process)

Pour une définition précise des fonctions (type, arg optionnels, etc...) voir aide mémoire IDRIS et manuel de référence MPI en ligne.

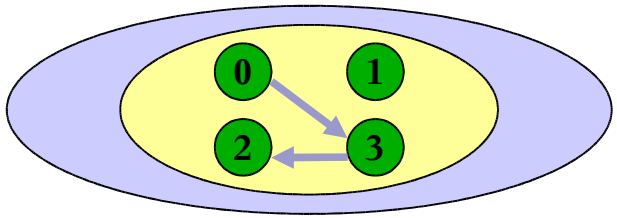
Goupes et communicateurs: objets de base de MPI

Groupes et communicateurs sont des objets MPI opaques: on les manipule par une "poignée" (variable de type INTEGER en fortran)

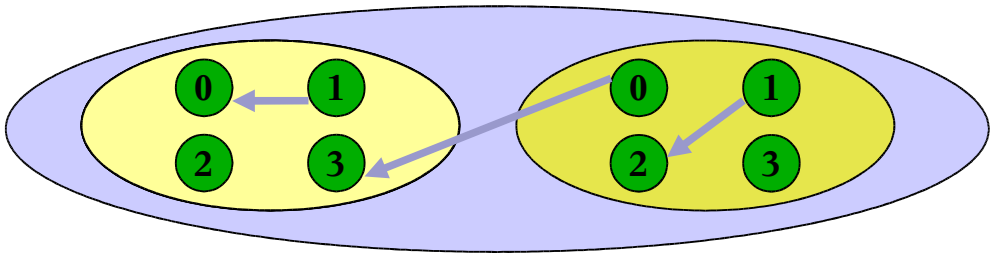
- **Groupe:** Ensemble de process.



- **Intracommunicateur:** Struture d'échange de messages entre process contenant un groupe **unique**.



- **Intercommunicateur:** Struture d'échange de messages entre process contenant **deux** groupes.



Manipulation des groupes et communicateur.

L'appel à MPI_INIT() définit un **communicateur par défaut: MPI_COMM_WORLD** (constante entière MPI)

Souvant, **MPI_COMM_WORLD** est le seul usage des communicateurs/groupes que fait un code. Mais il est parfois intéressant de créer des sous-groupes/sous-communicateur.

Quelques fonctions pour créer-manipuler-détruire les groupes et communicateur.

- **MPI_COMM_SIZE(comm,size,err)**: locale. Renvoie **size**, le nb de process dans **comm**.
- **MPI_COMM_RANK(comm,rank,err)**: locale. Renvoie **rank**, le n° du process appelant.

- **MPI_COMM_GROUP(comm,group,err)**: locale! Crée un groupe **group** avec tout les process de **comm.**
- **MPI_GROUP_INCL(group,n,rank_array,subgroup,err)**: locale. Crée un groupe **subgroup**, avec un sélection de **n** process de **group** définie par **rank_array**.
- **MPI_COMM_CREATE(comm,subgroup,subcomm,err)**: collective. Crée l'intra-communicateur **subcomm** de **subgroup**,

- **MPI_GROUP_SIZE(group,size,err)**: locale. Renvoie **size**, le nb de process dans **group**.
- **MPI_GROUP_RANK(group,rank,err)**: locale. Renvoie **rank**, le n° du process appelant.

- **MPI_GROUP_FREE(group,err)**: locale. Désalloue le groupe **group**.
- **MPI_COMM_FREE(comm,err)**: collective. Désallooue le communicateur **comm**.

MPI:

communications point à point

Structure d'un message MPI

Par l'intermédaire d'un message, un process peut envoyer des données à un autre.

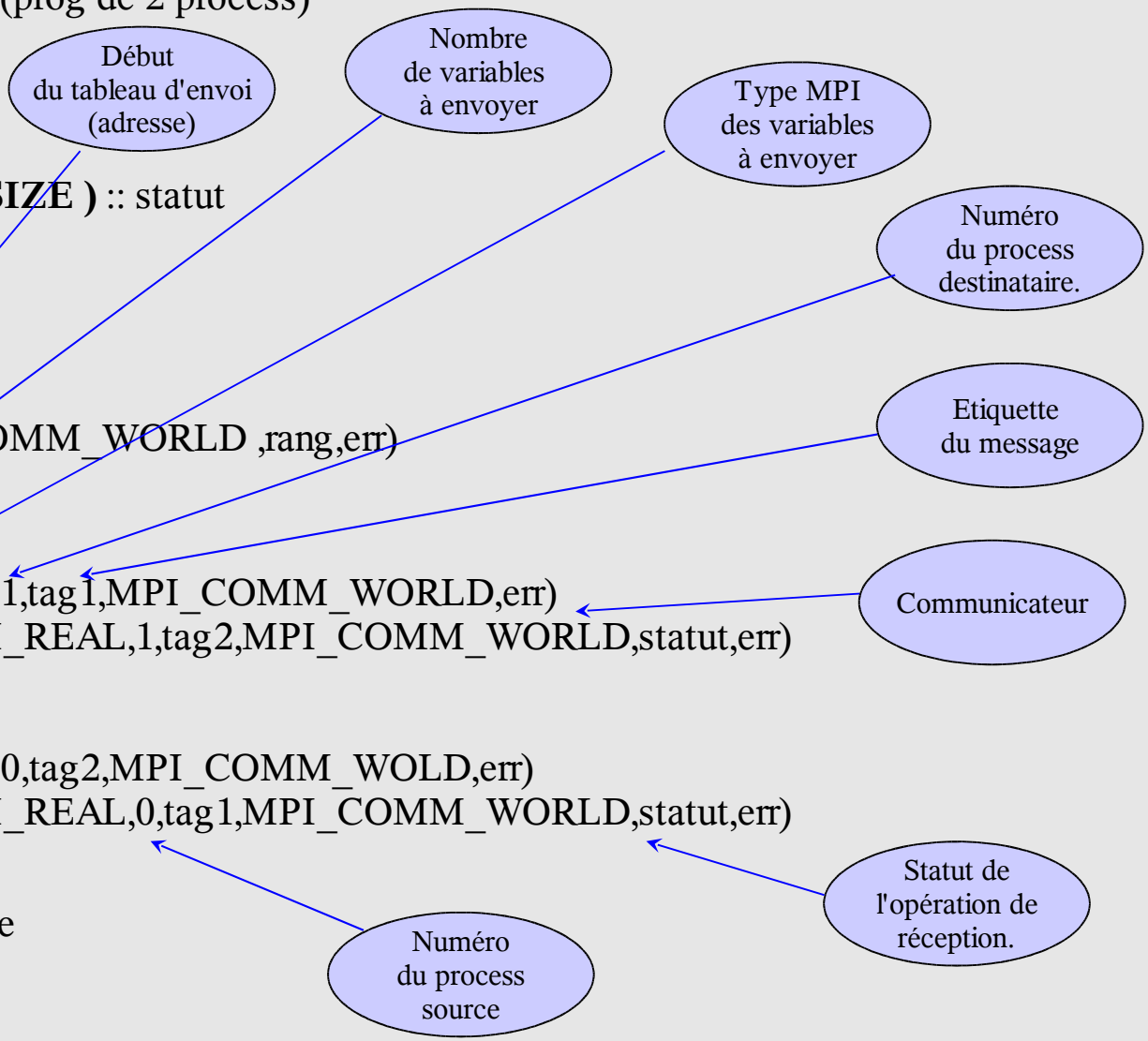
En plus des **données**, le message contient une "**enveloppe**" qui contient plusieurs champs:

- **Source**: rang du process qui envoie.
- **Destination**: rang du process qui reçoit
- **Etiquette**: entier qui identifie le message de manière unique.
- **Communicateur**: communicateur au sein duquel se fait l'échange.

Les fonctions d'envoi et de reception décrivent de manière **non-ambiguë** et **portable** la nature des données transmises.

MPI_SEND et MPI_RECV: communication de base.

```
program EX_1                          ! (prog de 2 process)

implicit none
include "mpif.h"
integer :: rang,err,tag1,tag2
integer, dimension( MPI_STATUS_SIZE ) :: statut
real, DIMENSION(10) :: x,x_remote

call MPI_INIT (err)
  tag1=1
  tag2=2
  call MPI_COMM_RANK (MPI_COMM_WORLD ,rang,err)
  x = rand()

  if(rang == 0) then
    call MPI_SEND(x,10,MPI_REAL,1,tag1,MPI_COMM_WORLD,err)
    call MPI_RECV(x_remote,10,MPI_REAL,1,tag2,MPI_COMM_WORLD,statut,err)
  endif
  if(rang == 1) then
    call MPI_SEND(x,10,MPI_REAL,0,tag2,MPI_COMM_WOLD,err)
    call MPI_RECV(x_remote,10,MPI_REAL,0,tag1,MPI_COMM_WORLD,statut,err)
  endif

  print*,'rang: ',rang, '  liste: ',x,x_remote
call MPI_FINALIZE (err)

end program EX_0
```



MPI_SEND et MPI_RECV: détails d'utilisation.

- Pour des raisons de **portabilité** (grille hétérogène), il faut donner un "**type MPI**" pour les variables envoyées. Voici la correspondance pour Fortran:

INTEGER:	**MPI_INTEGER**	**(MPI_INT en C)**
REAL:	**MPI_REAL**	**(MPI_FLOAT en C)**
DOUBLE PRECISION:	**MPI_DOUBLE_PRECISION**	**(MPI_DOUBLE en C)**
LOGICAL:	**MPI_LOGICAL**	
CHARACTER(1):	**MPI_CHARACTER**	**(MPI_CHAR en C)**

- On peut aussi envoyer des types **MPI_PACKED** et des types définis par l'utilisateur (structures).

- La taille du tableau (buffer) de reception doit être ≥ à la taille du message.

- Wildcards: MPI_RECV accepte **MPI_ANY_SOURCE** et **MPI_ANY_TAG** (constantes mpi) comme valeur pour la source et l'étiquette du message.

- La variable "statut" contient (en Fortran):
  - statut(MPI_SOURCE): source du message reçu.
  - statut(MPI_TAG): l'étiquette du message reçu.

- Entre 2 processeurs les messages sont reçus dans l'odre où ils sont envoyés.

Modes de communication: définitions

Communication bloquante: l'appel à la fonction ne "***retourne"*** que quand les ressources utilisées (p. e. emplacement mémoire de la variable envoyée) peuvent être réutilisées et que la fonction ***"complète"***.

Communication non-bloquante: l'appel à la fonction ***retourne*** avant que les ressources aient été libérées, et le programme continue. Il faut s'assurer qu'on ne modifie pas les ressources avant que la communication soit effectivement complétée. La fonction ne ***complètei*** qu'à ce moment là.

Mode d'envoi "buffered": le message est stoké dans une mémoire système locale avant d'être envoyé. La fonction d'envoi (bloquante ou non) ***complète*** quand la copie est finie mais avant l'envoi.

Mode d'envoi "synchrone": l'envoi effectif du message ne commence que quand le process reçoit le message « prêt » d'une commande de réception correspondante. La fonction d'envoi ne ***complète*** qu'à ce moment là.

Mode d'envoi "ready": le process émetteur *suppose* que le recepteur est près à recevoir sans vérifier, et envoie le message (meilleures performances). Si le récepteur n'a pas exécuté la commande de réception correcpondante -> Erreur! La fonction ***complète*** quand l'evoi est fini.

Mode d'envoi "standard": Suivant la disponibilité en mémoire système, MPI choisit lui même entre les modes "buffered" et "synchrone".

