# Introduction
# à la programmation par directives

O. Coulaud

IT389

Inria

# Plan

Introduction

Le modèle OpenMP

Aspect de base

- Région parallèle
- Distribution du travail
- Attributs de données
- Les directives de synchronisation

Approche à base de tâches

Approche SIMD

Web: https://moodle.bordeaux-inp.fr/course/view.php?id=2759

Email: olivier.coulaud@inria.fr

# Références

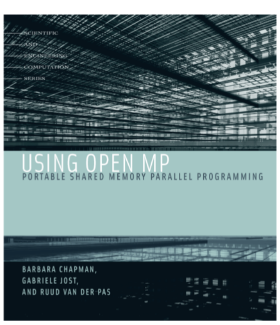
Site Web officiel : www.openmp.org

**Spécification OpenMP 4.5**

Video/slides https://www.openmp.org/resources/openmp-presentations/

Tutorials: https://www.openmp.org/resources/tutorials-articles/

Books:




Using OpenMP, Barbara Chapman, Gabriele Jost, Ruud Van Der Pas, Cambridge, MA : The MIT Press 2007, ISBN: 978-0-262-53302-7

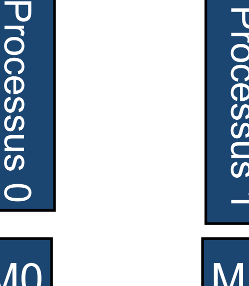
Communauté

La communauté des chercheurs et des développeurs d'OpenMP académique et industrielle

- http://www.compunity.org/

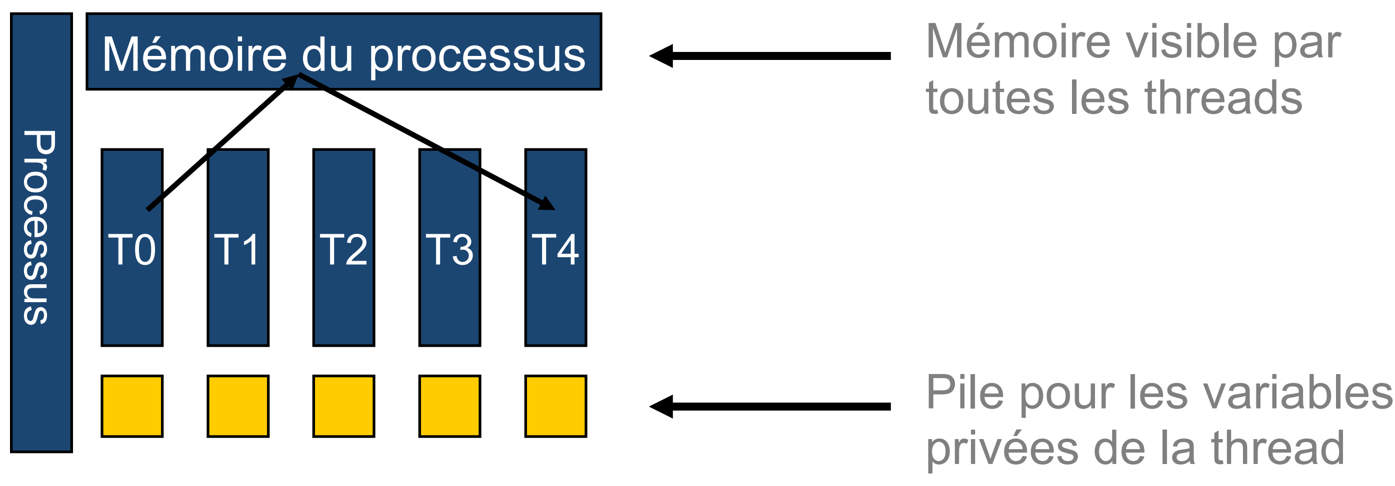
# Processus versus threads

Processus




Deux unités indépendantes = deux pids

Threads



# Les Threads POSIX

Une autre API de programmation mémoire partagée.

Bas niveau d'abstraction par rapport à OpenMP

- Seulement des fonctions, pas de directives ;
- Plus flexible, mais plus difficile à implémenter et à maintenir ;
- OpenMP peut être implémenté au-dessus des threads POSIX.

Disponibilité

- Pas d'interface en Fortran des threads POSIX

# Le modèle OpenMP

# Pourquoi OpenMP en 1997

Pas de portabilité pour les applications à mémoires partagées

- Chaque constructeur avait son API
- X3H5 et PCF n'ont pas abouti

Portabilité au travers de MPI

Pas de langages parallèles dominants

Les machines sont là

- Origin2000, SUN, …
- Exemplar, PC, ...

et les applications arrivent ...

# Ce qu'apporte OpenMP

Une portabilité

* Fortran 77 et 90 depuis novembre 97
* C et C++ depuis décembre 98
* Unix et NT

Haut niveau de programmation (Directives)

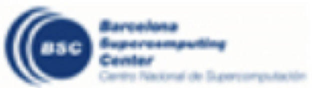
Modèles de programmation

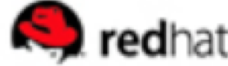
* parallélisme à grains fins (boucle)
* parallélisme à gros grains (SPMD)

Pour l'extensibilité

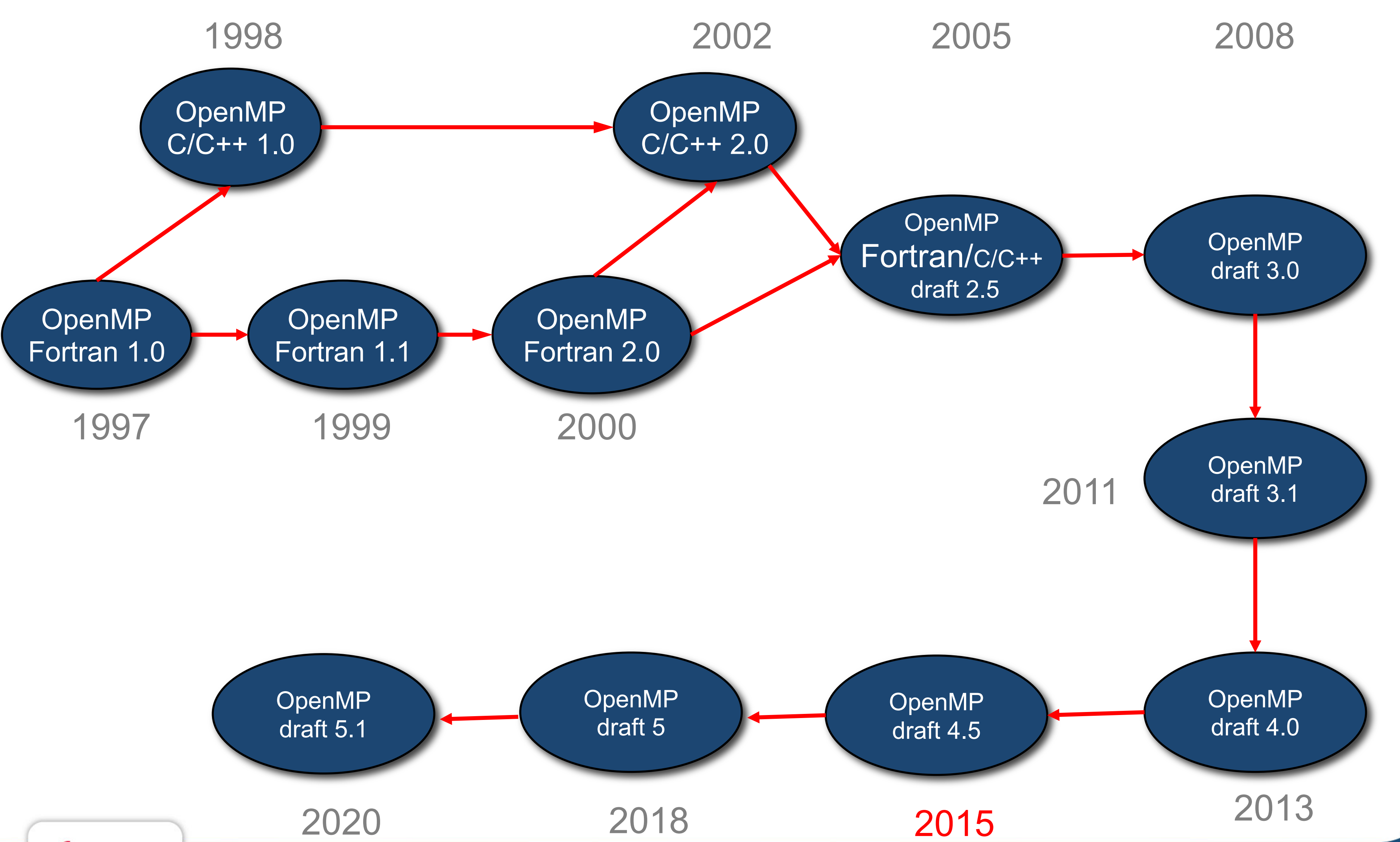
Une parallélisation incrémentale

# Principaux acteurs





# Évolution d’OpenMP



# Comment utiliser OpenMP

Annotation Source

Fortran/C/C++ compilateur

compilateur séquentiel

Programme Séquentiel

compilateur OpenMP

Programme Parallèle

# Architecture OpenMP

Couche utilisateur

Application

Utilisateurs finaux

Couche de Programmation API OpenMP

Directives
de Compilation

Bibliothèque OpenMP

Variables
d' environnement

Couche Système

Bibliothèque (Runtime)

système d' exploitation

# Modèle d'exécution

Thread maître

Thread initial

FORK

JOIN

FORK

JOIN

Région parallèle 1
7 threads

Région parallèle 2
5 threads

Partie séquentielle = région parallèle implicite

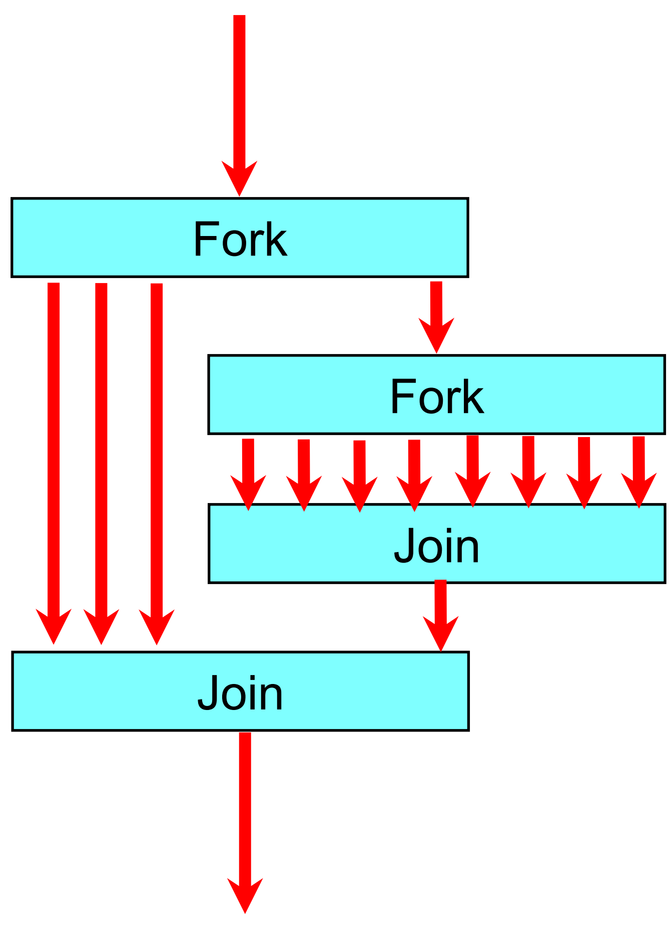
Modèle Fork and join : le maitre lance un ensemble de threads

team = master + autres threads

# Modèle d’exécution (suite)

Le modèle Fork/Join peut être emboité

- Géré automatiquement à la compilation
- Indépendant du nombre de threads s'exécutant actuellement.



# Modèle mémoire

Les threads communiquent en partageant des variables

Le partage est défini par :

- Toute variable qui est vue par deux ou plusieurs threads est partagée ( mémoire)
- Toute variable qui est vue par un seul thread est privée. ( sa propre mémoire)

Les situations de concurrence (*race condition*) sont possibles

- Utiliser des synchronisations pour éviter des conflits sur les données ;
- Changer le statut de la variable pour minimiser le besoin de synchroniser.

# Modèle mémoire

Une variable partagée est visible par tous les threads.

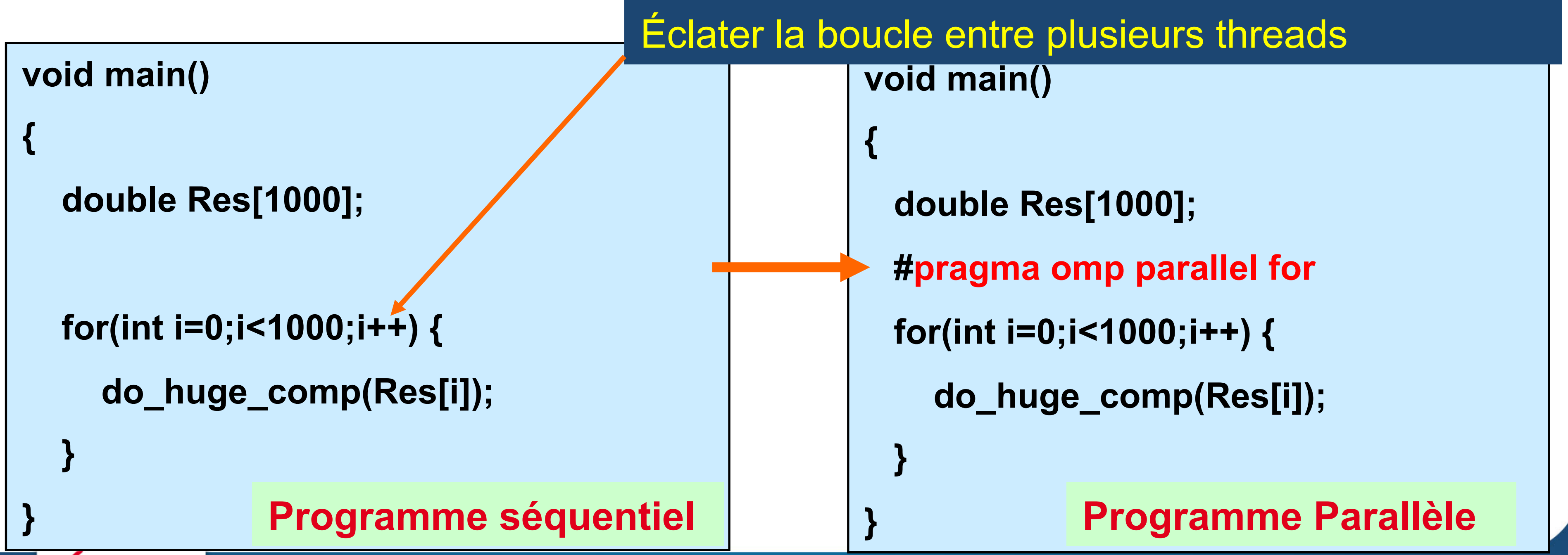
Une variable privée est dupliquée sur les threads.

