



O. Coulaud

IT389

# Plan

Quelques rappels

le modèle OpenMP, tâches, ...

Notions plus avancées

SIMD, dépendances (5.x), placement

TP

- N-body
- Produit matrice-matrice

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

Email: [olivier.coulaud@inria.fr](mailto:olivier.coulaud@inria.fr)

# Références



Site Web officiel : [www.openmp.org](http://www.openmp.org)

## **Spécification OpenMP 5.1**

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

**Tutoriels**: <https://www.openmp.org/resources/tutorials-articles/>

**Exemples** : <https://passlab.github.io/Examples>

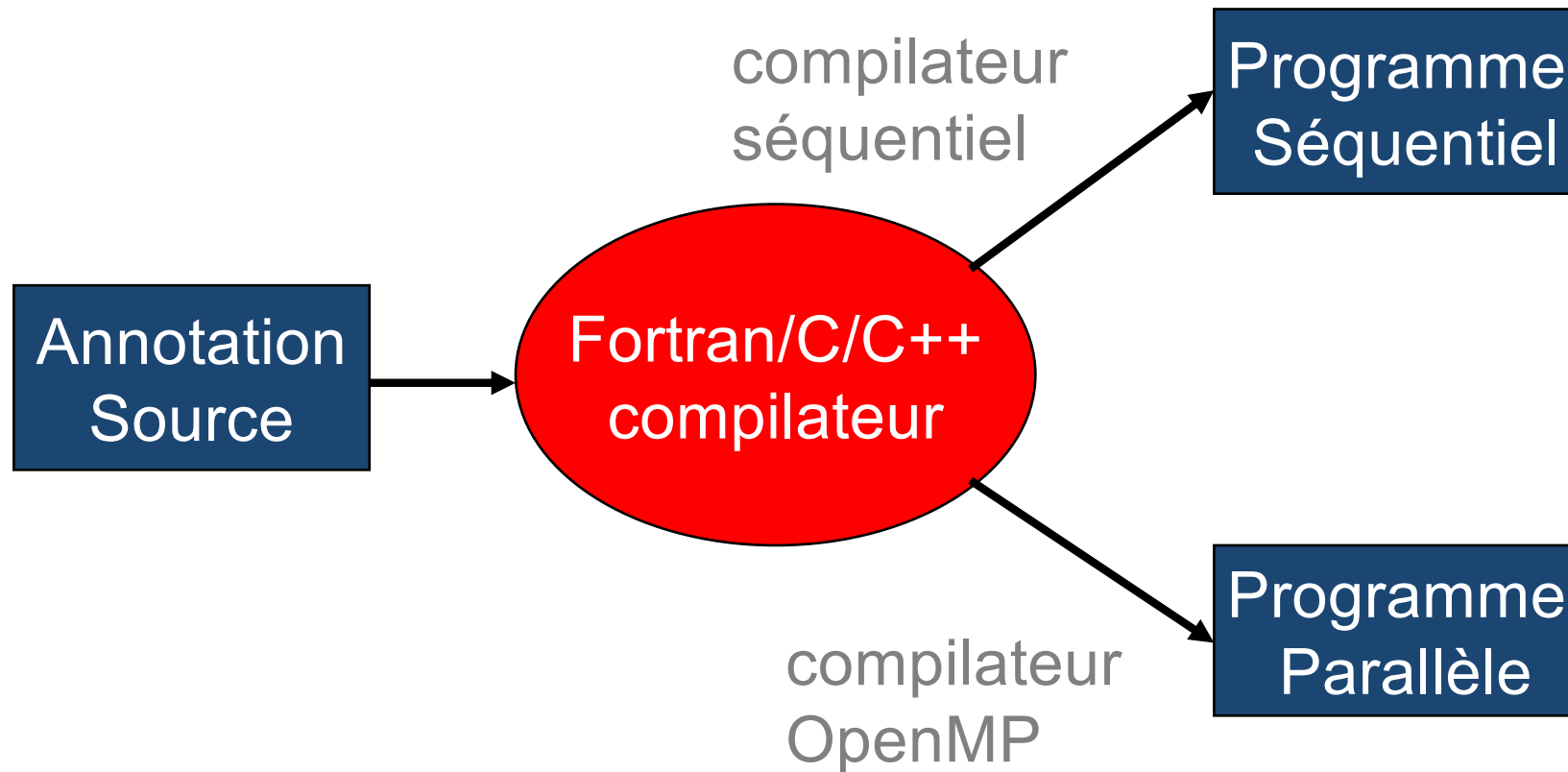
## Cours

Programmation Parallèle (PAP) <https://gforgeron.gitlab.io/pap/>  
Environnement EasyPAP