Il n'y a qu'un mode de reception.
MPI_SEND est une communication bloquante en mode standard.
MPI_RECV est une communication bloquante.

Autres fonctions de communication point à point

Il existe une panoplie de fonctions pour effectuer des communications bloquante ou non dans le différent mode. Elle s'utilisent avec les même arguements que MPI_SEND et MPI_RECV, plus un argument "request" pour les fonctions non-bloquantes.

	Bloquant	Non-bloquant
Standard	**MPI_SEND**	**MPI_ISEND**
Synchrone	**MPI_SSEND**	**MPI_ISSEND**
Ready	**MPI_RSEND**	**MPI_IRSEND**
Buffered	**MPI_BSEND**	**MPI_IBSEND**
Reception	**MPI_RECV**	**MPI_IRECV**

(nécessite **MPI_BUFFER_ATTACH**)

- Le mode buffered nécessite une copie mémoire de plus mais permet de continuer les calculs à coup sûr.
- Le mode ready diminue la latence, mais est délicat à utiliser (synchronisation).

Gérer la complétion d'une opération non-bloquante.

Syntaxe d'un envoi non-bloquant:

MPI_ISEND(val,count,datatype,dest,etiquette,comm,**requete**,err)

Le code peut continuer mais garde la trace de l'envoi grâce à **requete** (poignée vers un objet MPI). Il peut utiliser **requete** ultérieurement pour contrôler si la communication est complète:

- **MPI_WAIT(requete,statut,err)**: attentant que la communication associée à requete soit complétée. statut contient des infos sur la communication.
- **MPI_TEST(requete,flag,statut,err)**: flag=true si la communication associée à requete est complétée. Sinon, flag=false, et requete est désallouée! Pas de vérification ultérieure possible.
- **MPI_REQUEST_GET_STATUS(request,flag,status,err)**: teste la complétion sans désallouer la requête, même si flag=true.
- **MPI_REQUEST_FREE(request,err)**: désalloue la requête.

Il est possible de gérer des complétions mulitples grâce à MPI_WAITANY, MPI_WAITALL, MPI_WAITSOME et les équivalent pour TEST.

Attention: MPI_REQUEST_FREE n'anulle pas la communication non bloquante. Pour cela il faut utiliser **MPI_CANCEL(resquest,err).**

Communications non-prédictibles

Dans certains algorithmes, le besoin de communications peut dépendre des données initiales (ex: modif dynamique des domaines):

- Les besoins de l'envoi sont déterminés **localement** par le calcul.
- **Comment déterminer** les réceptions à effectuer?

On peut tester périodiquement l'arrivée de messages sans les recevoir.

MPI_IPROBE(source,etiquette,comm,flag,status): non bloquant!
flag détermine si un message est arrivé ou non. **source** et **etiquette** peuvent prendre les valeurs MPI_ANY_SOURCE et MPI_ANY_TAG, si **flag**=true, **status** contient la source et l'étiquette.

Et si on ne connait pas la longueur du message?

- L'étiquette peut servir à coder le nombre et le type de donnée.
- Une précommunication à un format standardisé peut annoncer l'étiquette, le type et le nombre de données d'un message à venir.
- Il est peut-être temps de passer à des communications « one-sided ».

Synchronisation

Synchronisation globale:

MPI_BARRIER(comm,ierr): fonction collective

Permet de bloquer les process du communicateur **comm** jusqu`à ce que le dernier soit arrivé à la barrière.

Synchronisation locale:

On peut utiliser une **communication synchrone bloquante** (MPI_SSEND/RECV) pour synchroniser 2 process.

Si on a besoin de synchroniser de façon répétée un sous-groupe de process, il faut sans-doute définir un nouveau sous-groupe-MPI et un nouvel intra-communicateur.

MPI:

communications globales (fonctions collectives)

Communication de type "broadcast"

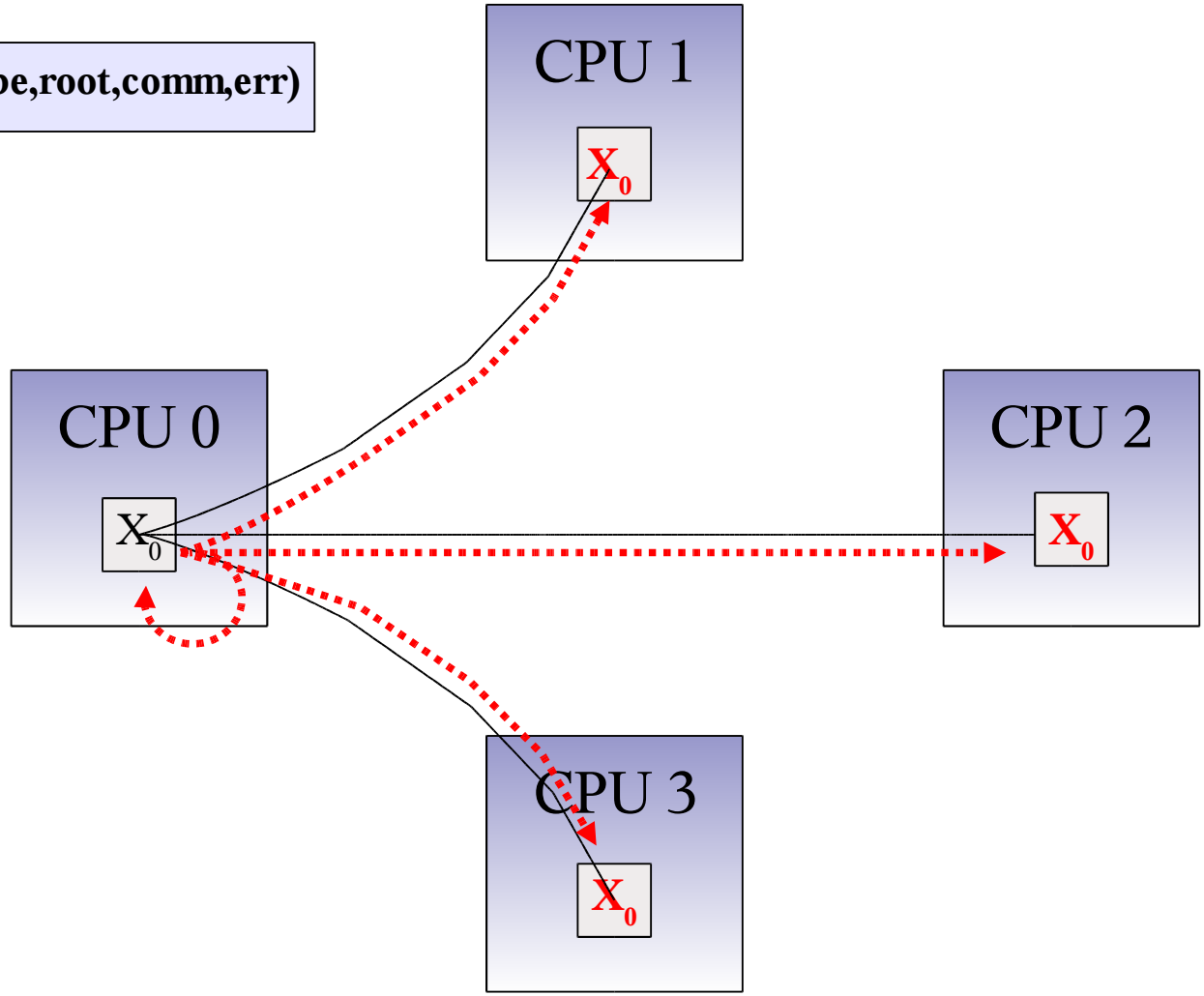
Il s'agit de diffuser aux autres processeurs une valeur connue sur un seul:

MPI_BCAST(address,count,datatype,root,comm,err)

root désigne le rang du process qui diffuse l'information. Les autres arguments ont la même signification que dans MPI_SEND.

C'est une **fonction collective** : elle doit être appelée par **tous** les process.

Il faut privilégier les communications collectives pour la simplicité de la programmation.



Communication de type "gather"

Il s'agit de **rassembler** sur l'un des process des données réparties sur l'emsemble des process du communicateur :

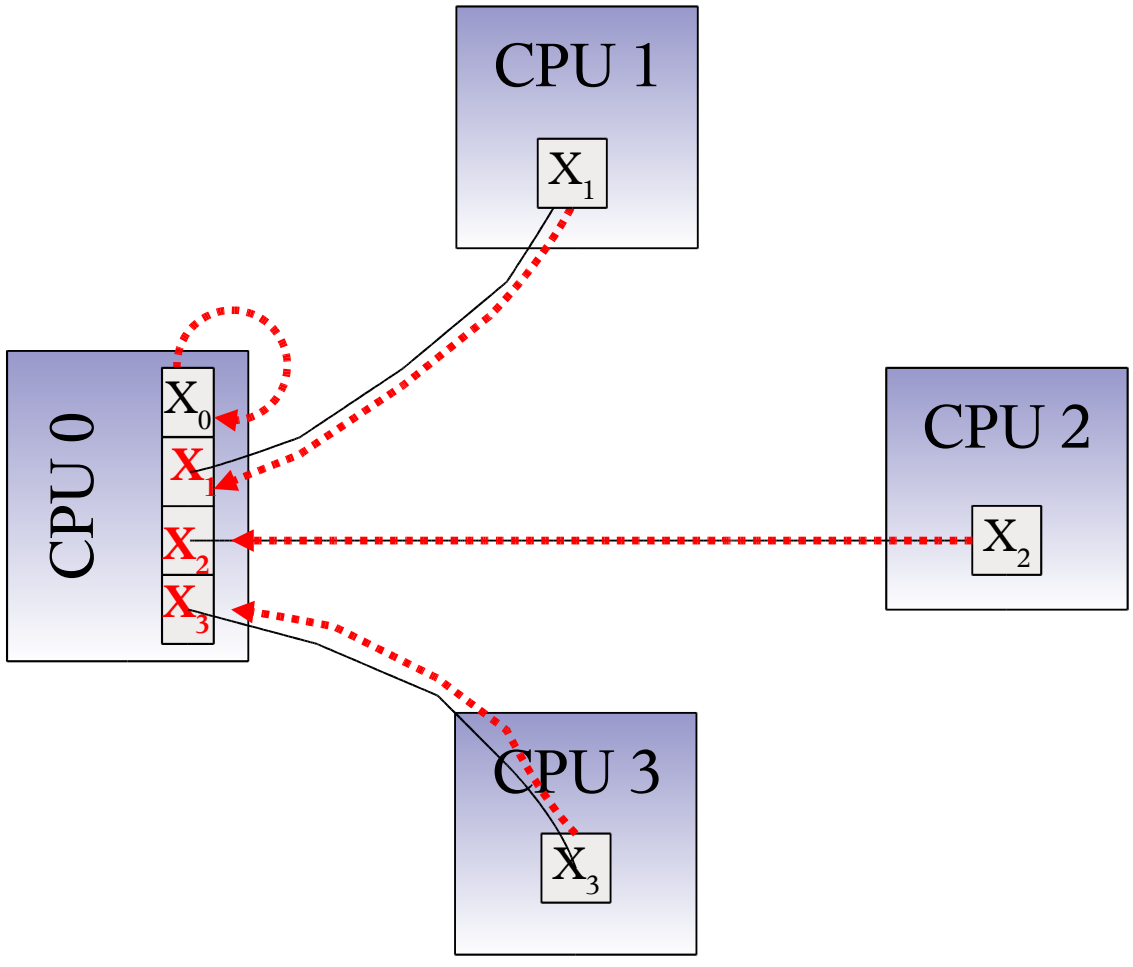
MPI_GATHER(s_add , s_count , s_type , r_add , r_count , r_type , root , comm , err)

- **root** désigne le rang du process qui reçoit les données. **r_add** désigne l'adresse (nom de variable en fortran) où les données reçues sont stockées.