Une variable interne dans une région parallèle est une variable appartenant à la mémoire propre du thread (pile).

# Comment est utilisé classiquement OpenMP ?

OpenMP est classiquement utilisé pour paralléliser des boucles :

- trouver la boucle la plus consommatrice en temps CPU.
- éclater sur plusieurs threads/processeurs.



# Exemple

## Code source

```
#include "omp.h"
….;
#pragma omp parallel
{
        … région parallèle
}
```

## Compilation

GNU : gcc **-fopenmp** filename.cc –o filename

INTEL : icc **-qopenmp** filename.cc –o filename

## **Exécution**

```
$ export OMP_NUM_THREADS=4
$ filename
```

# Un exemple de programme

```
#include <omp.h>

main () {

 int var1, var2, var3;

Serial code
 . . .
 // Début de la section parallèle.  Lancement d'un ensemble de threads
//   Précisez la portée des variables
 #pragma omp parallel private(var1, var2) shared(var3)
 {
  //  Section Parallèle exécutée par tous les threads
 . . .
 // Tous les threads rejoingnent le thread maitre et disparaissent
}
Reprise du code de séquentiel
. . .
 }
```

Master

# OpenMP: Blocs structurés (C/C++)

La plupart des constructeurs OpenMP s'appliquent à des blocs structurés

- **Bloc structuré** : un bloc avec un point d'entré au début et un point de sortie à la fin.
- Les seuls branchements autorisés sont le STOP en Fortran et exit() en C/C+

```
#pragma omp parallel
{
        int id = omp_get_thread_num();
more:  res(id) = do_big_job(id);
        if(conv(res(id)) goto more;
}
   printf(" All done \n");
```

Un bloc structuré

```
     if(go_now()) goto more;
#pragma omp parallel
{
        int id = omp_get_thread_num();
more:  res(id) = do_big_job(id);
        if(conv(res(id)) goto done;
         go to more;
}
done:      if(!really_done()) goto more;
```

Un bloc non structuré

# OpenMP: blocs structurés

C/C++: un bloc est une déclaration unique ou un groupe d'instructions entre les accolades { }

```
#pragma omp parallel
{
    id = omp_thread_num();
    res(id) = lots_of_work(id);
}
```

```
#pragma omp for
  for(I=0;I<N;I++){
      res[I] = big_calc(I);
      A[I] = B[I] + res[I];
  }
```

Fortran: un bloc est une déclaration unique ou un groupe d'instructions entre la paire de directives.

```
C$OMP PARALLEL
10    wrk(id) = garbage(id)
      res(id) = wrk(id)**2
      if(conv(res(id)) goto 10
C$OMP END PARALLEL
```

```
C$OMP PARALLEL DO
      do I=1,N
           res(I)=bigComp(I)
       end do
C$OMP END PARALLEL DO
```

# Syntaxe d'OpenMP

La plupart des fonctionnalités en OpenMP sont des directives de compilations.

Les directives prennent la forme suivante :

**Sentinelle directive [clause [clause]…]**

| FORTRAN | C/C++ |
| --- | --- |
| Format fixe : **C$OMP !$OMP *$OMP**<br>Format libre : **!$OMP** | **#pragma omp** |
| Module Fortran 95 **OMP_LIB** | Fichier d'include **:** ***omp.h*** |

Un programme OpenMP par directives peut-être compilé par un compilateur qui ne supporte pas openMP.

# Syntaxe d'OpenMP

Compilation conditionnelle

- Macro du préprocesseur _OPENMP

Exemple

C/C++/Fortran

```
#ifdef _OPENMP
     iam = omp_get_thread_num()
#endif
```

# Directives OpenMP

Directives pour créer une zone parallèle :

région parallèle

Directives pour partager du travail :

- Do/for, workshare, sections, task

Directives pour le statut des données :

- shared, private, firstprivate, lastprivate threadprivate
- reduction, ...

Directives de synchronisation :

- barrier, critical, atomic, flush
- nowait

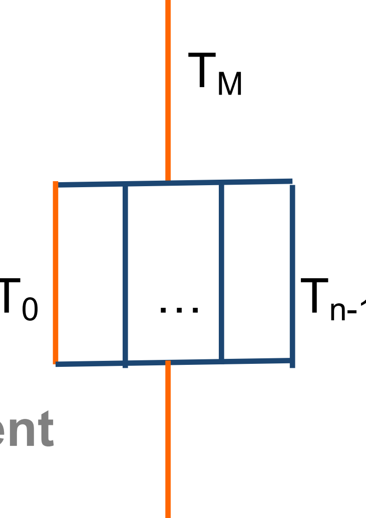
# OpenMP en 1 transparent

Extensions OpenMP

| Structure de contrôle parallèle | Partage de travail | Données | Synchronisation | Fonctions, Environnement runtime |
|---|---|---|---|---|
| Gouverne le flow de contrôle du programme<br><br>Directive<br>parallel | Distribue le travail sur les threads<br><br>Directives<br>for<br>sections<br>tasks,<br>taskIloop | Porté des variables<br><br>Clauses<br>shared,<br>private<br>firstprivate,<br>… | Coordonne l'activité des threads<br><br>Directives<br>critical, atomic<br>barrier, …<br>taskwait, … | Coordonne l'activité des threads<br><br>Directives<br>omp_set_num_threads()<br>omp_get_ thread_num ()<br>…<br>Variables<br>OMP_NUM_THREADS, … |

# RÉGION PARALLÈLE

# OpenMP : Région Parallèle (1)

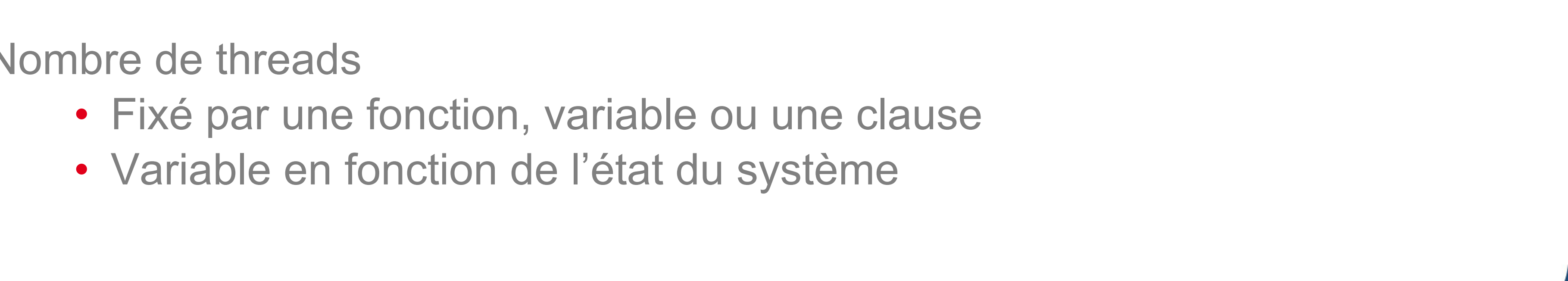



Création uniquement par

**#pragma omp parallel** [clause(,), clause, ….]
bloc de code à exécuter par chaque thread

Duplication de l'exécution

- **chaque thread joue le même code – les données peuvent être différentes**
- autorise du travail en fonction du numéro du thread
- Seul le thread maître continue à la fin



Barrière implicite à la fin

Nombre de threads

- Fixé par une fonction, variable ou une clause
- Variable en fonction de l'état du système

# OpenMP : Région Parallèle (2)

Clauses

- shared (list)
- private (list)
- firstprivate (list)
- default (shared | none) en C/C++.

- reduction (opérateur : list)
- copyin (list)

- if (expression logique scalaire)
- num_threads (expression entière scalaire)

} Contrôler la granularité.

- proc_bind (master | close | spread) — Contrôler le placement.

# OpenMP : Région Parallèle (3)

Restriction

- Pas de branchement d'entré ou de sortie (non conforme)
  le code doit être un bloc structuré
- Le code ne doit pas dépendre de l'ordre l'exécution des instructions
- une seule condition *if*
- une seule *num_threads* avec une expression positive
- Fortran
  - PARALLEL / END PARALLEL dans la même unité
  - Comportement des I/O asynchrones par différentes threads est non spécifié
- C/C++
  - Un throw exécuté dans un thread est attrapé par le même thread et reste dans la même région parallèle.

# OpenMP : Région Parallèle (4)

Exemple

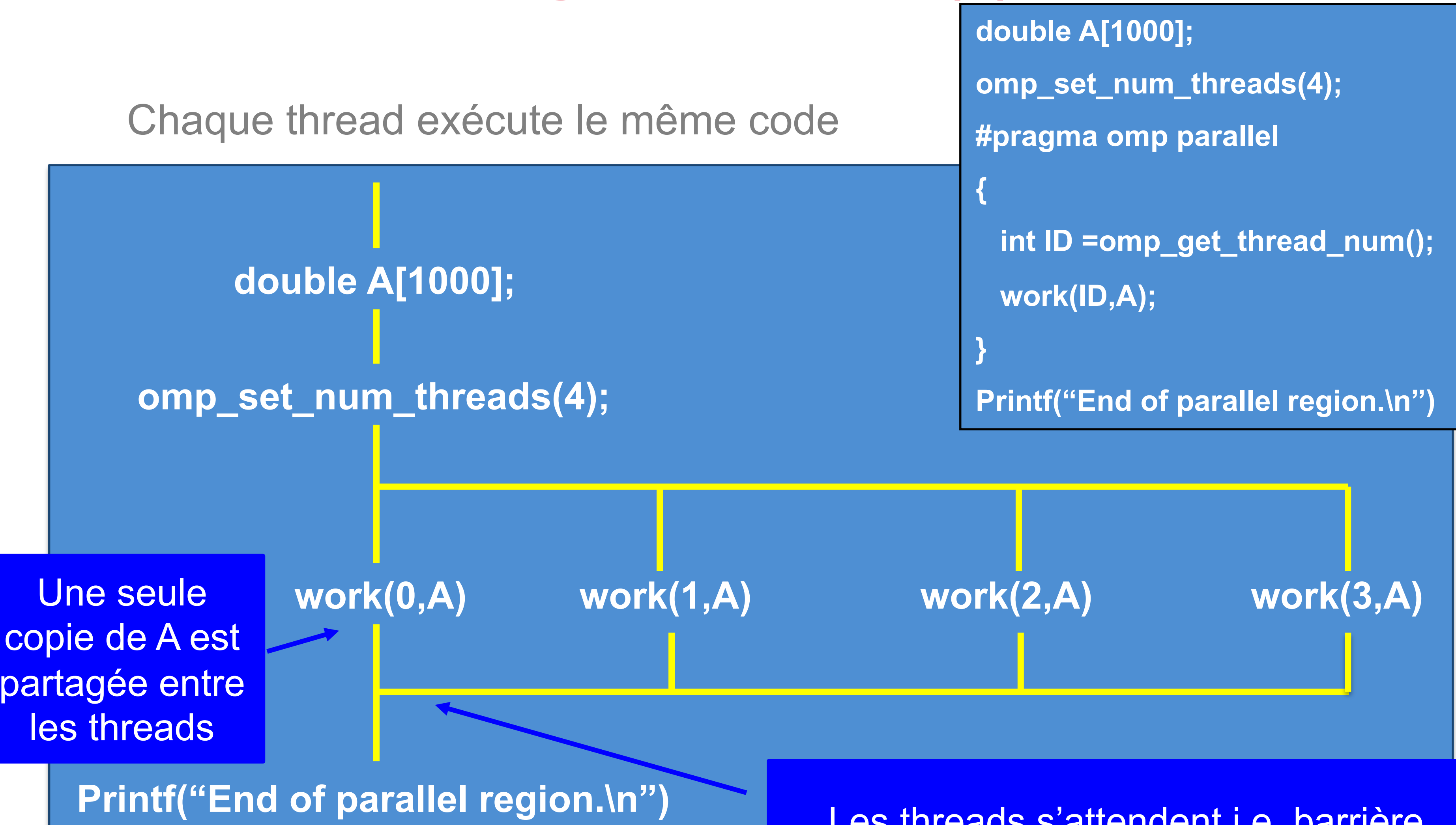
```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID =omp_get_thread_num();
    work(ID,A);
}
```

Chaque thread exécute le code redondant dans le bloc structuré

Chaque thread appelle work(ID,A) pour `ID = 0` to `3`

# OpenMP : Région Parallèle (5)

Chaque thread exécute le même code



# OpenMP: Région Parallèle
# La clause IF

Peut être utilisé pour désactiver la parallélisation dans certains cas

```
#pragma omp parallel if (expression)
```

Exemple : quand la taille d'un paramètre est trop petite

```
int id, N
#pragma omp parallel  private(id) shared(res)  if( n >= 1000)
{
        id = omp_get_thread_num()
        res(id) = big_job(id)
}
```

La région parallèle est exécutée sur N threads seulement si l'expression logique est .TRUE.

# OpenMP: Région Parallèle
# la clause num_threads

Contrôler le nombre de threads utilisés dans une région parallèle

```
#pragma omp num_threads (expression)
```

```
#include <omp.h>
 main() {

#pragma omp parallel num_threads(10)
   {
      ...
      parallel region
     ...
   }
}
```

# OpenMP: Portée des directives

Les directives OpenMP peuvent être sur plusieurs fichiers.