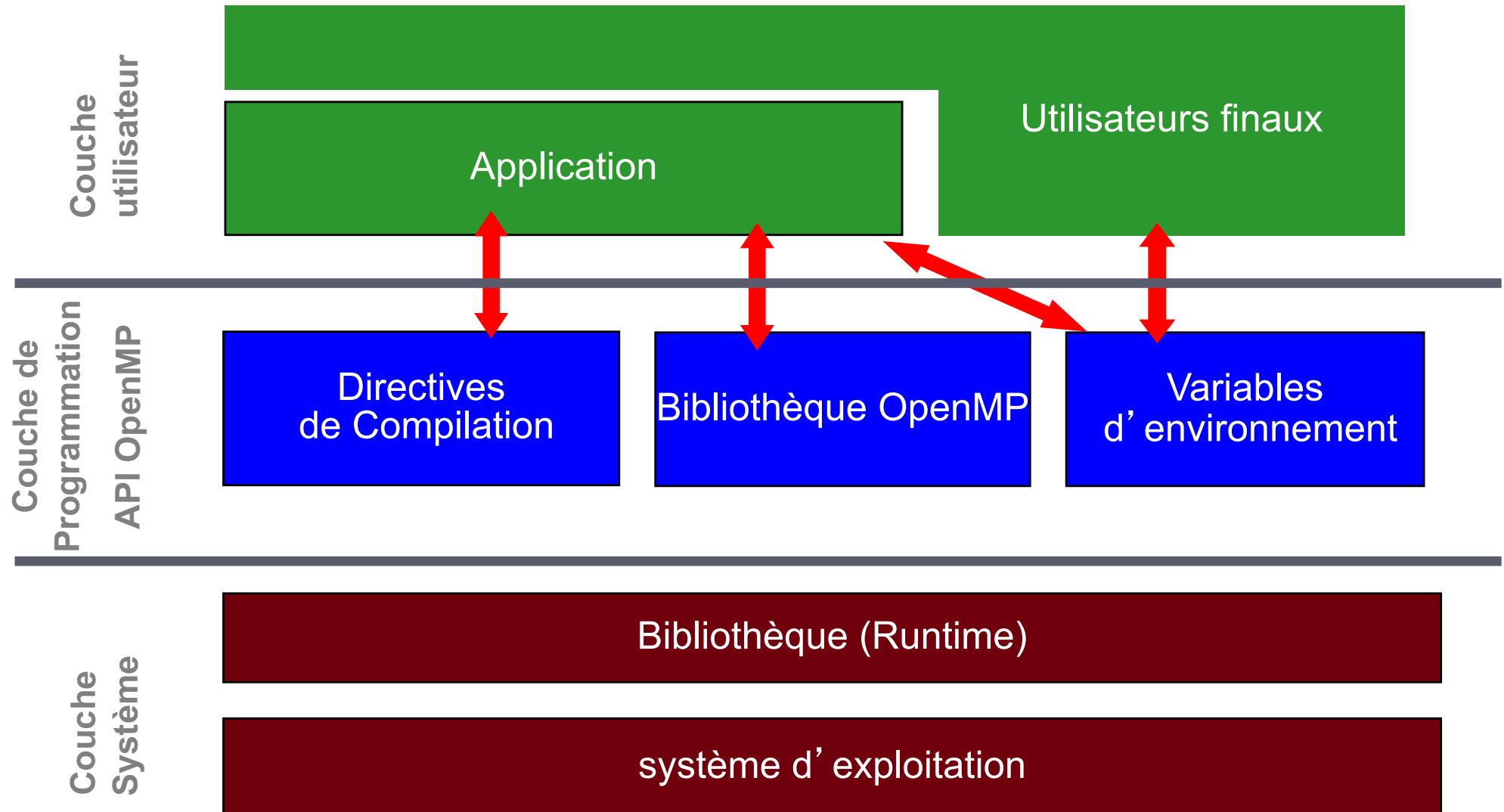
# QUELQUES RAPPELS SUR OPENMP

# Le modèle OpenMP

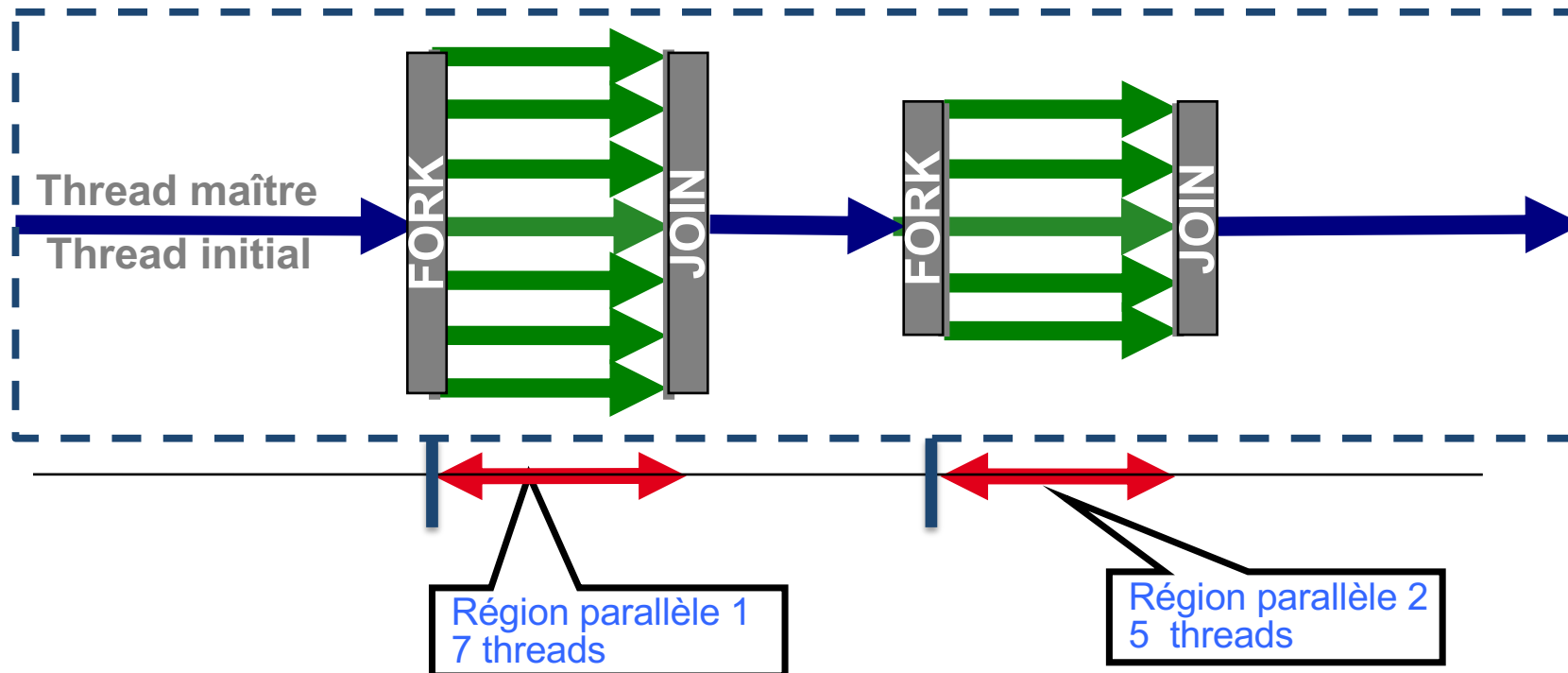
# Comment utiliser OpenMP



# Architecture OpenMP



# Modèle d'exécution



➡ Partie séquentielle = région parallèle implicite

Modèle **Fork and join** : le maître lance un ensemble de threads

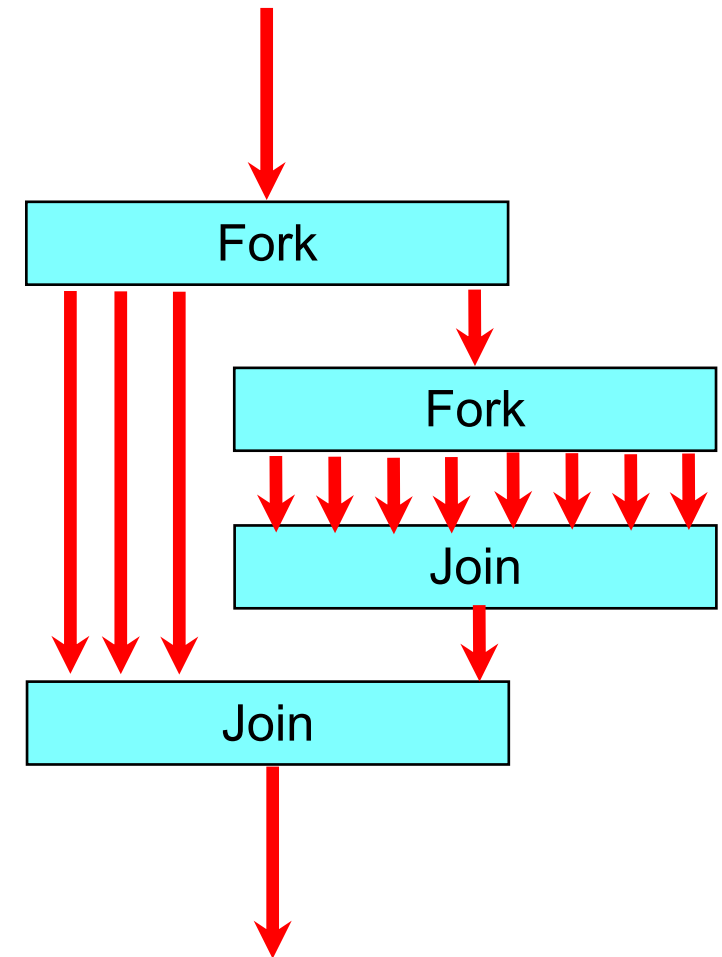
team = master + autres threads



# Modèle d'exécution (suite)

Le modèle Fork/Join peut être emboîté

- 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.

Éclater la boucle entre plusieurs threads

```
void main()
{
    double Res[1000];

    for(int i=0;i<1000;i++) {
        do_huge_comp(Res[i]);
    }
}
```

Programme séquentiel

```
void main()
{
    double Res[1000];

    #pragma omp parallel for
    for(int i=0;i<1000;i++) {
        do_huge_comp(Res[i]);
    }
}
```

Programme Parallèle

# Exemple

## Code source

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

## Compilation

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

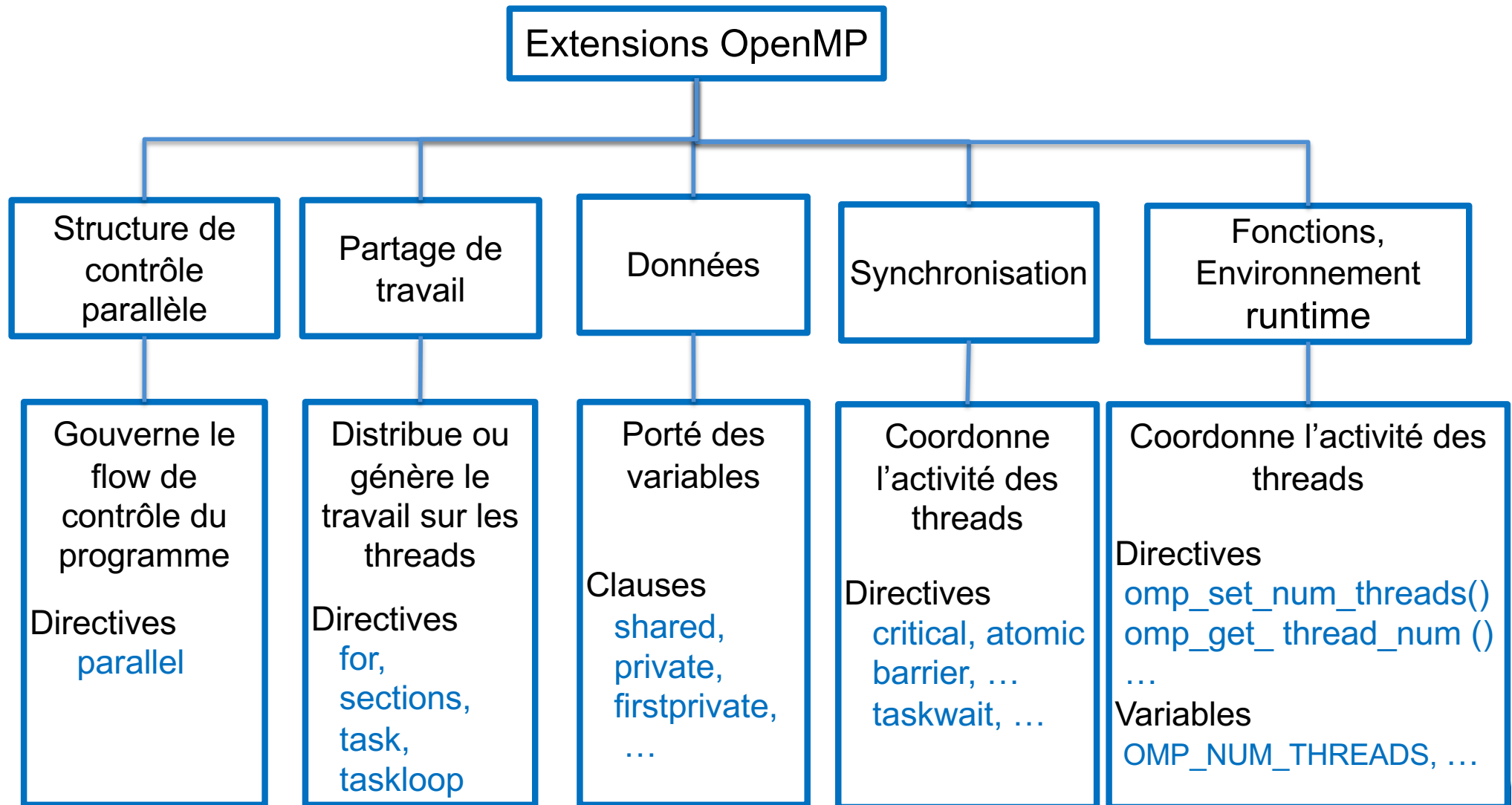
INTEL : icc **-qopenmp** filename.cc -o filename

## Exécution

\$ export OMP\_NUM\_THREADS=4

\$ filename

# OpenMP en 1 transparent



# RÉGION PARALLÈLE

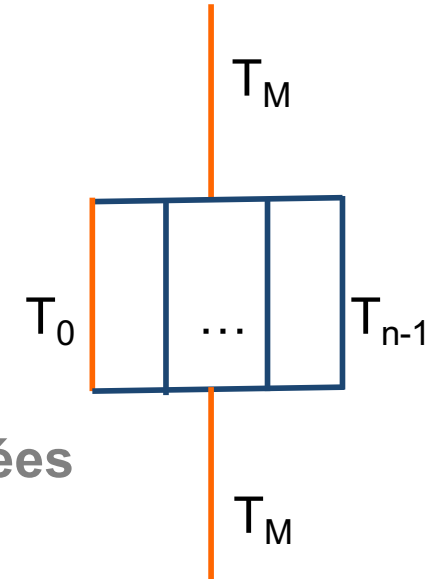
# OpenMP : Région Parallèle (1)

Création uniquement par

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

Duplication de l'exécution

- **chaque thread exécute 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, une 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)

`proc_bind` (master | close | spread)



Contrôler la granularité.