- Les données sont stockées dans l'ordre de rang des processeurs qui envoient.

- **r_count** est le nombre de variables de type **r_type** reçue de **chaque** process. Donc, la plupart du temps:
s_count=r_count=constante !

- Sur le process **root,** s_add peut être remplacé par MPI_IN_PLACE. On suppose alors que les données à envoyer pour le process **root** sont déjà à leur place dans le tableau de reception.



Autres communications de type "gather"

Pour recevoir une **quantité** de données **différente** de chaque process ou les disposer de manière **non-consécutive** dans le tableau de réception, on utilise la version vecteur:

MPI_GATHERV(s_add , s_count , s_type , r_add , r_counts , disp , r_type , root , comm , err)

- **s_count** peut maitenant être différent sur chaque process.
- **r_counts** est un tableau de NB_PROCS entiers.
- **r_counts(i)** doit avoir la valeur de **r_count** sur le process i.
- **disp** est un tableau de NB_PROCS entiers.
- Les données reçues du process i sont stockée à l'adresse: **r_add+disp(i)*sizeof(r_type)**

Pour réaliser une opération sur les données reçues, on utilise une **fonction de réduction**:

MPI_REDUCE(s_add , r_add , count , datatype , op , root , comm , err)

- Les valeurs dans **s_add** sur les différents process sont combinées, élément à élément si **count≠1**, et stockées dans **r_add** sur le process **root.** La combinaison est faite par l'opérateur **op**.
- En fortran **op** peut prendre comme valeurs: **MPI_SUM, MPI_PROD, MPI_MAX, MPI_MIN**, etc...
- Il est possible de créer ses propres opérateur grâce à **MPI_OP_CREATE.**

Communications de type "scatter"

Il s'agit de **répartir** sur les process du communicateur des données présentes sur l'un des process:

MPI_SCATTER(s_add , s_count , s_type , r_add , r_count , r_type , root , comm , err)

- root désigne le rang du process qui envoie les données. **s_add** désigne l'adresse (nom de variable en fortran) où les données à envoyer sont stockées.

- Les données sont envoyées par paquet de **s_count** aux process du communicateur par ordre de rang.

- En général **r_count = s_count.**

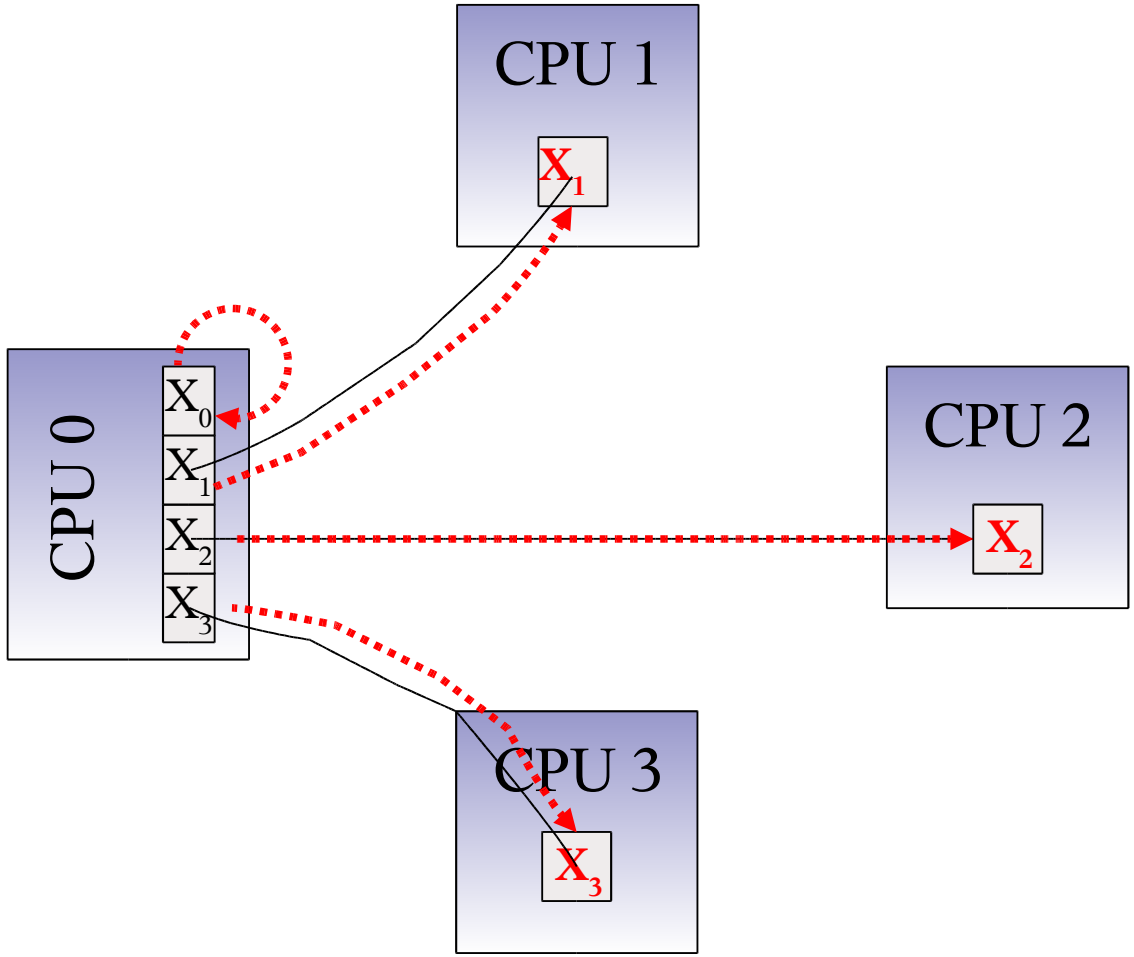
- Il existe une variante "vecteur":

MPI_SCATTERV

- Il existe aussi:

MPI_REDUCE_SCATTER

Opére une reduction sur des tableaux, élément à élément, et stocke le résultat pour les i-ème éléments sur le process de rang i.