**poo.c**

```
#pragma omp parallel
{
    whoami();
}
```

**bar.c**

```
void whoami() {
    int iam, omp_get_thread_num;
    iam = omp_get_thread_num();
#pragma omp critical
    {
        printf("Hello from %d ", iam);
    }
```

*Directives orphelines*

*Étendue lexicale de la région parallèle*

*Étendue dynamique de la région parallèle*

+

# CONSTRUCTION DE TRAVAIL PARTAGÉ

# Construction de travail partagé

Partager le travail parmi les threads

Pas de création de threads

Pas de barrière implicite en entrée mais une en sortie

Les directives :

- la directive DO ou for
- la directive SECTIONS
- la directive SINGLE
- La directive WORKSHARE

Restrictions

- Chaque région de partage doit être rencontrée par tous les threads ou par aucun
- La séquence de partage et de barrière doit être la même sur chaque thread

# Construction de travail partagé
# La directive for (1)

Partager les itérations d'une boucle à travers l'équipe

```
!$OMP DO [clause(,), clause, ….]
DO I=1, N
        code de la boucle à exécuter
    par chaque thread
      END DO
!$OMP END DO [NOWAIT]
```

```
#pragma omp for [clause(,), clause, ….]
  for (i=1; i<n; i++) {
      code de la boucle à exécuter par
  chaque thread
}
```

master thread
FORK
DO / for loop
team
JOIN
master thread

Parallélisme de données

# Construction de travail partagé
# La directive for (2)

## Clauses

- private, firstprivate, lastprivate
- linear (list:[:linear-step])
- reduction ( operator :  list)
- schedule ( type [,chunck])
- collapse(n)
- ordered [(n)]
- nowait

## Restrictions

- boucle avec un indice entier, A(i),
- contrôle de boucle  (pas de do while)

C++

Il faut utiliser des itérateurs aléatoires pour accéder aux données en temps constant. Sinon il faut utiliser un parallélisme de tâche.

# Construction de travail partagé La directive for (3)

```
#pragma omp parallel
{
     init(a)
#pragma omp for
     for (i=1 ; i < N ; ++i) {
        …
        …
     }
  display(a)
}
```

*Exécution dupliquée*

*Travail partagé : exécute différentes itérations*

*Exécution dupliquée*

# La clause collapse

collapse(n) : indique le nombre de boucles dans un ensemble de boucles imbriquées qui doivent être regroupées en une seule itération.

```
void bar(float *a, int i, int j, int k);

int kl, ku, ks, jl, ju, js, il,
void sub(float *a) {
     int i, j, k;
      #pragma omp for collapse(2) private(i, k, j)
      for (k=kl; k<=ku; k+=ks)
           for (j=jl; j<=ju; j+=js)
                for (i=il; i<=iu; i+=is)
                     bar(a,i,j,k);
}
                              C/C++
```

Fusionne les boucles k et j

# Construction de travail partagé
# La directive DO ou for (un exemple)

```
#pragma omp parallel shared(a,b,c,n) private(i)
{
#pragma omp for schedule(static,chunk) nowait
{
   for (int i=0; i<N; i++)
        C(I) = A(I) + B(I)  ;
}
}
```

Plus de barrière

# OpenMP: différentes approches de a = b+c

Code séquentiel

```
for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
```

Parallélisation manuelle

```
#pragma omp parallel
{
  int id     = omp_get_thread_num();
  int Nthr = omp_get_num_threads();
  int istart = id*N/Nthr, iend = (id+1)*N/Nthr;
  for (int i=istart; i<iend; i++) { a[i]=b[i]+c[i]; }
}
```

Parallélisation automatique

```
#pragma omp parallel
#pragma omp for schedule(static)
{
  for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
}
```

# Problème avec les boucles

## Equilibrage de charge

- Si toutes les itérations s’exécutent à la même vitesse, les processeurs sont utilisés de manière optimale ;
- Si certaines itérations sont plus rapides que d'autres, certains processeurs seront plus lents pour traiter leurs itérations, réduisant ainsi l'accélération ;
- Si on ne connait pas à priori la répartition du travail, il peut être nécessaire de redistribuer dynamiquement la charge.

## Granularité

- La création de threads et la synchronisation prennent du temps ;
- Affectation de travail pour les threads peut prendre plus de temps que l'exécution elle-même! ;
- Besoin de fusionner le travail (grain grossier)  pour recouvrir le surcout des threads.

Compromis entre l'équilibrage de charge et de la granularité!

# L'ordonnancement du travail
# La clause SCHEDULE

Format : **schedule ([modifier:,] type [,chunk])**

**type** est à choisir parmi

- static ( chunk)
- dynamic (chunk)
- guided ( chunk)
- auto c'est le compilateur ou le runtime qui décide
- runtime

  Décidé à l'exécution et spécifié par une variable OMP_SCHEDULE

  setenv OMP_SCHEDULE STATIC,100

Si pas de cause, l'ordonnancement **dépend de l'implémentation** !!!

# L’ordonnancement du travail
# La clause SCHEDULE

Format : **schedule ([modifier:,] type [,chunk_size])**

**modifier** est à choisir parmi

- simd: chunk_size doit être un multiple de la largeur du simd
- monotonic: Si un thread a exécuté l'itération i, alors le thread doit exécuter des itérations plus grandes que i par la suite.
- non-monotonic: Les itérations sont exécutés dans n’importe quelle ordre. Uniquement pour guided et dynamique

# L’ordonnancement du travail
# Le type STATIC

0 1 2 0
chunk = 150

0 1 2
chunk = 250

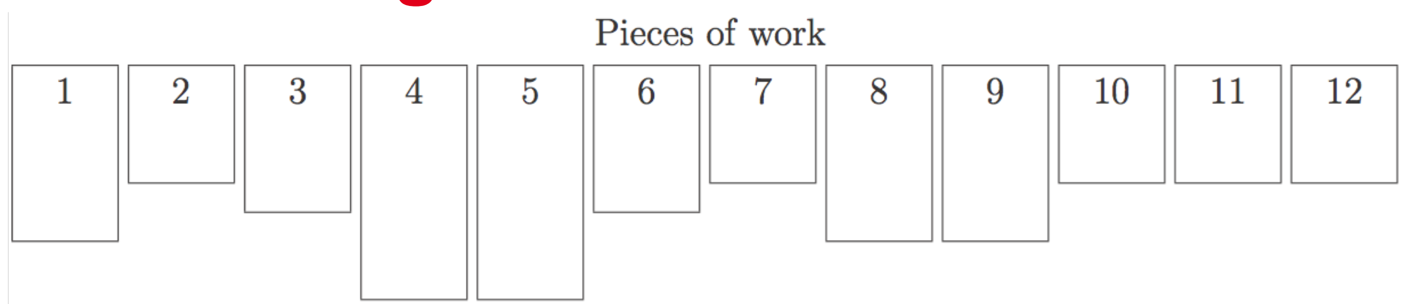
0 1
chunk = 300

0 1 2
default

0 50 100 150 200 250 300 350 400 450 500 550 600

Exemple :
- 600 itérations
- SCHEDULE(STATIC, chunk)

# Cas irrégulier



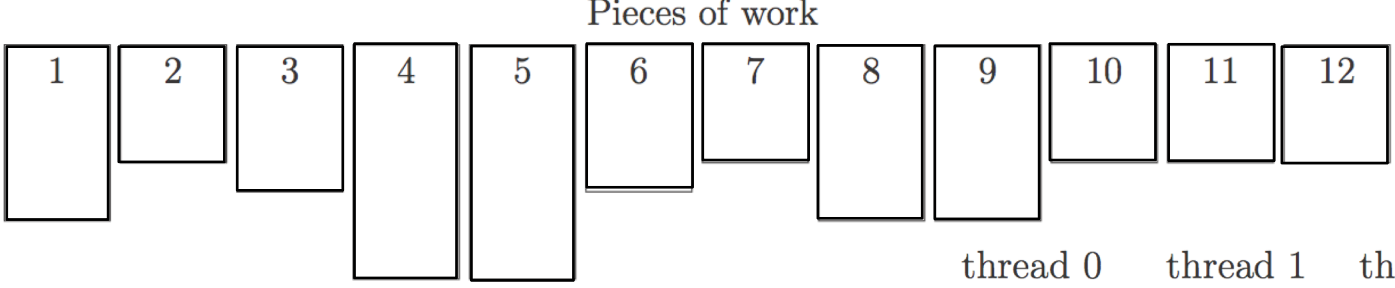

Thread 1 | Thread 2 | Thread 3
1, 2, 3, 4 | 5, 6, 7, 8 | 9, 10, 11, 12

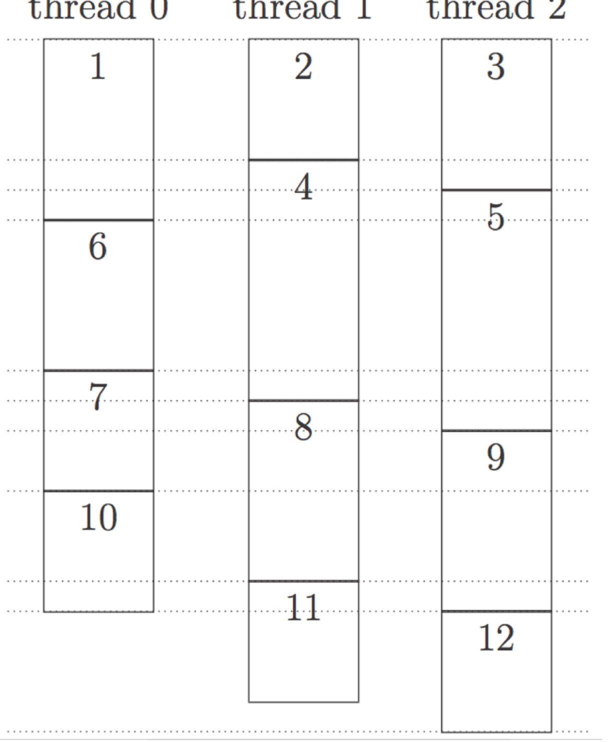
EXECUTION

Thread 1 | Thread 2 | Thread 3 | Thread 4
1, 2, 3 | 4, 5, 6 | 7, 8, 9 | 10, 11, 12

EXECUTION

# L'ordonnancement du travail
# Le type DYNAMIC




Affectation dynamique des itérations de taille chunck.

Défaut chunck = 1

**Schedule(dynamic, chunk)**

# L'ordonnancement du travail
# Le type GUIDED

Approche dynamique mais avec un nombre d'itérations variable

Nombre d'itérations = n et nombre de threads = p

Chunk size = k = nombre minimal d'itérations traité par un thread

Alors le nombre d'itérations pour le premier thread est

q= ceil(n/(a*p)) (entier supérieur)

Puis on itère avec n = max( n – q , a*k*p) avec a = 1 ou 2 (implémentation)

Exemple :

800 itérations éclatées sur 2 threads k = 80

**schedule(guided, 80)**

a = 2 → 800 /(2*2) = 200, 150, 113, 85, 80 (63), 80, 80, 12

a = 1 → 800 /2 = 400, 200, 100, 80, 20

# Comment choisir le type de l’ordonnanceur ?

| Ordonnancement | Quand l’utiliser |
| --- | --- |
| static | Travail par itération est prédictible et similaire |
| dynamic | Travail par itération est imprévisible et très variable |
| guided | Cas spécial du cas dynamique pour réduire le surcoût. |
| auto | Aucune idée |

# Construction de travail partagé
# La directive sections

Zone non itérative, chaque section est exécutée par un thread

```
!$OMP SECTIONS clause[[,] clause…]
!$OMP SECTION
      < code bloc 1>
!$OMP SECTION
      < code bloc 2>
!$OMP SECTION
         . . .
!$OMP END SECTIONS [NOWAIT]
```

Clauses :

- private
- firstprivate
- Lastprivate
- reduction

```
#pragma omp sections clause[[,] clause…]
{
#pragma omp section
       < code bloc 1>
#pragma omp section
      < code bloc 2>
#pragma omp section
          . . .
}
```

Clauses :

- private
- firstprivate
- lastprivate
- reduction
- nowait

master thread

FORK

SECTIONS

team

JOIN

master thread

Parallélisme procédurale

# Construction de travail partagé
# La directive sections (un exemple)

```
!$OMP PARALLEL
!$OMP SECTIONS

!$OMP SECTION
        CALL XAXIS()
!$OMP SECTION
        CALL YAXIS()
!$OMP SECTION
        CALL ZAXIS()

!$OMP END SECTIONS NOWAIT

!$OMP END PARALLEL
```

```
#pragma omp parallel default(none)\
      shared(n,a,b,c,d) private(i)
{
#pragma omp sections nowait
  {
#pragma omp section
      for (i=0; i<n-1; i++)
        b[i] = (a[i] + a[i+1])/2;
#pragma omp section
      for (i=0; i<n; i++)
        d[i] = 1.0/c[i];
   } /*-- End of sections --*/
} /*-- End of parallel region --*/
```

# La directive MASTER

Détermine un section où uniquement la thread maître exécute le code

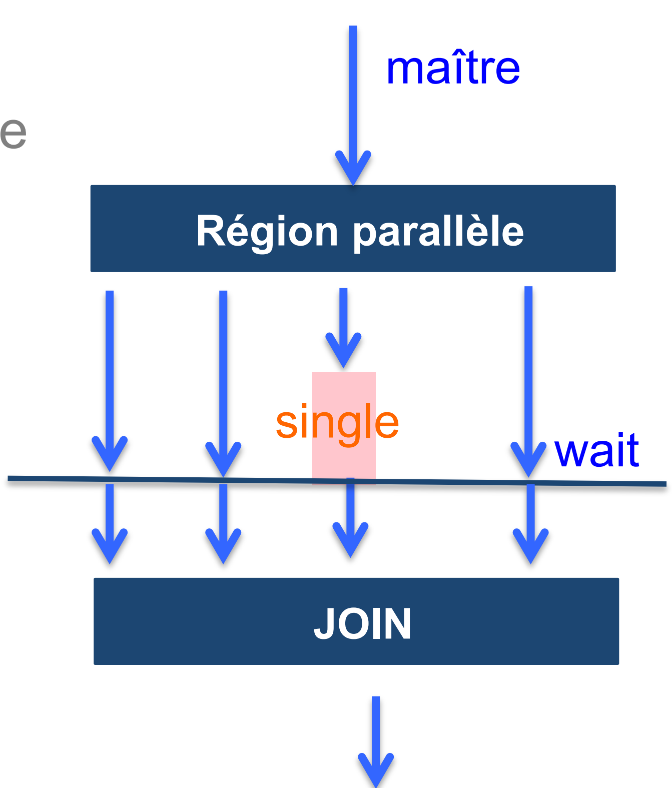
- Le reste de l'équipe saute la section et continue l'exécution après la fin de la section maître
- **Pas de barrière implicite** à l'entrée et à la sortie de la section

```
#pragma omp master
    { bloc structuré }
```

```
#pragma omp parallel {
        …. ;
        #pragma omp master
         { printf(" Hello \n"); }

        …..
}
```

# La directive SINGLE

Un seul thread du groupe va exécuter le code




```
#pragma omp single clause[[,] clause…]
    { bloc structuré }
```

Clauses :

- private, firstprivate
- copyprivate, nowait

# Construction de travail partagé
# La directive SINGLE (un exemple)

```
#pragma omp parallel shared(a,b,c,n) private(i)
 {
#pragma omp single
     output(…)  ;

}
```

Restrictions

- La clause copyprivate ne peut pas être utilisée avec nowait ;
- Une seule clause nowait.