Contrôler le placement.

# 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 `for, taskloop`
- la directive `sections`
- la directive `single`
- La directive `tasks` ← pas de barrière implicite

Restrictions

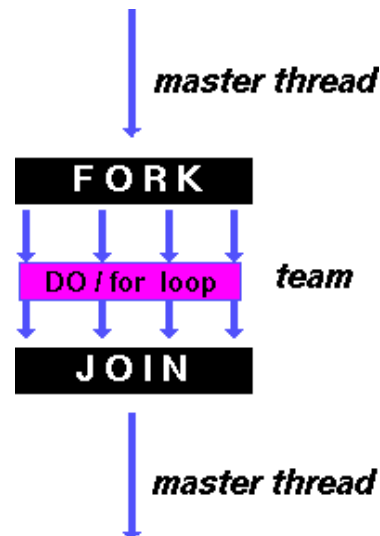
- Chaque région de partage doit être rencontrée par tous les threads ou par aucun (sauf `tasks`)
- La séquence de partage et de barrière doit être la même sur chaque thread

# Construction de travail partagé

## La directive for (1)

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

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



Parallélisme de données

# Construction de travail partagé

## La directive for (2)

### Clauses

- private, firstprivate, lastprivate
- reduction (operator : list)
- linear (list[:linear-step])
- 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*

Les threads s'attendent  
i.e. barrière

# OpenMP: différentes approches de $a = b+c$

Code séquentiel

```
for (int i=0; i<N; 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]; }  
}
```