Communications de type "allgather"

Même fonction que MPI_GATHER, mais chaque process reçoit le résultat:

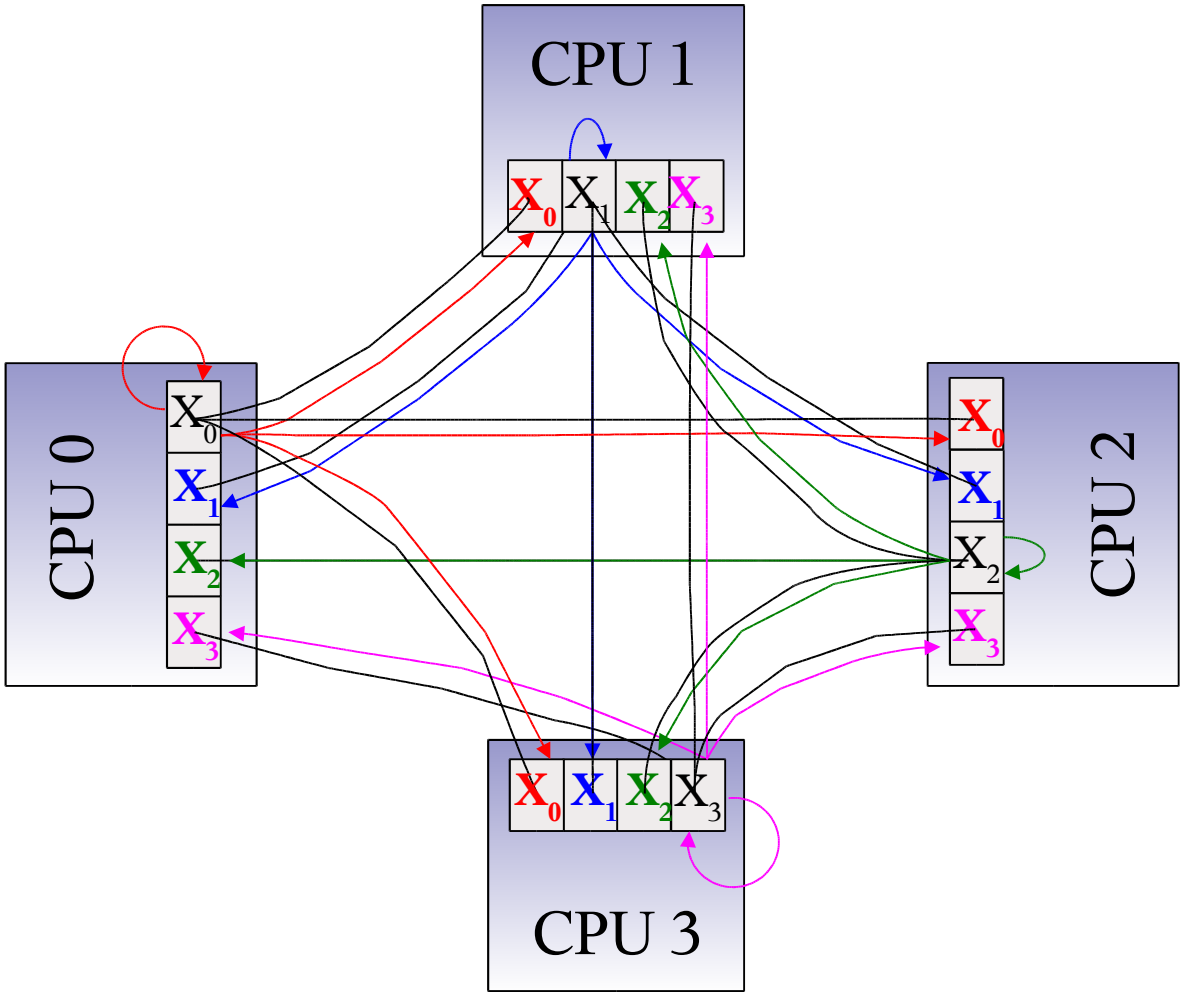
MPI_ALLGATHER(s_add , s_count , s_type , r_add , r_count , r_type , comm , err)

- Pas de **root** !
- Autres arguments identiques à MPI_GATHER.
- Equivalent à NB_PROCS appels à MPI_GATHER avec à chaque fois un process différent comme root.
- Il existe une version vecteur:

  MPI_ALLGATHERV

- Et une version avec réduction:

  MPI_ALLREDUCE



Communications de type "all-to-all"

Même fonction que MPI_ALLGATHER, mais chaque process reçoit des données différentes:

MPI_ALLTOALL(s_add , s_count , s_type , r_add , r_count , r_type , comm , err)