# Construction de travail partagé
# Restriction

Les directives doivent se trouver dans une région parallèle

Ces directives doivent être rencontrées

- par tous les threads du groupe
- dans le même ordre

# Combiner région parallèle et travail partagé

On peut combiner les directives région parallèle et travail partagé :

- Directive do parallèle

```
#pragma omp parallel for [ … ]
boucle for
```

- Directive sections parallèle

```
#pragma omp parallel sections […]
{
    #pragma omp section
      Bloc 1
    #pragma omp section
      Bloc 2
    ...
}
```

# ATTRIBUTS DE DONNÉES

# Les différents statuts

PRIVATE (list)

SHARED (list)

DEFAULT( SHARED | NONE)

FIRSTPRIVATE (list)

initialise chaque copie locale par la valeur originale

LASTPRIVATE (list)

le dernier thread met à jour la variable

THREADPRIVATE (list)

# Un exemple (1)

```
void saxpy(z, a, x, y, n){
     int   i, n
     float z[n], a, x[n], y

#pragma omp for
     for(  i = 0 ; i<  n ; ++i) {
          z[i] = a * x[i] + y
     }

}
```

# Un exemple (2)

Mémoire partagée
Globale

| z | a | x | y | n | i |
|---|---|---|---|---|---|

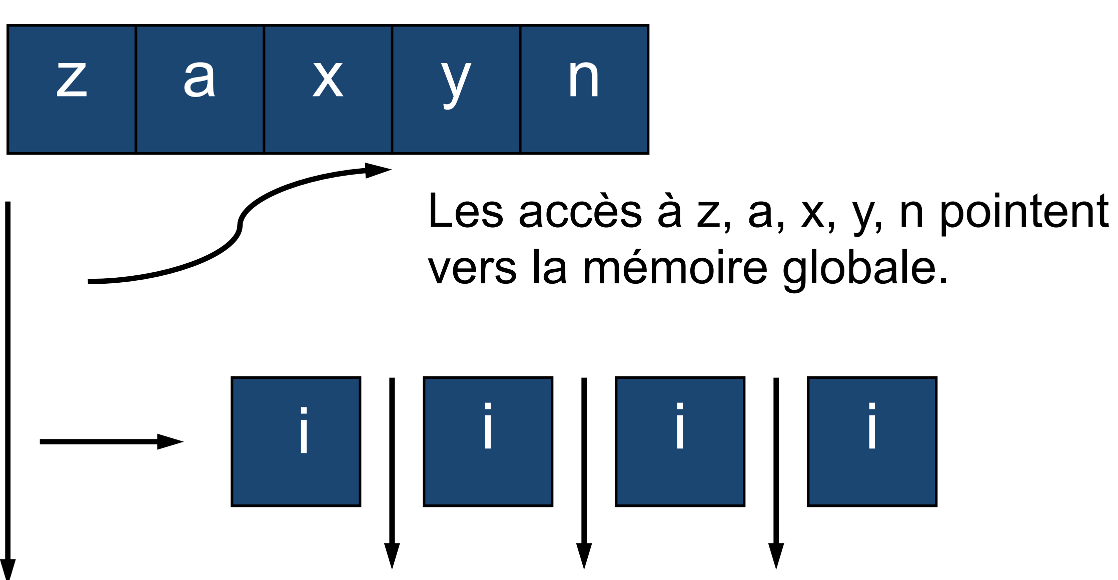
**Exécution séquentielle.**

Toutes les données pointent sur la mémoire globale

Mémoire partagée
Globale

**Exécution parallèle**

Chaque thread possède une copie privée de i
Les accès à i vont vers la copie privée



# Un exemple (3)

## Division du travail entre les threads

Mémoire partagée globale.

| Z(1) | | Z(10) | Z(11) | | Z(20) | Z(21) | | Z(30) | Z(31) | | Z(40) | a |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| X(1) | | X(10) | X(11) | | X(20) | X(21) | | X(30) | X(31) | | X(40) | y |
| | | | | | | | | | | | | n |

```
void saxpy(z, a, x, y, n){
      int   i, n
      float z[n], a, x[n], y
#pragma omp for
      for(  i = 0 ; i<  n ; ++i)
      {
          z[i] = a * x[i] + y
      }
}
```

Mémoire locale

| i = 1, 10 | i = 11, 20 | i = 21, 30 | i = 31, 40 |
|---|---|---|---|

**n = 40, 4 threads**

# La clause shared

shared (var)

Tous les threads accèdent à la même zone mémoire de la variable var

- Attention au conflit d'écriture, mise à jour, …

Uniquement possible avec la directive de région parallèle ou des tâches

```
Subroutine saxpy(z, a, x, y, n)
integer i, n
real z(n), a, x(n), y
!$omp parallel do shared (z, a, x, y) private(i)
  do i = 1, n
      z(i) = a * x(i) + y
  end do
end
```

# La clause private

## private (VAR)

Création d'une copie locale de VAR dans chaque thread :

- La valeur n'est pas initialisée
- Pas de lien entre le stockage de la copie privée et de la variable originale
- En sortie dépend de la version.

```
     is = 0
#pragma omp parallel for private(is)
     for(int j = 0 ; j <1000 ++j}{
        is  += j ;
       ….
     }
printf( "is: %d", is);
```

IS n'est pas initialisé

IS = 0

# La clause firstprivate

firstprivate (var1, var2, …)

firstprivate est un cas spécial du statut private

- Création d'une copie locale dans chaque thread
- Initialisation de chaque copie par la valeur de la variable provenant de la thread maître.

```
void useless() {
    int tmp = 0;
#pragma omp for firstprivate(tmp)
    for (int j = 0; j < 1000; ++j)
        tmp += j;
    printf("%d\n", tmp);
}
```

Chaque thread a une copie de tmp et est initialisée à 0

tmp = 0

# La clause lastprivate

lastprivate (var1, var2, )

lastprivate est un cas spécial du statut private

- En sortie, affecte la valeur de la dernière itération ou section (séquentielle)
- Si NOWAIT; la variable est indéfinie tant qu'une barrière de synchronisation n’a pas été réalisée.

```
void useless() {
	int tmp = 0;
#pragma omp for lastprivate(tmp)
	for (int j = 0; j < 1000; ++j)
		tmp += j;
	printf(“%d\n”, tmp);
}
```

Chaque thread a une copie de tmp

tmp = valeur pour j = 999

# La clause default

default (shared | none) C/C++

Précise le statut des variables dans la portée.

**none**

- Pas de statut par défaut
- Il faut préciser le statut de toutes les variables.

**shared**

- Toutes les variables sont partagées.
- C'est le statut par défaut en l'absence de la clause none.

# Un test

Considérons l'exemple suivant

Variables : A,B et C =1

```
#pragma omp for private(B) firstprivate(C)
```

Q1 A,B et C sont-elles locales à chaque thread ou partagées dans une région parallèle ?

Q2 Préciser les valeurs initiales et finales (après la zone parallèle)

Dans la région parallèle

- A est partagée A = 1
- B et C sont privées
    - La Valeur initiale de B n'est pas définie
    - La Valeur initiale de C est 1

En dehors de la région parallèle

Les valeurs de B et de C sont définies dans la région mais pas en dehors de la région parallèle

# La clause threadprivate

#pragma omp threadprivate(list)

Directive déclarative

Duplique la variable, chaque thread possède sa propre copié

Différent de les considérer comme PRIVATE

- Avec PRIVATE les variables globales sont masquées.
- THREADPRIVATE préserve la portée globale à l'intérieur de chaque thread

Les variables threadprivate peuvent être initialisées en utilisant la clause **copyin**.

Restrictions

- Le nombre de threads ne doit pas varier (omp_dynamic = false)
- La directive doit suivre la déclaration.

# La clause Threadprivate (exemple)

Créer un compteur privé dans chaque thread.

```
int counter = 0;
#pragma omp threadprivate(counter)

int increment_counter()
{
    counter++;
    return (counter);
}
```

```
#include <omp.h>
int a, b, i, tid;
float x;
#pragma omp threadprivate(a, x)

main () {
      printf("1st Parallel Region:\n");
#pragma omp parallel private(b, tid)
    {  tid = omp_get_thread_num();
    a = tid ;  b = tid;   x = 1.1 * tid +1.0;
     printf("Thread %d: a,b,x= %d %d %f\n",tid,a,b,x);
     } /* end of parallel section */
  printf("************************************\n");
  printf("Master thread doing serial work here\n");
  printf("************************************\n");
  printf("2nd Parallel Region:\n");
#pragma omp parallel private(tid)
  { tid = omp_get_thread_num();
    printf("Thread %d: a,b,x= %d %d %f\n",tid,a,b,x);
  } /* end of parallel section */
}
```

```
% export OMP_NUM_THREADS=4
% ./a.out
1st Parallel Region:
Thread 0: a,b,x= 0 0 1.000000
Thread 1: a,b,x= 1 1 2.100000
Thread 2: a,b,x= 2 2 3.200000
Thread 3: a,b,x= 3 3 4.300000
************************************
Master thread doing serial work here
************************************
2nd Parallel Region:
Thread 0: a,b,x= 0 0 1.000000
Thread 3: a,b,x= 3 0 4.300000
Thread 1: a,b,x= 1 0 2.100000
Thread 2: a,b,x= 2 0 3.200000
```

# La clause copyin

## copyin (list)

donne un moyen d'assigner la même valeur **aux variables threadprivate** pour tous les threads.

La variable du thread master est utilisée comme valeur à copier

Les commons privés (thread) sont initialisés par la valeur du thread maître.

# La clause copyprivate

## copyprivate(list)

Pour diffuser une variable privée dans une variable globale ou un pointer

- list ne doit pas avoir le statut de PRIVATE, FIRSTPRIVATE
- uniquement avec la directive SINGLE (1 seul thread)

```
float x, y;
#pragma omp threadprivate(x, y)
void init(float a, float b)
{
   #pragma omp single copyprivate(a,b,x,y)
   {
       get_values(a,b,x,y);
    }
}
```

# Les clauses copyin et copyprivate

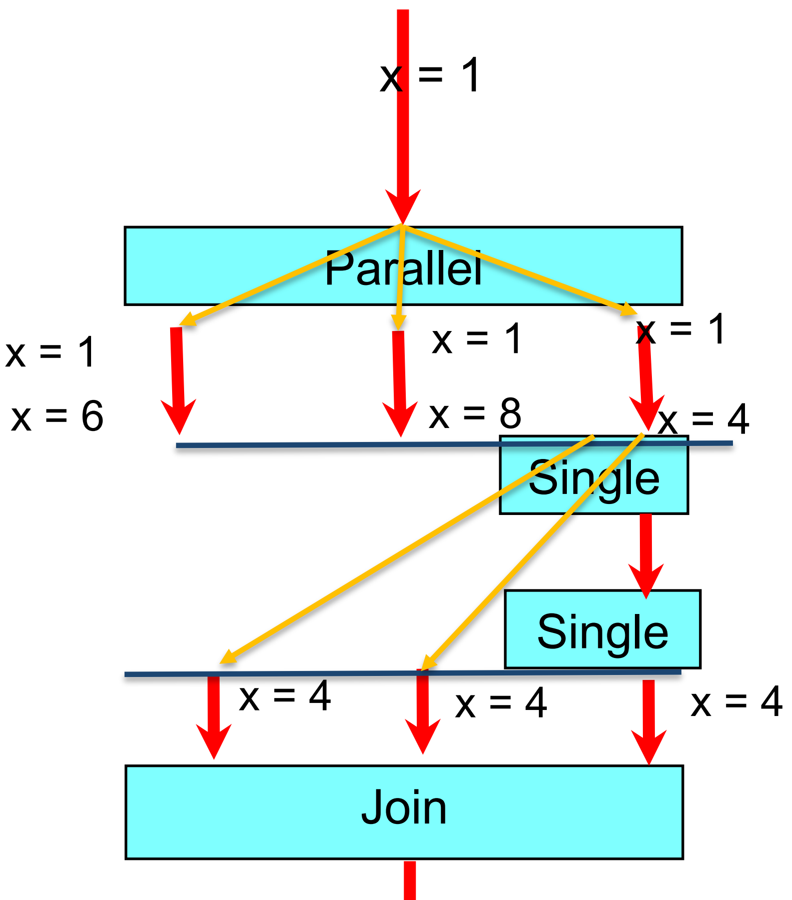
Int x = 1 ;

#pragma omp threadprivate(x)

#pragma parallel copyin(x)

Operations sur x

#pragma single copyprivate(x)



# La clause reduction

Une autre clause qui affecte comment une variable est partagée

**reduction (op : list)**

Les variables dans “list” doivent être partagées dans le région parallèle.
Les opérateurs possibles sont :

C/C++ : +, *, -, &, |, ^, &&, max, min

Dans une région parallèle :

- Une copie locale des variables de la liste est faite et initialisée en fonction de l’opérateur op de la réduction. (ex.. 0 for “+”).
- Les copies locales sont réduites en une seule valeur et combinée avec la valeur de la variable globale (originale).

# La clause reduction

Restriction :

- List peut contenir des sections de  tableaux
- Les copies locales sont réduites en une seule valeur et combinée avec la valeur de la variable globale (originale).

Exemple:

```
#pragma omp for reduction(+: A, Y) reduction(.OR.: AM)
```

# Résumé des clauses

| Clause | Directive | | | | | |
|---|---|---|---|---|---|---|
| | PARALLEL | DO/for | SECTIONS | SINGLE | PARALLEL DO/for | PARALLEL SECTIONS |
| IF | ● | | | | ● | ● |
| PRIVATE | ● | ● | ● | ● | ● | ● |
| SHARED | ● | ● | | | ● | ● |
| DEFAULT | ● | | | | ● | ● |
| FIRSTPRIVATE | ● | ● | ● | ● | ● | ● |
| LASTPRIVATE | | ● | ● | | ● | ● |
| REDUCTION | ● | ● | ● | | ● | ● |
| COPYIN | ● | | | | ● | ● |
| COPYPRIVATE | | | | ● | | |
| SCHEDULE | | ● | | | ● | |
| ORDERED | | ● | | | ● | |
| NOWAIT | | ● | ● | ● | | |

# L' API OPENMP

# Les variables d'environnement

Fixe le nombre de threads à utiliser

**OMP_NUM_THREADS *int_literal***

Autorise d'utiliser un nombre de thread différents dans chaque région ?