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]; }  
}
```

# 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 connaît 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 clause, l'ordonnancement **dépend de l'implémentation !!!**



# L'ordonnancement du travail

## La clause SCHEDULE

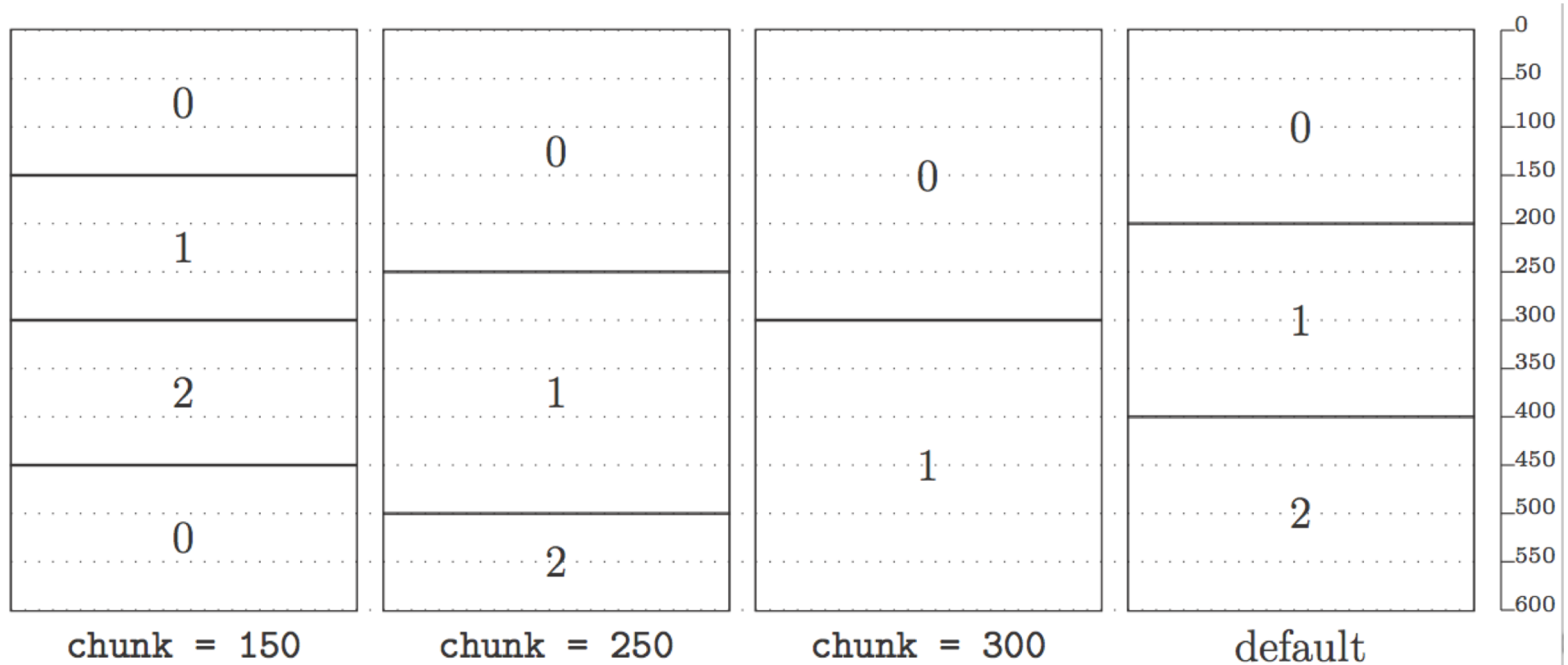
Format : **schedule** ([**modifieur**[:,] **type** [,**chunk\_size**])

**modifieur** 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

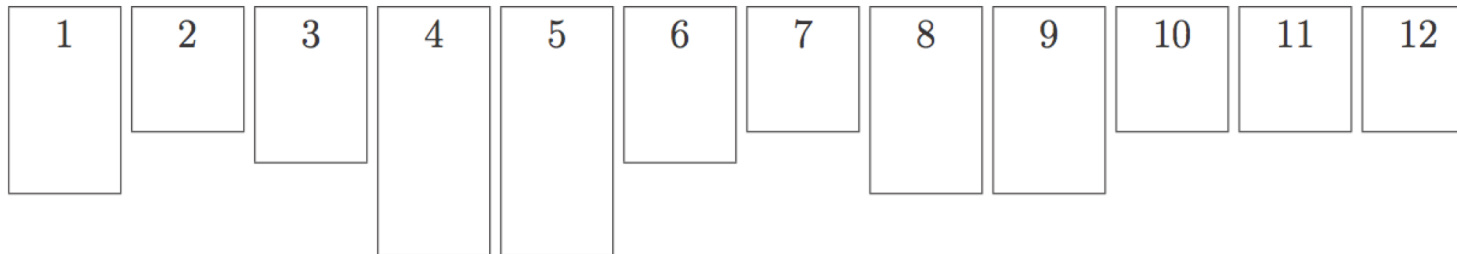
## Le type STATIC



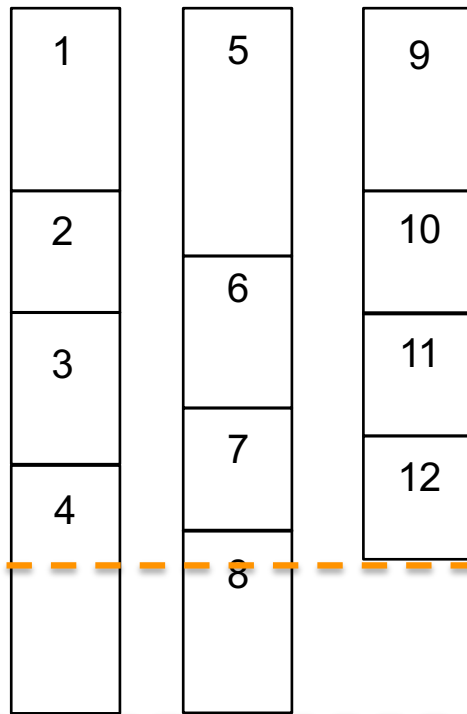
Exemple :  
600 itérations  
**SCHEDULE(STATIC, chunk)**

# Cas irrégulier

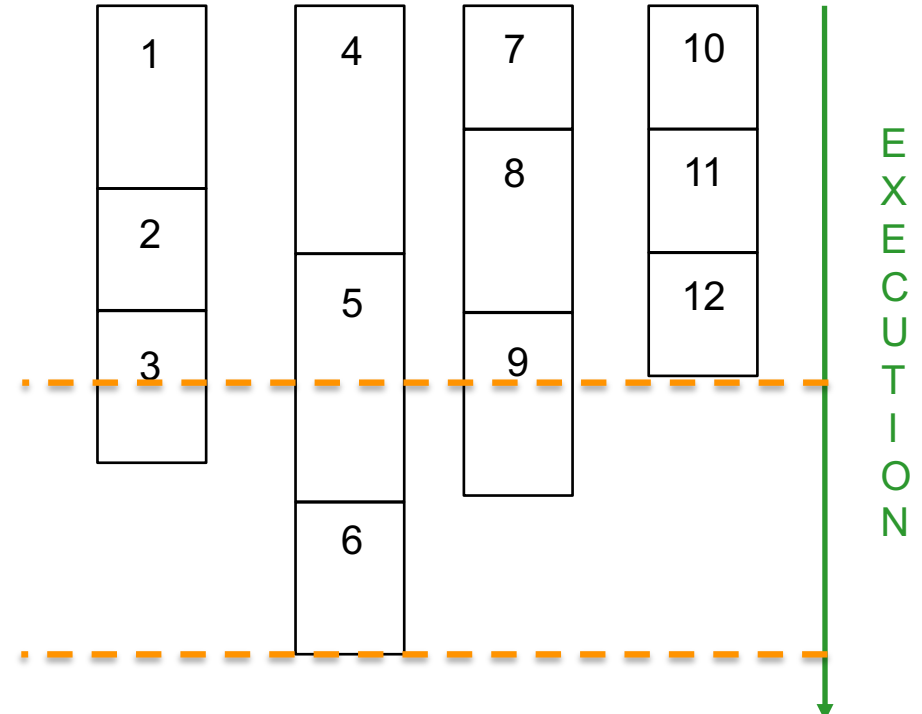
Pieces of work



Thread 1 Thread 2 Thread 3

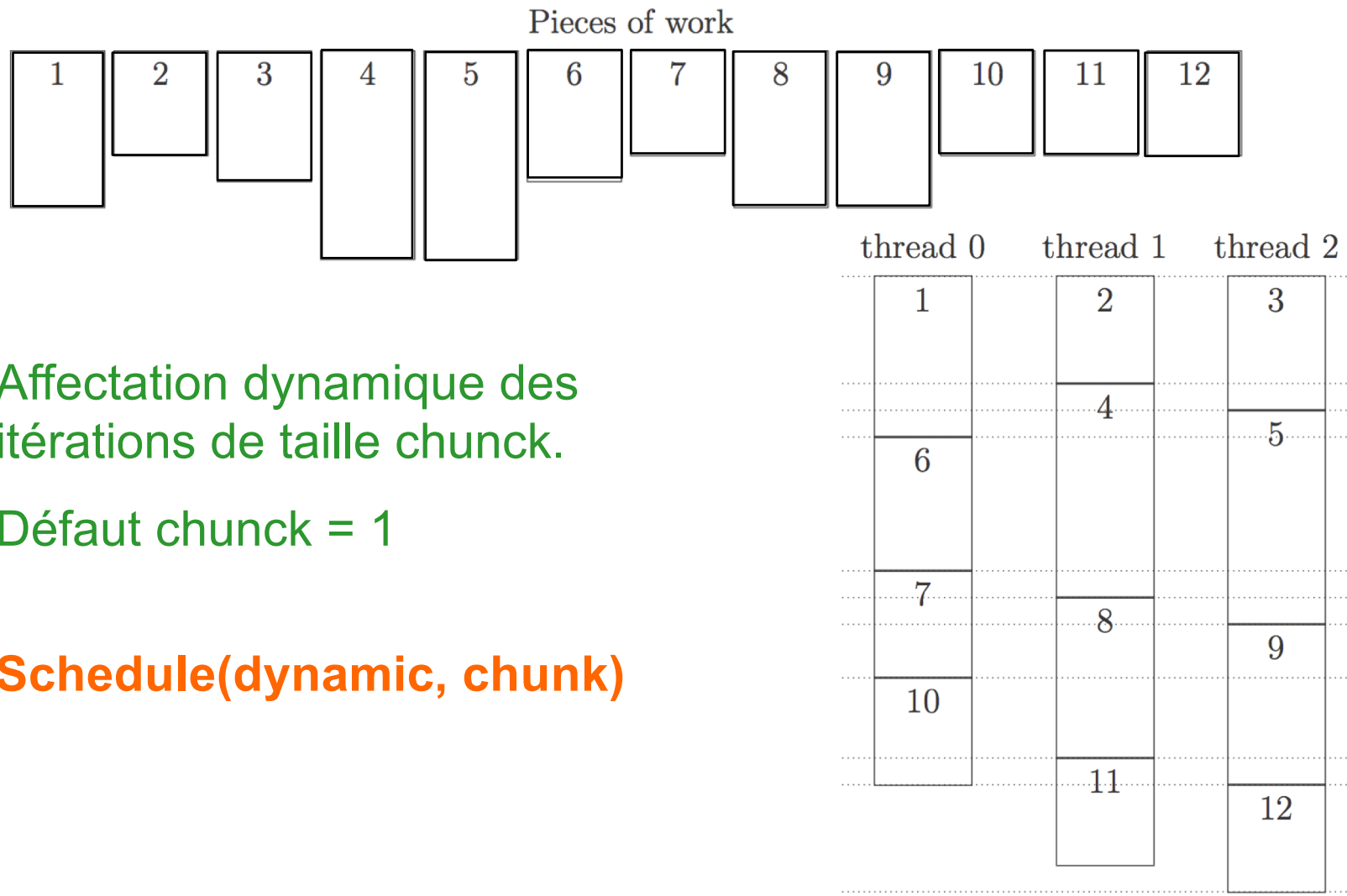


Thread 1 Thread 2 Thread 3 Thread 4



# L'ordonnancement du travail

## Le type DYNAMIC



# 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

$q = \text{ceil}(n/(a \cdot p))$  (entier supérieur)