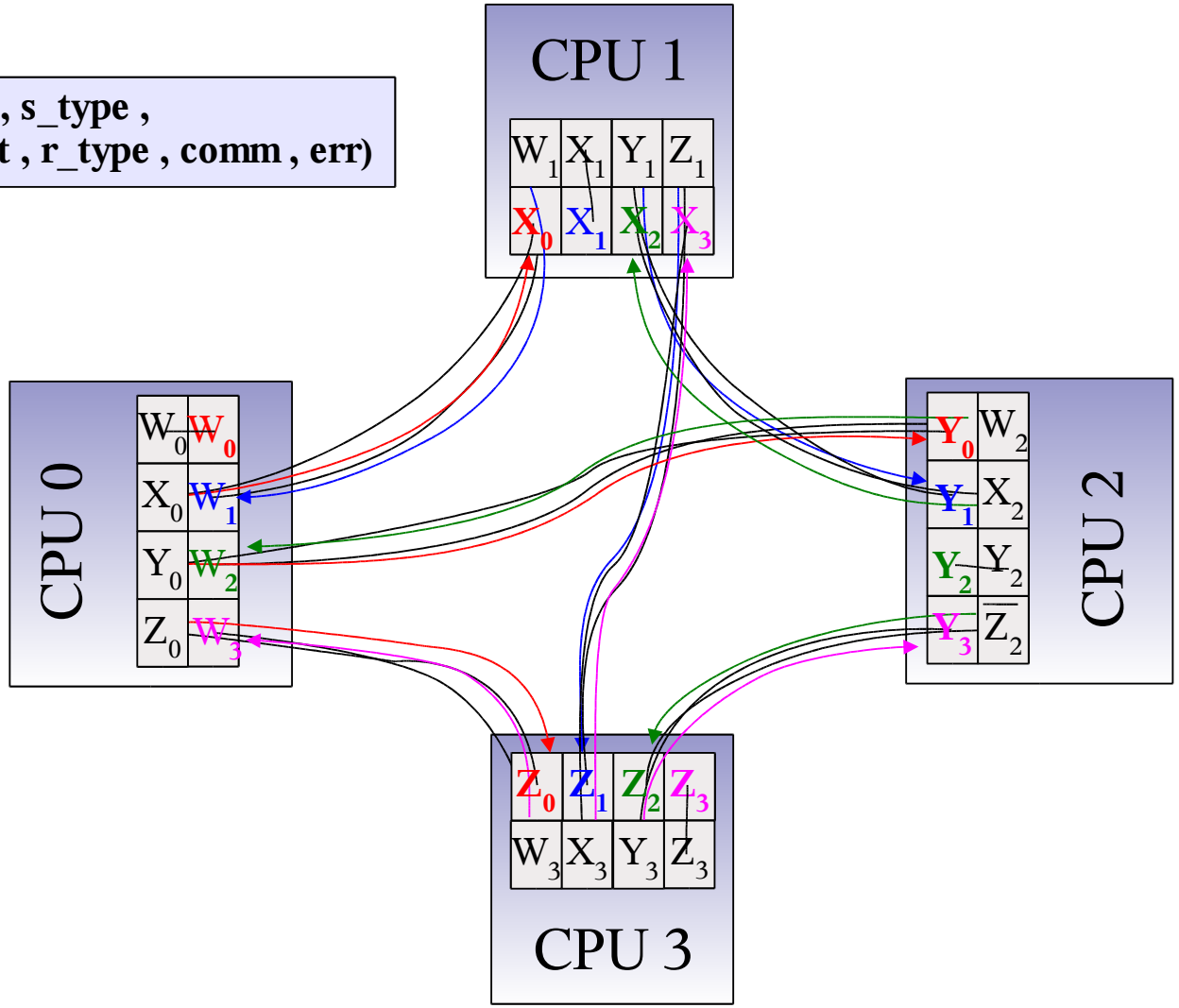
- Arguments identiques à MPI_ALLGATHER.

- Mais ici s_add et r_add désignent des tableaux de même taille.

- Il existe deux versions vecteurs, pour pouvoir faire varier le nombre, le type et la position des données envoyées et reçues:

MPI_ALLTOALLV
MPI_ALLTOALLW

Avec MPI_ALLTOALLW on peut tout faire! Ou presque...



Types dérivés:

variables MPI définies par l'utilisateur

Déclarer un type dérivé

Les fonctions de communication MPI utilise comme argument des types MPI standard comme MPI_INTEGER ou MPI_REAL. Il est possible de définir des types plus complexes. Cela peut servir, par exemple:

- A envoyer une section de tableau.

- A envoyer une variable de type structure définie dans le programme.

Exemple de déclaration d'un type dérivé simple:

```
INTEGER :: err,MPI_vector
call MPI_TYPE_CONTIGUOUS(3,MPI_REAL,MPI_vector,err)
call MPI_TYPE_COMMIT(MPI_vector,err)
... communications ...
call MPI_TYPE_FREE(MPI_vector,err)
```

On définit ici un type **MPI_vector**, constitué de **3 MPI_REAL** stockés consécutivement en mémoire. MPI_vector peut servir à définir d'autres types dérivés.

Il faut "compiler" le type avec **MPI_TYPE_COMMIT** avant de pouvoir l'utiliser dans une communication.

Transmettre une section de tableau

Les sections de tableaux sont un outils puissant de fortran. On peut définir des types dérivés pour envoyer des sections de tableau. Exemple:

Les éléments foncés constituent la section x(2:12:3) du tableau x(1:12). Définissons un type MPI correspondant.

```
INTEGER :: err,MPI_vector_section
REAL, DIMENSION(12) :: x,y

call MPI_TYPE_VECTOR(4,1,3,MPI_REAL,MPI_vector_section,err)
call MPI_TYPE_COMMIT(MPI_vector_section,err)

call MPI_ALLREDUCE(x(2),y(2),1,MPI_vector_section,MPI_SUM,MPI_COMM_WORLD,err)
```

MPI_TYPE_VECTOR construit le type MPI_vector_section, constitué de 4 blocs de 1 MPI_REAL, avec un pas entre les débuts de blocs de 3 MPI_REAL. 4, 1 et 3 peuvent varier à volonté.

Attention: les tableaux multidimensionnel sont stockés en mémoire sous forme de vecteur 1D. Il faut savoir quel est l'indice qui varie le plus vite. On peut alors définir des types sections recursivement sur les dimensions.