**OMP_DYNAMIC TRUE || FALSE**

Contrôle comment OpenMP ordonnance pour la clause schedule (RUNTIME) le travail partagé.

**OMP_SCHEDULE "schedule[, chunk_size]"**

**setenv OMP_SCHEDULE "guided,4**

Précise si l'on souhaite faire du parallélisme emboîté avec une nouvelle équipe de thread ou non

**OMP_NESTED TRUE || FALSE**

**OMP_MAX_ACTIVE_LEVELS i*nt_literal*** contrôle le nombre maximal de parallélisme emboitée dans une région parallèle. La valeur est un entier positif.

# Les variables d'environnement

**OMP_PROC_BIND** : permet de fixer un thread sur le processeur.

OMP_PROC_BIND   true/false

**OMP_STACKSIZE** : contrôle la taille de la pile pour les threads créés (sauf le master)

OMP_STACKSIZE n [B,K,M,G]

**OMP_WAIT_POLICY** : permet de préciser la politique d'attente des threads

OMP_WAIT_POLICY   active/passive

**OMP_THREAD_LIMIT** : Fixe le nombre de threads utilisé dans le programme

**OMP_CANCELATION :** true, false spécifie l'effet du constructeur cancel

# Les variables d'environnement

**OMP_ DISPLAY_ENV=TRUE | FALSE | VERBOSE**

Affiche les informations du runtime que l'utilisateur peut changer. Verbose donne aussi les informations sur le runtime (vendeur) qui peuvent être modifiées

```
export OMP_DISPLAY_ENV=TRUE
./pgm_omp
```

```
OPENMP DISPLAY ENVIRONMENT BEGIN
  _OPENMP = '201511'
  OMP_DYNAMIC = 'FALSE'
  OMP_NESTED = 'FALSE'
  OMP_NUM_THREADS = '4'
  OMP_SCHEDULE = 'DYNAMIC'
  OMP_PROC_BIND = 'FALSE'
  OMP_PLACES = ''
  OMP_STACKSIZE = '2097152'
  OMP_WAIT_POLICY = 'PASSIVE'
  OMP_THREAD_LIMIT = '4294967295'
  OMP_MAX_ACTIVE_LEVELS = '2147483647
  OMP_CANCELLATION = 'FALSE'
  OMP_DEFAULT_DEVICE = '0'
  OMP_MAX_TASK_PRIORITY = '0'
…
OPENMP DISPLAY ENVIRONMENT END
```

# L'API OpenMP
# Les fonctions de la bibliothèque

1. OMP_SET_NUM_THREADS
2. OMP_GET_NUM_THREADS
3. OMP_GET_MAX_THREADS
4. OMP_GET_THREAD_NUM
5. OMP_GET_THREAD_LIMIT
6. OMP_GET_NUM_PROCS
7. OMP_IN_PARALLEL
8. OMP_SET_DYNAMIC
9. OMP_GET_DYNAMIC
10. OMP_SET_NESTED
11. OMP_GET_NESTED
12. OMP_SET_SCHEDULE
13. OMP_GET_SCHEDULE
14. OMP_SET_MAX_ACTIVE_LEVELS
15. OMP_GET_MAX_ACTIVE_LEVELS
16. OMP_GET_LEVEL
17. OMP_GET_ANCESTOR_THREAD_NUM
18. OMP_GET_TEAM_SIZE
19. OMP_GET_ACTIVE_LEVEL
20. OMP_INIT_LOCK
21. OMP_DESTROY_LOCK
22. OMP_SET_LOCK
23. OMP_UNSET_LOCK
24. OMP_TEST_LOCK
25. OMP_INIT_NEST_LOCK
26. OMP_DESTROY_NEST_LOCK
27. OMP_SET_NEST_LOCK
28. OMP_UNSET_NEST_LOCK
29. OMP_TEST_NEST_LOCK
30. OMP_GET_WTIME
31. OMP_GET_WTICK

# L'API OpenMP
# Les fonctions de la bibliothèque

Fonctions de l'API

- Tester/ modifier le nombre de threads
  omp_set_num_threads(), omp_get_num_threads(),
  omp_get_thread_num(), omp_get_max_threads()

- Modifie le mode dynamique i.e le nombre de threads peut varier entre deux constructions parallèles
  omp_set_dynamic(), omp_get_dynamic()

- Modifie le parallélisme emboité
  omp_set_nested(), omp_get_nested(),

- Sommes nous dans une région parallèle ?
  omp_in_parallel()

- Combien de processeurs dans le système ?
  omp_num_procs()

# L'API OpenMP
# Les fonctions de la bibliothèque

Fonctions sur les verrous

omp_init_lock(), omp_init_nest_lock(),
omp_destroy_lock(), omp_destroy_nest_lock(),
omp_set_lock(), omp_set_nest_lock(),
omp_unset_lock(), omp_unset_nest_lock(),
omp_test_lock(), omp_test_nest_lock()

Fonction pour mesurer le temps

omp_get_wtime(), omp_get_wtick()

```
double start;
double end;
start = omp_get_wtime();
... work to be timed ...
end = omp_get_wtime();
printf_s("Work took %f sec. time.\n", end-start);
```

# SYNCHRONISATION

# OpenMP: Synchronization

Différentes directives permettent de synchroniser les

Haut niveau

- Barrier
- critical
- atomic
- ordered

Bas niveau

- flush
- lock

# La directive barrier

**!$omp barrier** **#pragma omp barrier**

Le thread attend jusqu'à ce que toute l'équipe arrive sur la directive

```
#pragma omp parallel shared (A, B, C) private(id)
{
      id=omp_get_thread_num();
      A[id] = big_calc1(id);
#pragma omp barrier
#pragma omp for
      for(i=0;i<N;i++){C[i]=big_calc3(I,A);}
#pragma omp for nowait
      for(i=0;i<N;i++){ B[i]=big_calc2(C,  i); }

      A[id] = big_calc3(id);
}
```

Barrière implicite à la fin de bloc de la directive for

Pas de barrière implicite à cause de **nowait**

Barrière implicite à la fin de la région parallèle

# La directive critical

```
!$omp critical [(nom)]
      bloc d'instructions
!$omp end critical [(nom)]
```

```
#pragma omp critical [(name)]
      {bloc structuré}
```

Empêcher l'accès simultané pas les threads à un bloc d'instruction

Un thread à la fois dans le bloc d'instructions parmi toutes les threads du programme (pas de l'équipe!)

**La section critique**

- On peut la nommer
  - Le nom est une entité globale
  - Ne doit pas entrer en conflit avec le nom d'une procédure, common, …
- Un thread à la fois dans la section

# La directive critical (exemple)

Accumulation dans une boucle

```
!$omp parallel do shared(A) private(ALOCAL)
      ……
!$OMP CRITICAL (left)
      A(index(i)) = A(index(i)) + Alocal
!$OMP END CRITICAL (left)
      ……
!$omp end parallel
```

```
#pragma omp for shared(A) private(Alocal)
 for (…) {
      ……
#pragma omp critical (left)
      A[index[i]] =  A[index[i]] + Alocal
……
}
```

# La directive atomic

La directive assure qu'une variable partagée est lue et modifiée en mémoire par un seul thread à la fois

Ne s'applique qu'à l'instruction suivant la directive

```
#pragma omp atomic [read | write | update Í capture]
        {bloc structuré}
```

Clauses :

read v = x

write x = v

update (défaut) x = x binop operation (++x, x--, …)

capture fait un atomic update de x + capture la valeur d'origine ou finale

# La directive atomic (exemple)

```
#pragma omp parallel
{
        double tmp, B;
        B     = DOIT();
        tmp = other_work(B);

#pragma omp atomic update
         x  +=  tmp ;

}
```

fournit l’exclusion mutuelle mais s'applique uniquement à la mise à jour d'un emplacement mémoire

# La directive ordered

Les itérations d'une boucle seront exécutées dans l'ordre des itérations. Le constructeur va séquentialiser l'exécution du bloc

```
#pragma omp ordered
   {  bloc  }
```

Restriction

- Uniquement dans un bloc for
- La directive for doit aussi avoir la clause ordered

```
#pragma omp parallel private (tmp)
#pragma omp for ordered
        for (I=0;I<N;I++){
                tmp = NEAT_STUFF(I);
#pragma ordered
                res += consum(tmp);
        }
```

# La directive flush

Permet à un thread de construire une vue cohérente de la mémoire

```
!$omp flush (list)
```

```
#pragma omp flush (list)
```

Après cet appel

- Toutes les opérations mémoires sont terminées
- Variables dans les registres, buffers d'écriture doivent être mis à jour en mémoire ;

Autorise une mécanisme de synchronisation point à point

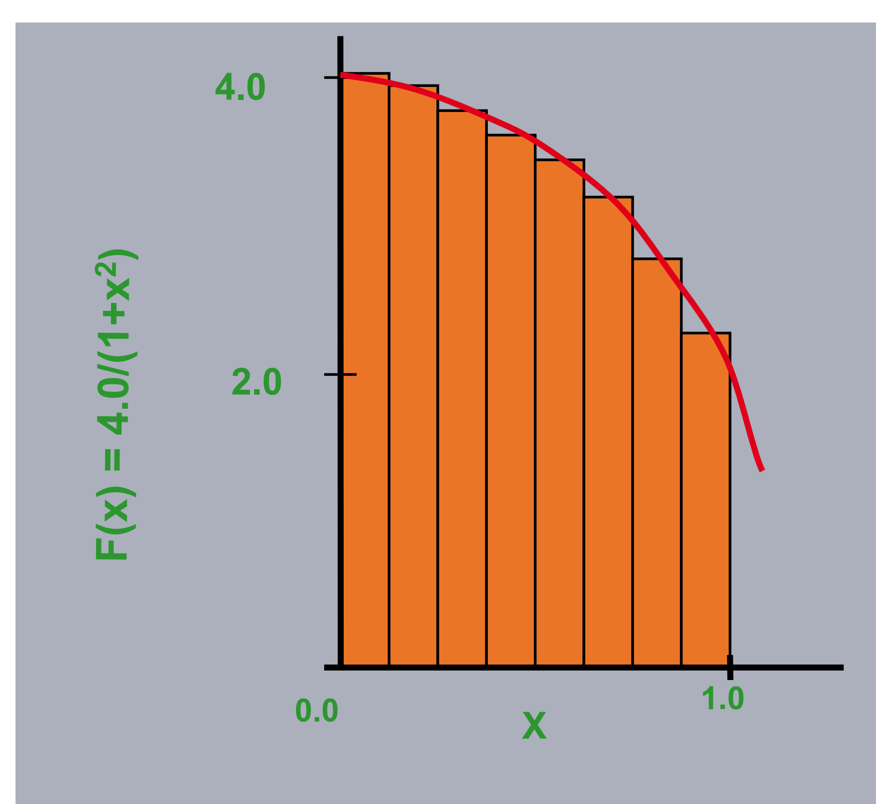
La dernière valeur de la variable est visible pour tous les threads

# La directive flush (exemple)

Permet de synchroniser deux threads, la variable ISYNC doit être partagée.

```
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(ISYNC)
IAM      = OMP_GET_THREAD_NUM() ;  ISYNC(IAM) = 0
NEIGH = GET_NEIGHBOR (IAM)
!$OMP BARRIER
CALL WORK()
C I am done with my work, synchronize with my neighbor
ISYNC(IAM) = 1
!$OMP FLUSH(ISYNC)
C Wait until neighbor is done
DO WHILE ( ISYNC(NEIGH) .EQ. 0)
!$OMP FLUSH(ISYNC)
END DO
!$OMP END PARALLEL
```

# Exemple : calcul de pi (1)




On sait que

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

On peut approcher l'intégrale par la somme des rectangles :

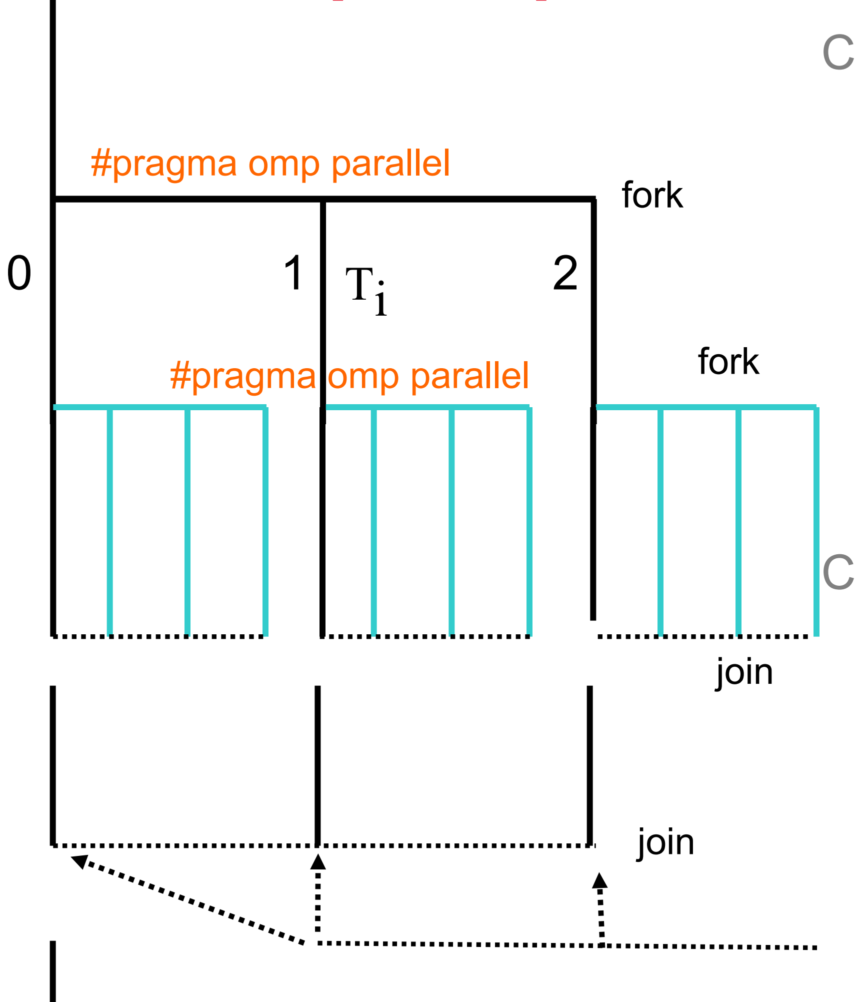
$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Chaque rectangle à une largeur de $\Delta x$ et une hauteur de $F(x_i)$ au milieu de l'intervalle i.