Puis on itère avec  $n = \max(n - q, a \cdot k \cdot p)$  avec  $a = 1$  ou  $2$  (implémentation)

Exemple :

800 itérations éclatées sur 2 threads  $k = 80$

**schedule(guided, 80)**

$a = 2 \rightarrow 800 / (2 \cdot 2) = 200, (800 - 200) / (2 \cdot 2) = 150, 113, 85, 80 (63), 80, 80, 12$

$a = 1 \rightarrow 800 / 2 = 400, 200, 100, 80, 20$

# 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

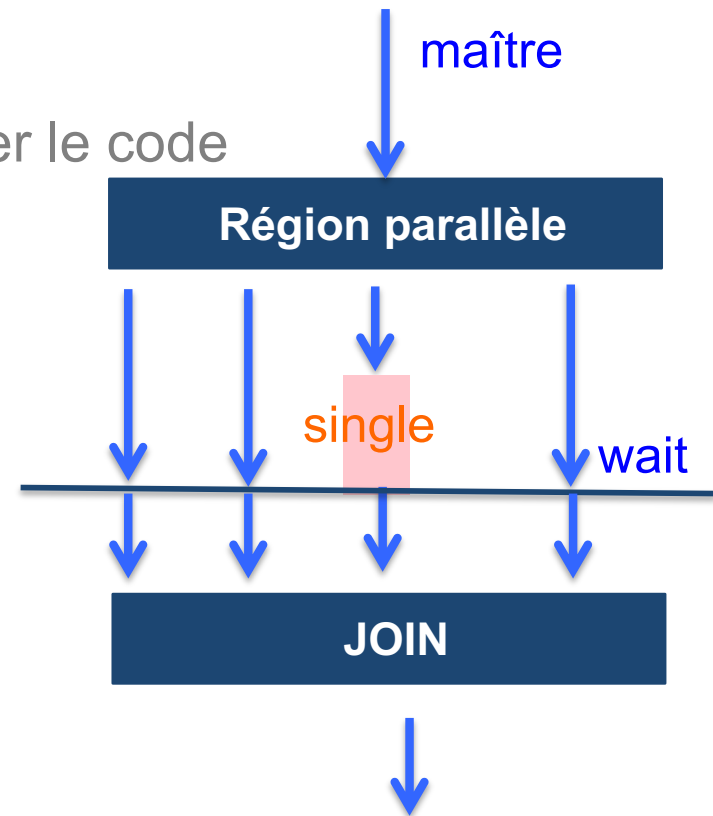
# La directive SINGLE

Un seul thread du groupe va exécuter le code

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

Clauses :

private, firstprivate  
copyprivate, 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** *int\_literal* contrôle le nombre maximal de parallélisme emboîtée 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\_CANCELLATION** : 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

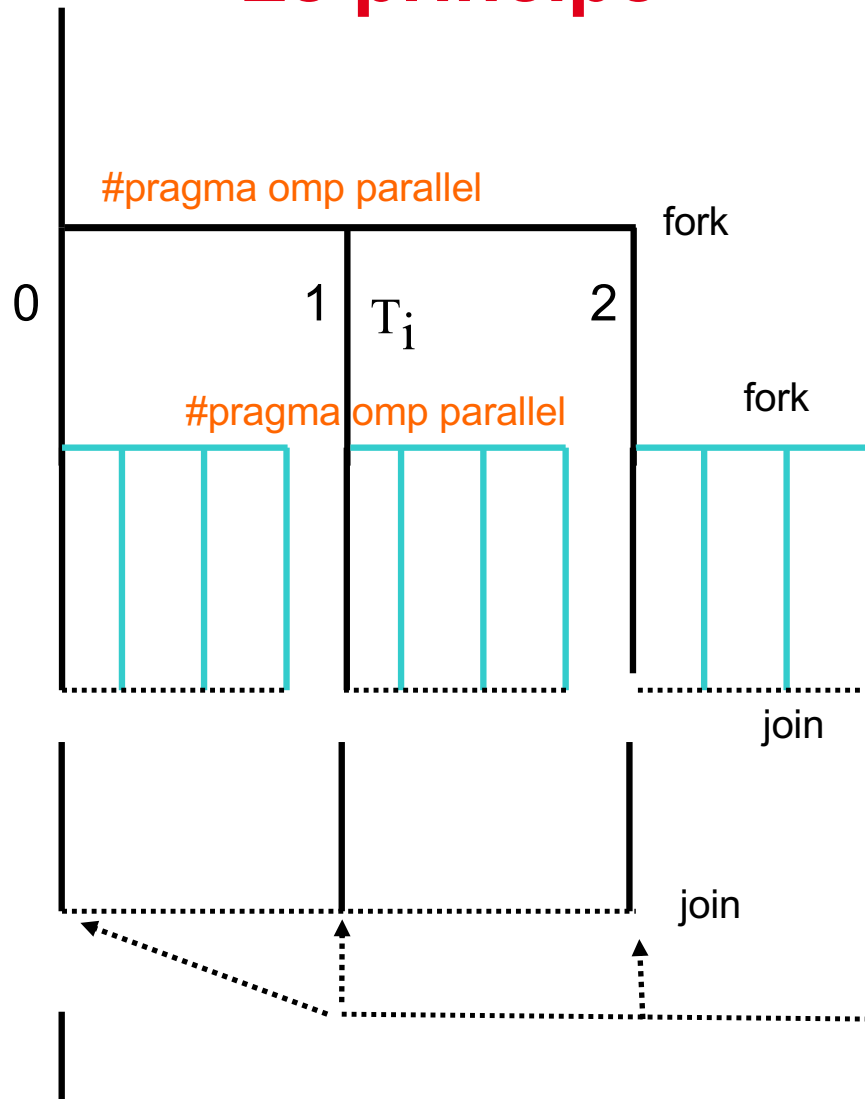
```
omp_get_wtime(), omp_get_wtick()
```