On peut définir des sections avec **pas variable entre blocs** avec **MPI_TYPE_CREATE_INDEXED_BLOCK**, et avec **pas et longueur de blocs variables** avec **MPI_TYPE_INDEXED.**

Type dérivé MPI correspondant à une struture

```
integer                       :: err,MPI_integer_length,MPI_real_length,MPI_logical_length
integer                       :: MPI_vector_length,MPI_vector,MPI_particle
integer,  dimension(10) :: array_of_block_length,array_of_types,array_of_displacement

call MPI_TYPE_EXTENT(MPI_INTEGER,MPI_integer_length,err)
call MPI_TYPE_EXTENT(MPI_REAL,MPI_real_length,err)
call MPI_TYPE_EXTENT(MPI_LOGICAL,MPI_logical_length,err)

call MPI_TYPE_CONTIGUOUS(3,MPI_REAL,MPI_vector,err)
call MPI_TYPE_COMMIT(MPI_vector,err)
call MPI_TYPE_EXTENT(MPI_vector,MPI_vector_length,err)

array_of_block_length(1:3) =  (/1,2,1/)
array_of_types(1:3) = (/MPI_INTEGER,MPI_vector,MPI_REAL/)
array_of_displacement(1) = 0
array_of_displacement(2) = MPI_integer_length
array_of_displacement(3) = array_of_displacement(2) + 2*MPI_vector_length

call MPI_TYPE_STRUCT(3,array_of_block_length(1:3),array_of_displacement(1:3)  &
          &       ,array_of_types(1:3),MPI_particle,err)
call MPI_TYPE_COMMIT(MPI_particle,err)
```

Pour la portabilité (mesure de taille mémoire en bits)

Définition d'un type vecteur

Description du type structure (déplacements en bits)

Déclaration du type

Pas de moyen de définir des types objets dans MPI de base!

MPI:

communications "one-sided"

Introduction

Que faire quand le process receveur ne sait pas qu'il doit recevoir?

- Lancer régulièrement de communications globales. Performances?
- Utiliser MPI_IPROBE et des communications non bloquantes... synchronisation délicate.
- Utiliser des **communications "one-sided".**

Les communications one-sided, permettent d'utiliser le caractère SMP d'une architecture pour améliorer les performances, mais ne constituent pas un modèle complet de programmation SMP.

Déclarer une « fenêtre » de communication one-sided.

Il faut définir une zone mémoire accessible par les process distants lors de communications one-sided:

MPI_WIN_CREATE(add , size , disp_unit , info , comm , win , err)

- **add**: addresse de début de la fenêtre (ex, nom de tableau).
- **size**: taille de la fenêtre en bit.
- **disp_unit**: unité de déplacement pour les accès ultérieurs.
- **info**: poignée vers un tableau où les 3 premiers seront stockés.
- **comm**: communicateur
- **win**: poignée (entier) attachée à la fenètre créée.
- **err**: code d'erreur

MPI_WIN_CREATE est une fonction **collective.** La fenêtre n'a pas forcement la même taille et position sur tous les process.

MPI_WIN_FREE(win,err) pour désallouer la fenêtre.

MPI_PUT / MPI_GET: écriture / lecture

Il y a 2 fonctions principales pour réaliser une communication one-sided:

MPI_PUT(loc_add , loc_count , loc_type , remote_rank , remote_disp &
& , remote_count , remote_type , win , err)
MPI_GET(loc_add , loc_count , loc_type , remote_rank , remote_disp &
& , remote_count , remote_type , win , err)

- **loc_add:** adresse de début des données sur le process qui appelle la fonction
- **remote_disp:** Définit l'endroit où lire/écrire les données par le déplacement en disp_unit par rapport au début de la fenêtre sur le process cible.
- **loc_count,remote_count:** Nombres de variables à lire/écrire.
- **loc_type,remote_type:** Type des variables à lire/écrire.
- **remote_rank:** Numéro du process cible.
- **win:** fenêtre dans laquelle se fait la communication.

Il s'agit de **communications non bloquantes !** Il faut vérifier la complétion par des appel de synchronisation.

Il existe aussi MPI_ACCUMULATE.

Complétion des communications one-sided.

```
CALL MPI_WIN_CREATE(x , 1000, MPI_real_length, info , MY_COMM_WORLD, x_win , err)
CALL MPI_WIN_FENCE(MPI_MODE_NOPRECEDE,x_win,err)

target=mod(rank+1,nb_proc)
index_to_put=local_computation()
CALL MPI_PUT(x(index_to_put),1,MPI_REAL,target,index_to_put,1,MPI_REAL,x_win,err)

CALL MPI_WIN_FENCE(MPI_MODE_NOSUCCEED,x_win,err)
CALL MPI_WIN_FREE(x_win,err)
```

MPI_WIN_FENCE est une **fonction collective**. Elle agit comme une barrière, **aucun** process du groupe ne continue tant que toutes les communications de la fenêtre ne sont pas complétées. Le premier argument sert à l'optimisation, il peut toujours valoir 0. Il peut prendre les valeurs:

- MPI_MODE_NOSTORE: pas d'**écriture** en local dans la fenêtre depuis le dernier FENCE.
- MPI_MODE_NOPUT: Pas de PUT vers la fenêtre locale d'ici le prochain FENCE.
- MPI_MODE_NOPRECEDE: Pas de communications antérieures à compléter.
- MPI_MODE_NOSUCCED: Pas de communications ultérieures.

On peut combiner ces valeurs (voir manuel de réference). Les optimisations correspondantes ne sont parfois pas implémentées.

Il est possible de **synchroniser les process 2 à 2** avec **MPI_WIN_START**, **MPI_WIN_COMPLETE**, **MPI_WIN_POST** et **MPI_WIN_WAIT**.

Ce que je n'ai pas traité:

- Topologie de process
- Entrés-sorties parallèles
- Inter-communications
- Création dynamique de process
- Packing de données