Exercice

# LE PARALLÉLISME EMBOÎTÉ

# Parallélisme emboîté
# Le principe




Chaque thread rencontre une région parallèle

- Autoriser le parallélisme emboîté
  setenv OMP_NESTED TRUE
  Utiliser **omp_set_nested()**
- Sinon la partie est sérialisée (1 thread)

Création des nouveaux threads

- thread $T_i$ devient master
- master + slaves = team
- à la fin les slaves sont en mode sleep ou spin
  - Dépend de **OMP_WAIT_POLICY**

# Parallélisme emboîté
# Les restrictions

Attention il faut imbriquer des régions parallèles et pas du partage de travail.

```
#pragma omp parallel
{
 #pragma omp parallel
          {  bloc 2}
}
```

```
#pragma omp parallel
#pragma omp for
{
 #pragma omp for
          {  bloc 2}
}
```

Directives interdites :

- barrier, master, single, ordered
- critical avec le même nom

# Exemples (1)

```
#include <omp.h>

int main() {
  int rang = -1, rang2 = -1 ;
  omp_set_nested(1);
#pragma omp parallel default(none) num_threads(2) private(rang,rang2)
  {
   rang = omp_get_thread_num() ;
#pragma omp parallel default(none) num_threads(3) private(rang2) ,shared(rang)
   {
    rang2 = omp_get_thread_num() ;
    printf("Mon rang dans region 1 est : %d  dans la region 2 est %d \n",rang, rang2) ;
   }
  }
  return 0 ;
}
```

# Exemples (2)

```
$ ./nested
Mon rang dans la region 1 est : 0 et  dans la region 2 est 0
Mon rang dans la region 1 est : 0 et  dans la region 2 est 1
Mon rang dans la region 1 est : 0 et  dans la region 2 est 2
Mon rang dans la region 1 est : 1 et  dans la region 2 est 0
Mon rang dans la region 1 est : 1 et  dans la region 2 est 1
Mon rang dans la region 1 est : 1 et  dans la region 2 est 2
```

# Exemples (3)

```
#pragma omp parallel default(none) num_threads(2) private(rang)
 {
   rang = omp_get_thread_num() ;
    printf("Mon rang dans region 1 est : %d \n",rang) ;
#pragma omp parallel default(none) num_threads(3) private(rang)
   {
     rang = omp_get_thread_num() ;
     printf("Mon rang dans la region 2 est %d \n",rang) ;
   }
 }
```

# Exemples (4)

$ ./nested_1

Mon rang dans region 1 est : 0
Mon rang dans la region 2 est 0
Mon rang dans region 1 est : 1
Mon rang dans la region 2 est 1
Mon rang dans la region 2 est 2
Mon rang dans la region 2 est 0
Mon rang dans la region 2 est 1
Mon rang dans la region 2 est 2

# Exemples (5)

```
#pragma omp parallel default(shared)
{
 #pragma omp for
      for (i=0; i<n; i++) {
 #pragma omp parallel shared(i, n)
             {
#pragma omp for
              for (j=0; j<n; j++)
                   work(i, j);
             }
      }
 }
```

# Quelques méthodes utiles (1)

Fixer le nombre de threads par niveau

- Variable d'environnement : OMP_NUM_THREADS
- fonction: omp_set_num_threads() dans une région parallèle
- Clause : num_threads(10)

Fixer/obtenir le nombre de threads dans le programme

- Variable d'environnement : OMP_THREAD_LIMIT
- fonction: omp_get_thread_imit() pour connaître le nombre de threads disponible

# Quelques méthodes utiles (2)

Set/Get le nombre maximal de niveau de régions parallèles emboitées

- Variable d'environnement : OMP_MAX_ACTIVE_LEVELS
- Fonctions omp_set_max_active_levels(), omp_get_max_active_levels()

Fonctions de la bibliothèques pour déterminer

- La profondeur : omp_get_active_level() (1, 2, …, $level_{max}$)
- Id du père : omp_get_ancestor_thread_num(level)
- La taille de l'équipe d'un niveau level: omp_get_team_size(level)

# CONSTRUCTION DE TÂCHES

# Les tâches OpenMP

Limitations

- OpenMP doit tout connaitre à l'avance

    longueur d'une boucle, nombre de sections parallèle, …

Difficile pour les problèmes irréguliers

- Boucles non bornées  (boucle while) ;
- Algorithmes récursifs ;
- Schémas producteurs/consommateurs

La solution : les tâches (OpenMP 3.0, 4.0 ….).

MAIS

Tout doit être connu à la compilation (directives) !!

# Les tâches OpenMP

Les tâches OpenMP sont des unités de travail indépendantes qui s'exécutent en parallèle

Une tâche est constituée
- D'un code à exécuter ;
- D'un environnement de données initialisées à la création ;
- De variables de contrôle interne (ICV).

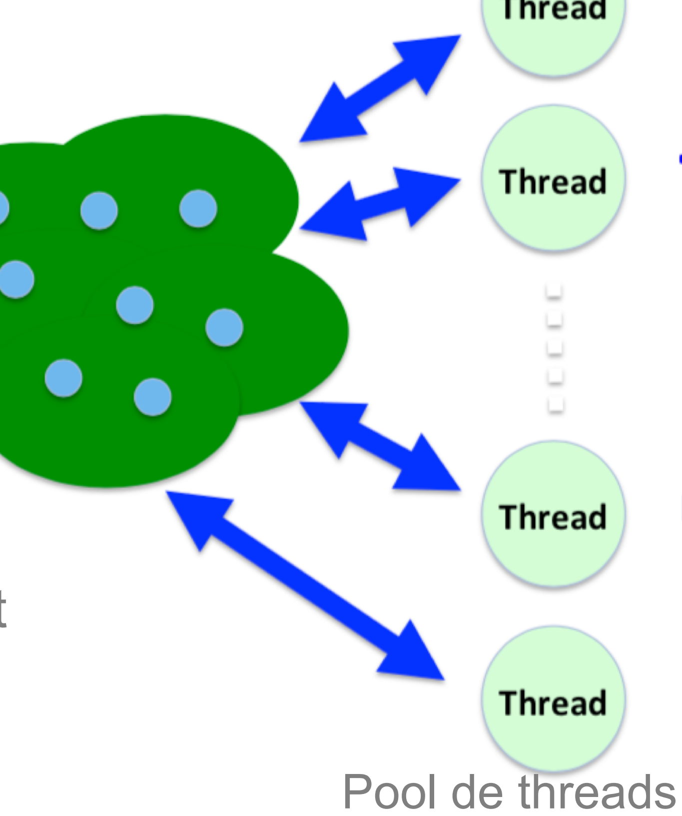
le système d'exécution décide si l'exécution est différée ou immédiate

- Les threads sont affectés pour effectuer le travail des tâches ;
- L'exécution des tâches peut être différée ou immédiate.

# Le concept des tâches dans OpenMP

Un thread OpenMP génère les tâches

Les tâches sont exécutées par le pool de threads de la région parallèle




Le système d'exécution décide si l'exécution est différée ou immédiate.

# Construction de tâches

Construction d'une tâche qui sera exécutée par un thread du pool de threads.

```
#pragma omp task [clause [[,]clause] ...]
    {     structured-block     }
```

On peut imbriquer le constructeur (parallélisme emboité)

Clauses :

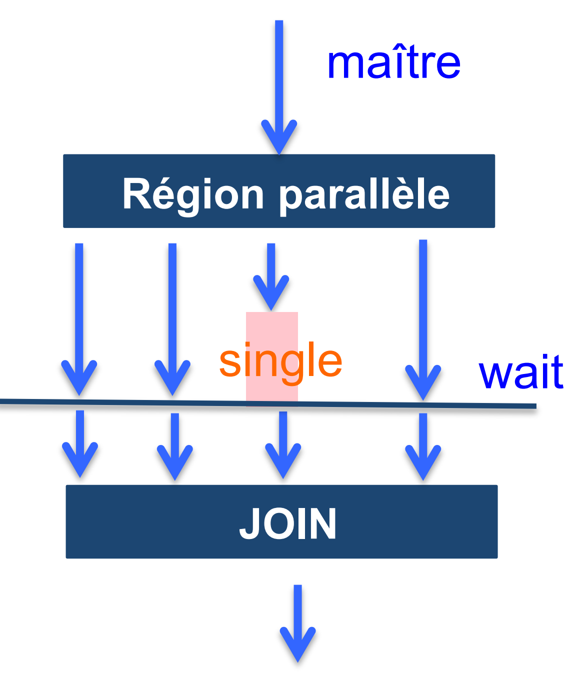
| Clauses | |
|---|---|
| default (shared \| none )<br>private (list) , firstprivate (list) , shared (list) | Partage de données |
| if (expression scalaire)<br>final (expression scalaire)<br>mergeable | Terminaison / limite |
| untied<br>depend (list)<br>priority (value) | Ordonnancement |

# Code classique de génération des tâches

```
…
#pragma omp parallel
{
#pragma omp single
   {
      ….
      #pragma omp task
      { Code de la tâche}

   } // end single region
  …
 } // end parallel region
```

→ création de l'équipe de threads

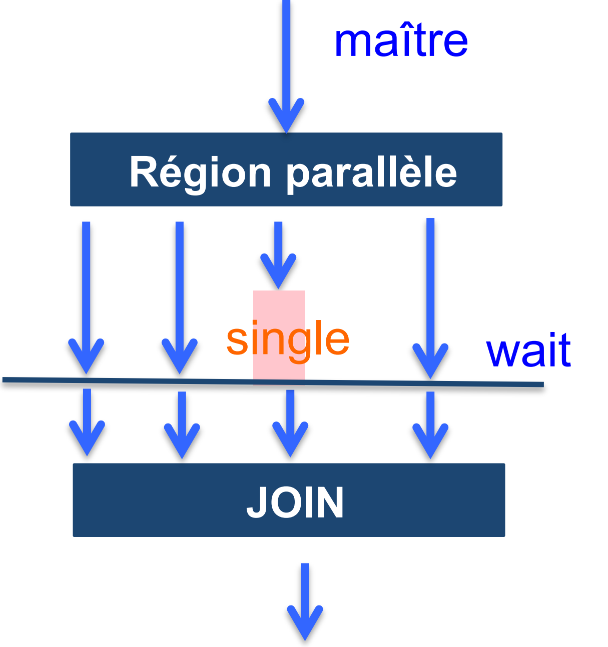
→ Un seul thread génère les tâches et les ajoutent à la queue appartenant à l'équipe



# Exemple : Hello world (1)

```
int main() {
#pragma omp parallel
 {
#pragma omp single
   {

      {  printf("Hello "); }

      { printf("World "); }

     printf("\nThank You ");
   } // End of single region
 } // End of parallel region
printf("\n");
 return(0);
}
```

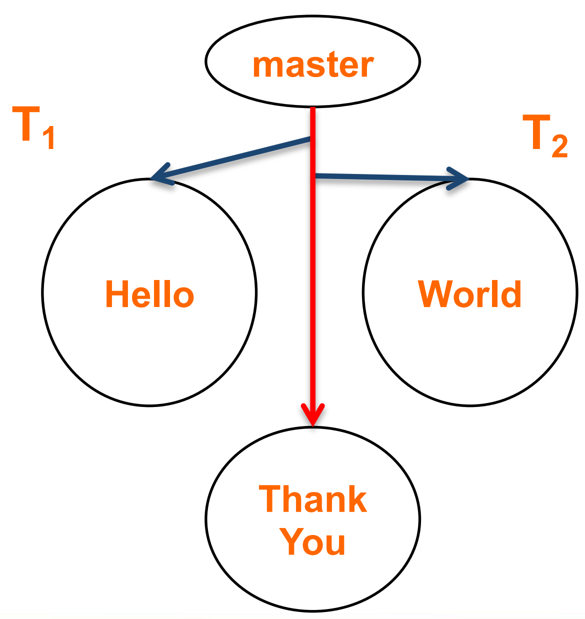
```
$ export OMP_NUM_THREADS=4
$ ./task_hello
Hello World
Thank You
```



# Exemple : Hello world (2)

```
int main() {
#pragma omp parallel
 {
#pragma omp single
  {
#pragma omp task
     {  printf("Hello "); }
#pragma omp task
     { printf("World "); }

    printf("\nThank You ");
  } // End of single region
 } // End of parallel region
printf("\n");
 return(0);
}
```

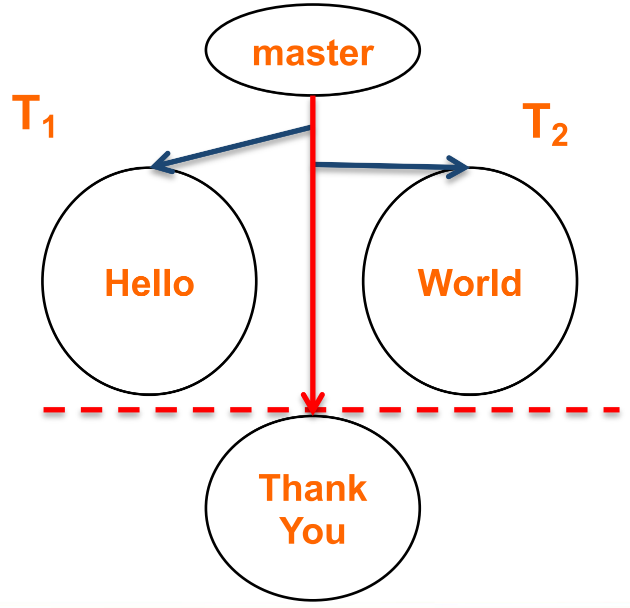
```
$./task_hello_1
Thank You World Hello

$./task_hello_1
Thank You World Hello

$ ./task_hello_1
Thank You Hello World
```



# Exemple : Hello world (3)

```
int main() {
#pragma omp parallel
  {
#pragma omp single
    {
#pragma omp task
      {  printf("Hello "); }
#pragma omp task
      { printf("World "); }
#pragma omp taskwait
      printf("\nThank You ");
    } // End of single region
  } // End of parallel region
printf("\n");
 return(0);
}
```

```
$./task_hello_2
World Hello
Thank You
```



# Le constructeur taskloop

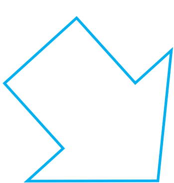
Permet d'éclater une boucle avec des tâches

```
#pragma omp taskloop [clause [[,]clause] ...
        for-loops
```

Clauses :

- default (shared | none )
- private (list) , firstprivate (list) , lsstprivate (list) , shared (list)
- colapse(n)
- if (expression scalaire)
- final (expression scalaire)
- mergeable
- grainsize(val), priority (value)
- nogroup,
- num_tasks(n)
- untied

```
for(i =0;i<SIZE;i+=1)
    {A[i]=A[i]*B[i]*S;}
```



```
for(i =0;i<SIZE;i+=TS){
        UB =SIZE <(i+TS)?SIZE:i+TS;
        for(ii=i;ii<UB;ii++){
                A[ii]=A[ii]*B[ii]*S;}
}
```

```
#pragma omp taskloop grainsize(TS)
for(i =0;i<SIZE;i+=1)
        {A[i]=A[i]*B[i]*S;}
```

```
for(i =0;i<SIZE;i+=TS){
        UB =SIZE <(i+TS)?SIZE:i+TS;
        #pragma omp task   firstprivate(i,UB) \
            shared(S,A,B)
        for(int ii=i;ii<UB;ii++){
                A[ii]=A[ii]*B[ii]*S;}
}
```

Technique de Blocking

Une tâche va traiter au moins TS itérations

et au plus moins de 2 TS itérations

# Clauses d'interruption

Permet d’éviter la création de tâches trop petite

- *if (expr)*

  Si *expr est* évaluée à faux

  - La tâche est exécutée immédiatement par le thread (pas de construction de tâche)
  - Permettre des optimisations définies par l'utilisateur

- *final (expr)*
  - Lorsque l'expression finale est évaluée à true, la tâche n'aura pas de descendants (feuille dans le DAG des tâches) qui seront créés dans le pool partagé des tâches.
  - Permet au runtime d'arrêter la génération et le report de nouvelles tâches (applications récursives et emboitées) et d'exécuter toutes les tâches futures de la directive de tâche actuelle directement dans le contexte du thread d'exécution.