```
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);
```

# 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**

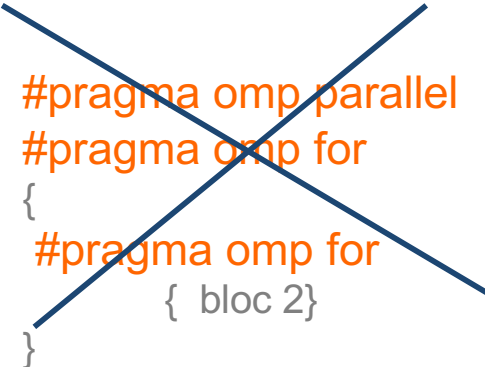


# Parallélisme emboîté

## 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_limit()` pour connaître le nombre de threads disponible



## Quelques méthodes utiles (2)

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

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, ...,  $\text{level}_{\max}$ )

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 connaître 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

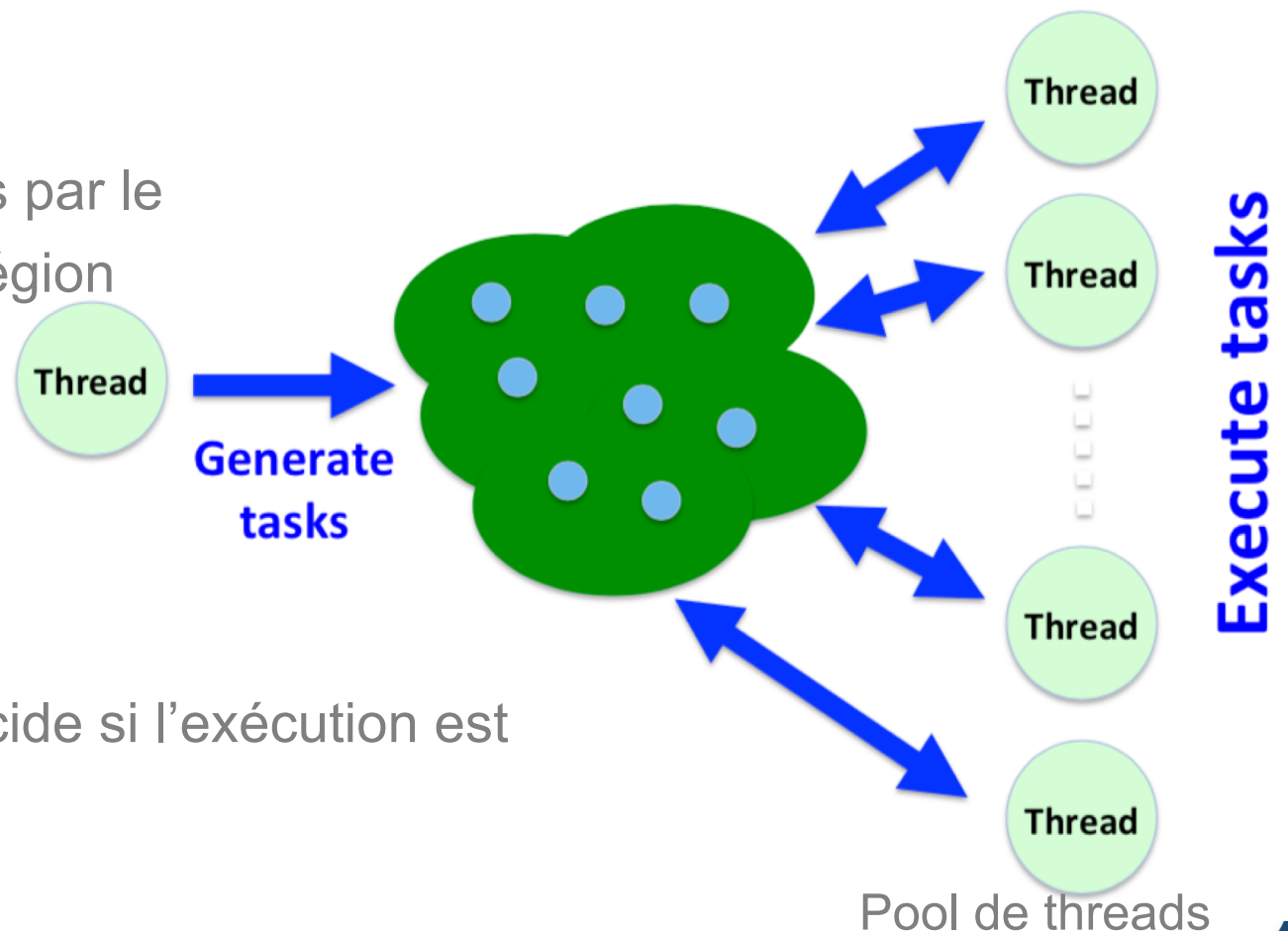
- Les threads sont affectés pour effectuer le travail des tâches ;
- L'exécution des tâches peut être **différée** ou **immédiate**.

# 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 :

default (shared | none )

private (list), firstprivate (list), shared (list),

Partage de données

if (expression scalaire)

final (expression scalaire)

mergeable

Terminaison / limite

untied

depend (list)

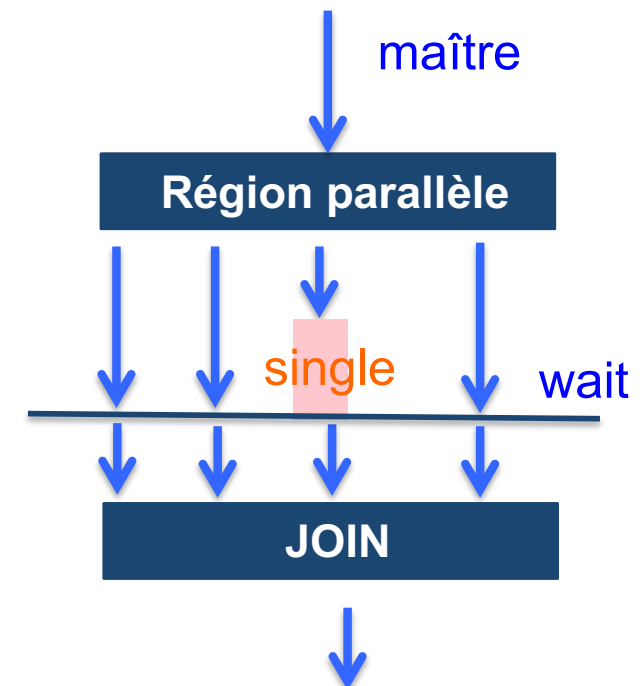
priority (value)

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
```

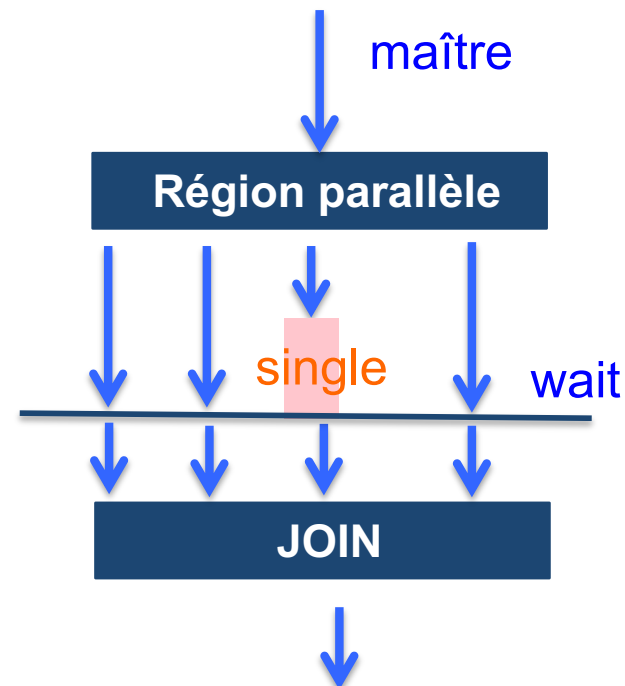
- création de l'équipe de threads
- Un seul thread génère les tâches et les ajoutent à la queue appartenant à l'équipe