- *mergeable* une tâche dont l'environnement de données est le même que celui de sa région de tâche de génération.

# Clauses d'ordonnancements

- *untied*
  - la tâche est liée par défaut. Il est garanti que le même thread exécutera toutes les parties de la tâche, même si l'exécution de la tâche a été temporairement suspendue.
  - Un générateur de tâches non lié peut être déplacé d'un thread à l'autre, ce qui permet aux tâches d'être générées par différentes entités.

- *priority(val)*
  - val est un indice pour le l'ordonnanceur. Une valeur numérique non négative, qui recommande l'exécution d'une tâche de priorité élevée avant une tâche de priorité inférieure.

```
int foo(int N, int **elems, int *sizes) {
    for (int i = 0; i < N; ++i) {
        #pragma omp task priority(sizes[i])
        compute_elem(elems[i]);
    }
}
```

  - La vraie priorité est

max(*val,* OMP_MAX_TASK_PRIORITY*)*

# Clause sur les variables

default définit les attributs de partage des données des variables qui sont référencées

private: chaque construction a une copie de l'élément de données

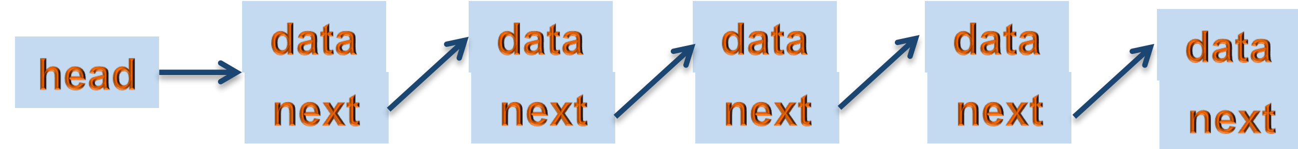
firstprivate: private + donnée initialisée à partir de la construction supérieure avant l'appel

lastprivate: chaque construction a une copie non initialisée, et sa valeur est mise à jour une fois la tâche terminée.

shared: Toutes les références à un élément de liste dans une tâche se réfèrent à la zone de stockage de la variable d'origine.

# Exemple : liste chainée

```
node *p   = listhead ;
while (p) {
        do_independent_work(p) ;
        p = p->next() ;
}
```




Difficile de le faire avant OpenMP 3.0
- Compter le nombre d'itérations
- Transformer en une boucle finie *for*

# Exemple : liste chainée

```
node *p   = listhead ;
#pragma omp parallel
 {
#pragma omp single
   {
      while (p) {
      #pragma omp task  firstprivate (p)
              {
              do_independent_work(p) ;
              }
              p = p->next()
      }
   } // END SINGLE
}   // END PARALLEL
```

- `#pragma omp parallel` ← Création des threads
- `#pragma omp single` ← Un thread exécute la boucle while
- `#pragma omp task firstprivate (p)` ← Création des tâches et exécution en parallèle
- `do_independent_work(p) ;` ← Chaque tâche s'exécute dans un thread
- `} // END SINGLE` ← Quand la tache se termine, le thread associé attend sur la barrière implicite de la construction ***single***

# La clause depend

Permet de préciser les dépendances entre les tâches

→ conduit à des contraintes sur l'ordonnancement des tâches

→ Permet de spécifier l'ordre d'exécution des tâches

***depend*** ***(****dependence-type* ***:*** *list****)***

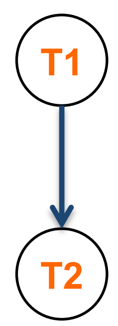
- *dependence-type* = in, out, inout
- *list* : une variable ou une section de tableau

```
depend(inout: x) , depend(inout: a[10:20])
```

- *dependence-type* = in, out, inout
  - in : la tâche dépend de toutes les autres tâches de même parent ayant la variable en out ou inout
  - inout, out : toutes les autres tâches de même parent précédemment produites qui font référence à au moins une des variables avec une dépendance de type **in**, **out**, or **inout**

# Exemple (1)

```
#include <stdio.h>
int main() {
      int x;
      #pragma omp parallel
       #pragma omp single
      {
      #pragma omp task shared(x) depend(out: x)
      x = 1;
      #pragma omp task shared(x) depend(out: x)
      x = 2;
      #pragma omp taskwait
      printf("x = %d.\n ", x);
      }
return 0;
}
```



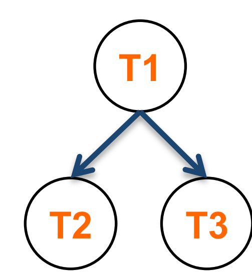

Le programme affiche toujours 2.

Force l’ordre d’exécution des tâches

Si pas de clause *depend* l’affichage est soit 1 soit 2

# Exemple (2)

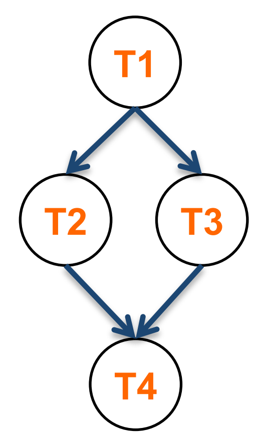
```
#include <stdio.h>
int main() { i
    int x= 1;
    #pragma omp parallel
     #pragma omp single
    {
    #pragma omp task shared(x) depend(out: x)
    x = 2;
    #pragma omp task shared(x) depend(in: x)
    printf("x + 1 = %d. ", x+1);
    #pragma omp task shared(x) depend(in: x)
    printf("x + 2 = %d\n", x+2);
    }
return 0;
}
```




Pas d'ordre sur T2 et T3

```
% ex2
x + 1 = 3. x + 2 = 4
% ex2
x + 2 = 4
x + 1 = 3.
```

# Exemple (3)




```
#include <stdio.h>
int main() { i
      int x= 1;
      #pragma omp parallel
       #pragma omp single
      {
      #pragma omp task shared(x) depend(out: x)
      x = 2;
      #pragma omp task shared(x) depend(in: x)
      printf("x + 1 = %d. ", x+1);
      #pragma omp task shared(x) depend(in: x)
      printf("x + 2 = %d\n", x+2);
      #pragma omp task shared(x) depend(out: x)
      printf("x + 4 = %d\n", x+4);

      }
return 0;
```

Pas d'ordre sur T2 et T3

% ex2

x + 1 = 3.x + 2 = 4

x + 4 = 6

% ex2

x + 2 = 4

x + 1 = 3.x + 4 = 6

# Cholesky

```
void blocked_cholesky( int NB, float A[NB][NB] ) {
   int i, j, k;
   for (k=0; k<NB; k++) {
     #pragma omp task depend(inout:A[k][k])
         spotrf (A[k][k]) ;
     for (i=k+1; i<NT; i++)
       #pragma omp task depend(in:A[k][k]) depend(inout:A[k][i])
           strsm (A[k][k], A[k][i]);
       // update trailing submatrix
       for (i=k+1; i<NT; i++) {
         for (j=k+1; j<i; j++)
           #pragma omp task depend(in:A[k][i],A[k][j])
                            depend(inout:A[j][i])
               sgemm( A[k][i], A[k][j], A[j][i]);
         #pragma omp task depend(in:A[k][i]) depend(inout:A[i][i])
             ssyrk (A[k][i], A[i][i]);
       }
   }
}
```

$$A_{k,k} = \sqrt{A_{k,k} - \sum_{j=1}^{i-1} L_{k,j}^2}$$

$$L_{ji} = \frac{A_{ij} - \sum_{k=1}^{i-1} L_{ik} L_{jk}}{L_{ii}}$$

Échanger j et k

# Terminaison

## #pragma omp taskwait

- spécifie une attente sur la terminaison des tâches produites dans la tâche actuelle, pas aux descendants

## #pragma omp barrier

- spécifie une attente à toutes les tâches générées dans la région parallèle actuelle jusqu'à la barrière

## #pragma omp taskgroup

- spécifie une attente sur la terminaison des tâches filles de la tâche actuelle et leurs tâches descendantes

# Construction taskyield

taskyield : spécifie que la tâche peut être suspendue au profit d'une autre tâche

```
void foo ( omp_lock_t * lock, int n ) {
  int i;
  for ( i = 0; i < n; i++ )
    #pragma omp task
    {
     something_useful();
     while ( !omp_test_lock(lock) ) {
           #pragma omp taskyield }
     something_critical();
     omp_unset_lock(lock);
     }
}
```

On test si le verrou est libre si oui on le prend pour faire la section critique

On suspend la tâche au profit d'une autre tâche

# Terminaison

Où et quand les tâches sont elles terminées ?

- sur les barrières implicites (fin de région parallèle, …)
- sur les barrières explicites

  #pragma omp barrier

  S'applique à toutes les tâches générées dans la région parallèle

Mais aussi

# Terminaison

Directive *taskwait*

- Attend jusqu'à ce que toutes les tâches définies par le constructeur soient terminées.
- Ne s'applique pas aux descendants

```
#pragma omp task
      {
        printf("task1 \n");
#pragma omp task
         printf("child task1\n ");
      }
#pragma omp taskwait
     printf("Coucou\n");
```

```
./a.out
task1
Coucou
child task1
```

Directive *taskgroup*

- Attend jusqu'à ce que toutes les tâches définies par le constructeur soient terminées ainsi que les descendantes.

```
#pragma omp taskgroup
      {
#pragma omp task
      {
        printf("task2 \n");
#pragma omp task
         printf("child task2\n ");
      }
}
     printf("Coucou\n");
```

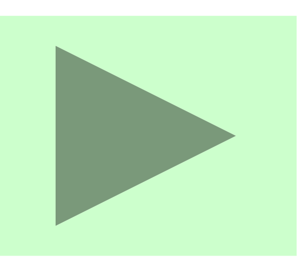
```
./a.out
task2
child task2
Coucou
```

# Exercice : le tri rapide récursif

Paralléliser l'algorithme du Quicksort écrit de manière récursive.

Code séquentiel disponible dans la section OpenMP

http://people.bordeaux.inria.fr/coulaud/Enseignement/PG305



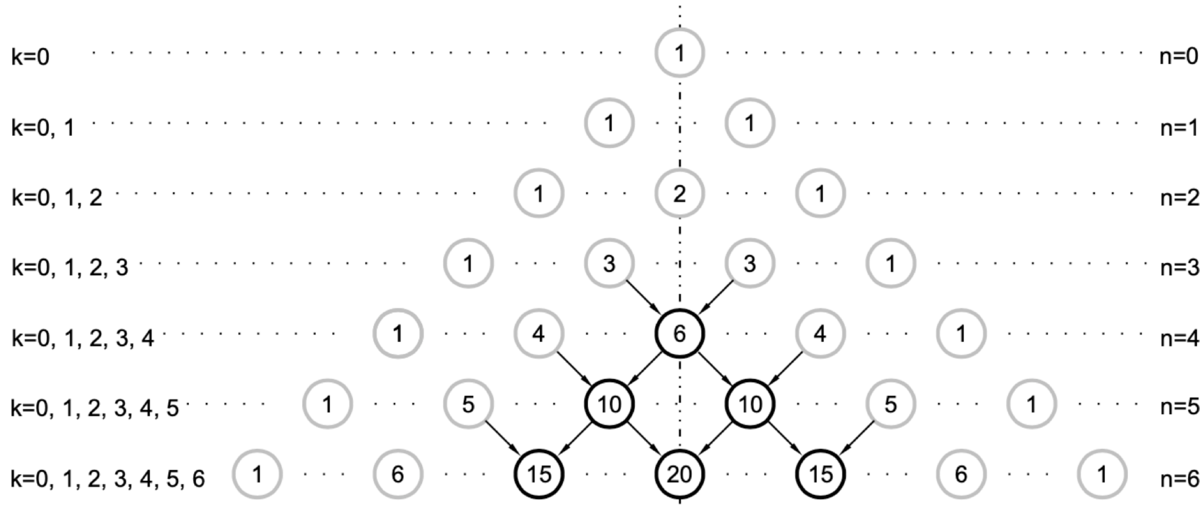
# Exercice : Le triangle de Pascal

Coefficient binomial : $\binom{n}{k} = \frac{n!}{(n-k)!k!}$

Pour les évaluer on va les calculer par la formule de récurrence suivante

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Triangle de Pascal



# Exercice : Les nombres de Fibonacci

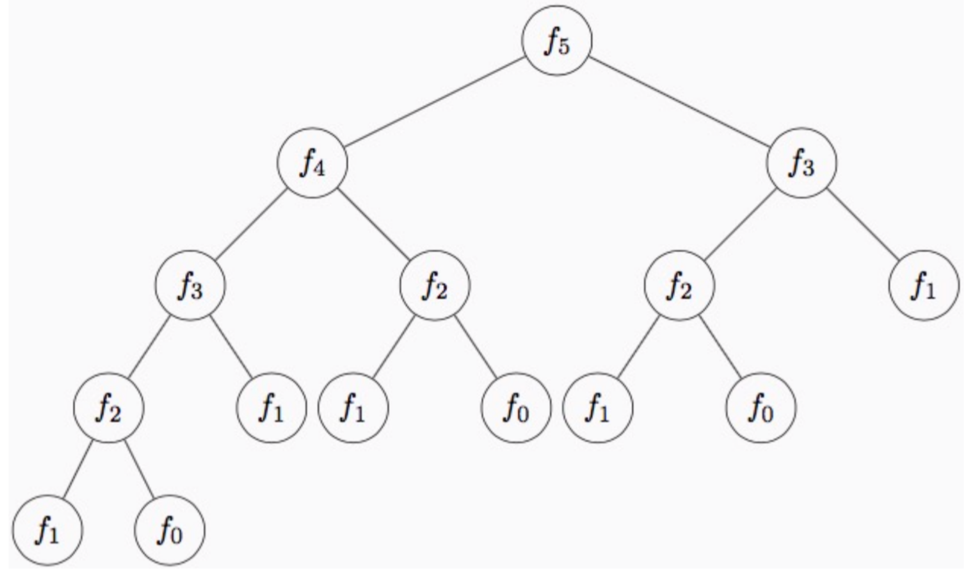
Les nombres de Fibonacci sont définis comme suit

F(0) = 1
F(1) = 1

F(n) = F(n-1) + F(n-2) (n=2,3, …)

Suite :

1, 1, 2, 3, 5, 8, 13, 21, 34



# NOTIONS AVANCÉES

# SIMD

## NOTIONS AVANCÉES

# SIMD

Utile pour faire de la vectorisation automatique

Compilateur ne vectorise pas

- une boucle si
  - Boucle complexe
  - Possède des dépendances
- une fonction « inlinée »

→Jeu d'instructions SIMD pour aller plus vite

# Boucle SIMD

Permet de transformer une boucle en boucle SIMD

→Jeu d'instructions SIMD

#pragma omp simd [clause [[,]clause] ...]

for-loops

Les clauses

**safelen**(length) **linear**(list[:linear-step])

**aligned**(list[:alignment]) (8,16,32 ou 64)

aligne les objets de la liste au nombre de bytes précisé. Si absent dépend de l'implémentation et de la machine

**collapse**(n)

**private**(list) **lastprivate**(list)

**reduction**(reduction-identifier:list)

# Exemple

```
void star( double *a, double *b, double *c, int n, int *ioff )
{
    #pragma omp simd
    for ( int i = 0; i < n; i++) {
        a[i] *= b[i] * c[ i+ *ioff];
    }
}
```

A la compilation :

- ioff n’est pas connu → compilateur suppose que ioff peut être ≤0 ou >0
- a, b et c sont peut être des alias

Remarque si a et c sont des alias et ioff = 2 → dépendance avant

→ Pas de vectorisation

Le pragma force la vectorisation

# Clause safelen(N)

Précise au compilateur qu'il n'y a pas de dépendance pour un vecteur de taille N ou en dessous.

Si la clause **safelen** n'est pas précisée N = le nombre d'itérations

Exemple

```
void work( float *b, int n, int m ) {
  int i;
  #pragma omp simd safelen(16)
  for (i = m; i < n; i++) {
    b[i] = b[i-m] - 1.0;
   }
}
```

Précise que la boucle est sûre sur une longueur de 16 (incluse)

```
b[m]     = b[0] - 1.0
b[m+1] = b[1] - 1.0
…
b[m+n] = b[n-m] - 1.0
```

si $m \geq 16$ code correct

si $m < 16$ comportement non défini

# Fonction SIMD

Autorise la création d'une ou plusieurs versions de la fonction qui peut contenir des arguments différents avec des instructions SIMD pour être appelée dans une boucle SIMD

**#pragma omp declare simd** [clause[[,] clause] ...]
Définition ou déclaration de la fonction

Les clauses

simdlen(length) aligned(argument-list])
linear(argument-list[:constant-linear-step])
uniform(argument-list)
inbranch notinbranch

# Clause uniform et linear

Clause **uniform(val)** précise que val est constant dans tous les appels concurrents

Clause **linear(var:step)** précise que pour chaque itération de la boucle scalaire var est augmenté de step. Défaut **linear(var) ←→linear(var:1)**

```
int main(int argc, char *argv[])
{
int i, k;
float a[1024], b[1024];
…
float op2 = cst
      #pragma omp simd
      for (k=0; k<N; k++) {
            b[k] = fSqrtMul(&a[k], op2);
       }
}
```

```
#pragam omp declare simd linear(op1)uniform(op2)
float fSqrtMul(float *op1, float op2) {
      return sqrt(*op1)*sqrt(op2);
}
```

a

| a[0] | a[1] | a[2] | | | | | |
|---|---|---|---|---|---|---|---|

# Construction de Boucle parallèle SIMD

Permet de spécifier qu'une boucle peut être exécutée en parallèle avec des instructions SIMD

```
#pragma omp for simd [clause [[,]clause] ...]
    for-loops
```

Les clauses sont les mêmes que pour les constructions for et simd.

Étapes

1. Distribution des itérations sur les threads (tâches implicites)
2. Paquets d'instructions sont convertis en instruction SIMD

# Réduction

## NOTIONS AVANCÉES

# Reduction definie par l'utilisateur

On peut définir sa propre fonction de reduction

```
#pragma omp declare reduction
        ( identifier : typelist : combiner )
        [initializer(initializer-expression)]
```

ou :

- identifier est le nom de la fonction
- typelist est la site de types
- combiner est une expression qui met à jour omp_out et omp_in
- Initializer (expression scalaire)

    **omp_priv =** *expression*

# Reduction definie par l'utilisateur

Vecteur de complexe V, et on souhaite faire le produit des éléments

$$Prod = v_0 * v_1 * \ldots * v_n$$

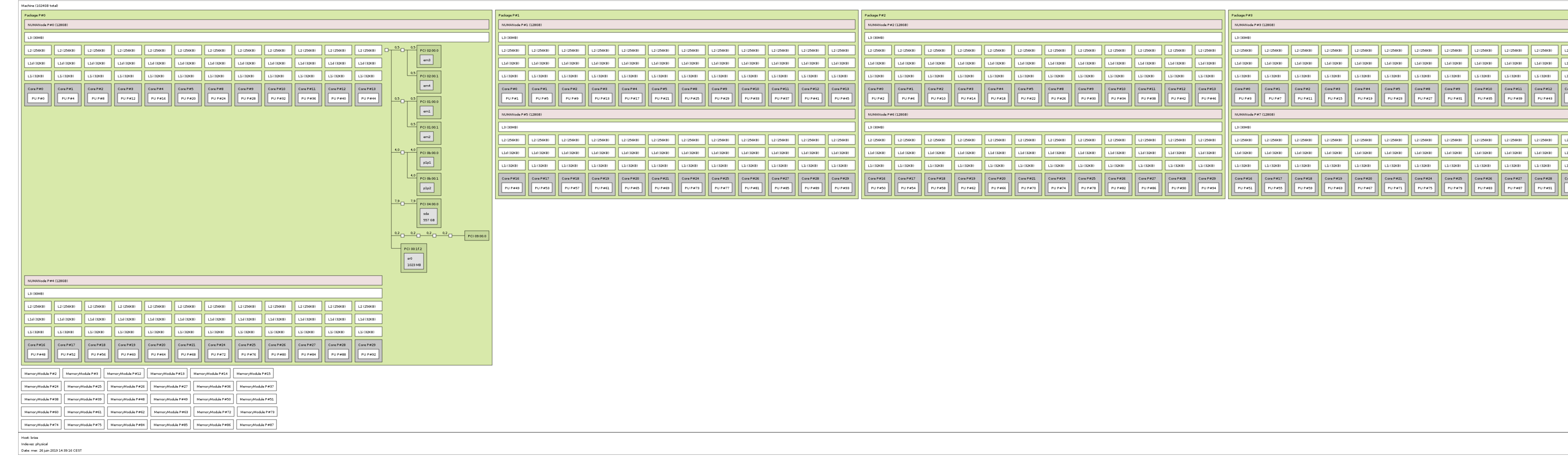
```
int main(){
  using T = std::complex<double> ;
  std::vector<T > vec = { T(1,2), T(3, 4), T(4, 5), T(2,3)};

#pragma omp declare reduction(mymult: T:  omp_out *= omp_in)\
  initializer(omp_priv=T(1))

  T Prod(1) ;
#pragma omp parallel for shared(vec) reduction(mymult:Prod)
num_threads(4)
for(int i = 0; i < vec.size(); i ++)
      { Prod *= vec[i] ; }

    std::cout << "reduction Prod : "<< Prod << std::endl;
}
```

# Placement

# NOTIONS AVANCÉES

# Le placement des threads (1)

L'affinité des threads est importante sur les nœuds multi-socket



Le placement peut être contrôlé par deux variables

1. OMP_PROC_BIND décrit comment les threads sont liés aux emplacement OpenMP.
2. OMP_PLACES décrit les placements en termes de hardware.

Bonne pratique : mettre OMP_DISPLAY_ENV a true

# Le placement des threads (2)

Fixer un thread sur un cœur (in OpenMP 3.1)

```
export OMP_PROC_BIND=true/false
```

Nouvelles extensions pour le placement des threads

1. Plus de possibilités pour OMP_PROC_BIND

   true, false, master, close ou spread

   Pour spécifier comment les tâches (implicites) sont assignées

   - master : affecte les threads de l'équipe à la même place que le thread master
   - close : affecte les threads de l'équipe proche de la place du thread parent
   - spread : étale les threads sur les emplacements disponibles

# Le placement des threads (3)

2. Variable d'environnement OMP_PLACES pour placer les threads
    - par un nom abstrait : threads, cores et sockets
    - par un ordre explicit

3. Ajout d'une nouvelle close pour la construction d'une région parallèle

    proc_bind (master | close | spread)

```
#pragma omp parallel proc_bind(spread) num_threads(4)
{
      work();
}
```

# Le placement des threads (4)

Exemples : architecture à 2 sockets et au total 16 cores

1. OMP_PLACES=sockets OMP_PROC_BIND=close.
   thread 0 va sur core 0 de la socket 0
   thread 0 va sur core1 de la socket 0 …
   thread 7 va sur core 7 de la socket 0
   thread 8 va sur core 8 de la socket 1 …

2. OMP_PLACES=sockets OMP_PROC_BIND=spread
   thread 0 va sur socket 0
   thread 1 va sur socket 1
   thread 2 va sur socket 0 …

# Le placement des threads (5)

Définition des emplacement OpenMP

Trois valeurs prédéfinies : sockets, cores et threads

threads est pertinente sur les processeurs qui ont des threads matériels.

La syntaxe générale pour définir le matériel est : location:number:stride

Différentes expressions

OMP_PLACES="{0:8:1},{8:8:1}“ = sockets 2-sockets et chaque socket a 8 cores consécutifs

OMP_PLACES="{0},{1},{2},...,{15}" = core

Trois placements identiques

setenv OMP_PLACES "{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15} "

setenv OMP_PLACES "{0:4},{4:4},{8:4},{12:4}"

setenv OMP_PLACES "{0:4}:4:4"

# First-touch

Mémoire = organisée en page mémoire. Les adresses virtuelles, mappées sur des adresses physiques, via une table de pages.

Initialisation est importante !!

Exemple

```
double *x = (double*) malloc(N*sizeof(double));

for (i=0; i<N; i++)
    { x[i] = 0; }
#pragma omp parallel  for
for for (i=0; i<N; i++)
     { .... Travail sur  x[i] ...}
```

Initialisation séquentielle
Sur la mémoire de la socket associé au thread master

La mémoire allouée avec malloc et des routines similaires n'est pas immédiatement mappée

# First-touch

Une solution

```
double *x = (double*) malloc(N*sizeof(double));
#pragma omp parallel
{
#pragma omp for schedule(static)
for (i=0; i<N; i++)
    { x[i] = 0; }

#pragma omp for schedule(static)
for for (i=0; i<N; i++)
     { .... Travail sur  x[i] ...}
}
```

# Conclusion (1)

Il est facile d'insérer des directives OpenMP

Toutefois, pour avoir de bonnes performances

- Le coût des synchronisations doit être réduit
- La localité des données doit être optimisée à tous les niveaux

Le style SPMD style conduit à de bonnes performances

- Mais demande beaucoup d'effort à programmer

OpenMP a la flexibilité pour permettre les deux sortes de programmation

# Conclusion (2)

- Les plus
  - Facilité de programmation
  - Parallélisation incrémentale
  - Haut niveau d'abstraction
  - Bonne performance (modèle SPMD)

- Les moins
  - Des manques (aspect NUMA thread ou data affinity)
  - * Manque de performances sur les tâches
  - * Pas d'information sur ce que fait le runtime

- Pour plus de détails : http://www.openmp.org

# Deux modèles

Fork and Join

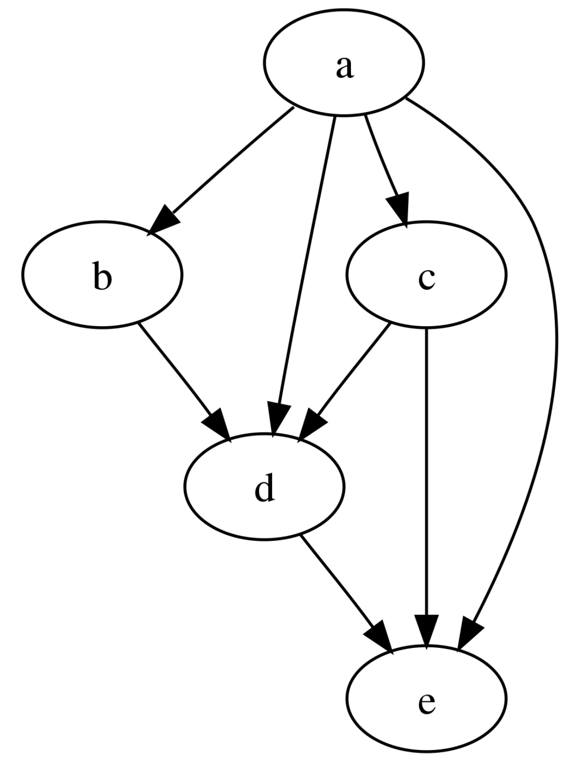
Fork

Fork

Join

Join

Data flow



# FIN

# 5.X

- Schedule modifier
  - Nonmonotonic schedule is the default if static, ordered are not used

# Le constructeur taskloop

Permet d'éclater une boucle avec des tâches

```
#pragma omp taskloop [clause [[,]clause] ...
        for-loops
```

Clauses :

default (shared | none )
private (list) , firstprivate (list) , lsstprivate (list) , shared (list)
colapse(n), reduction(op,liste), in_reduction(op:list)

if (expression scalaire)
final (expression scalaire)
mergeable

grainsize(val).
num_tasks(n)
untied
priority (value)