# 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
```





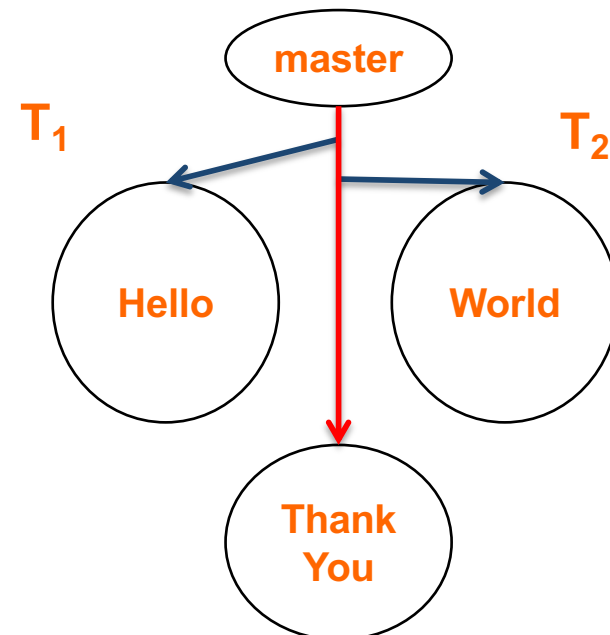
## Exemple : Hello world (2)

```
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
```

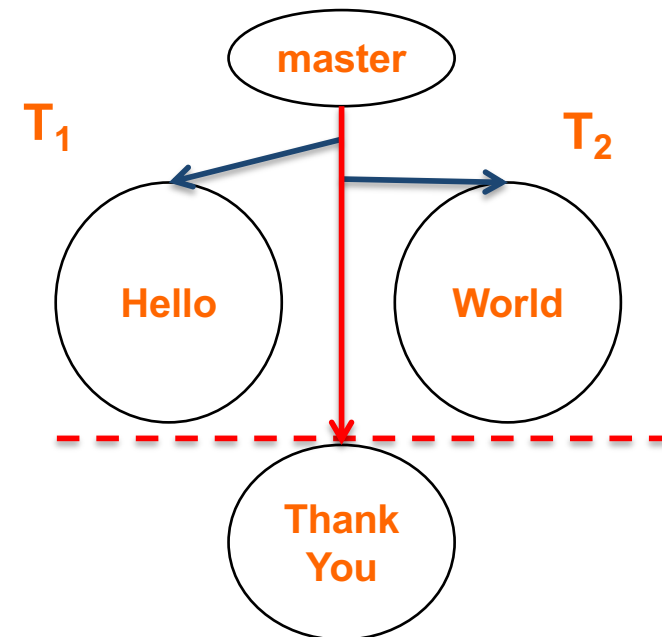
```
$ ./task_hello_1  
Thank You Hello World
```



## 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 de paralléliser avec des tâches explicites une boucle

```
#pragma omp taskloop [clause [[,]clause] ...
```

*for-loops*

Le thread génère les tâches de la boucle

**Clauses :**

default (shared | none )

private (list) , firstprivate (list) , lastprivate (list) , shared (list)

collapse(n)

if (expression scalaire)

final (expression scalaire)

mergeable

grainsize(val), priority (value)

nogroup,

num\_tasks(n)

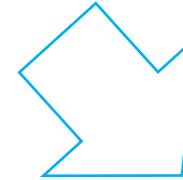
untied



**ATTENTION : ce n'est pas une directive de partage de travail. !!**



```
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;}
```

Technique de blocking  
Une tâche va traiter au moins  
TS itérations et au plus  
moins de 2 TS itérations

```
{
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;}
}
```

# 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 **final** 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 emboîtées) 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'ordonnements

- *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

$\min(\text{val}, \text{OMP\_MAX\_TASK\_PRIORITY})$

# 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 chaînée

```
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
```

← Création des threads

← Un thread exécute la boucle while

← Création des tâches et exécution en parallèle

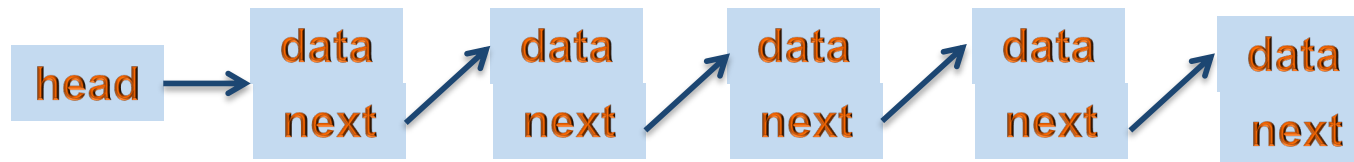
← Chaque tâche s'exécute dans un thread

← Quand la tâche se termine, le thread associé attend sur la barrière implicite de la construction **single**



## Exemple : liste chaînée

```
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*

# 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;
}
```



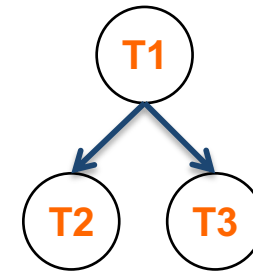
Le programme affiche toujours 2.

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() {
    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

$x + 1 = 3$ .  $x + 2 = 4$

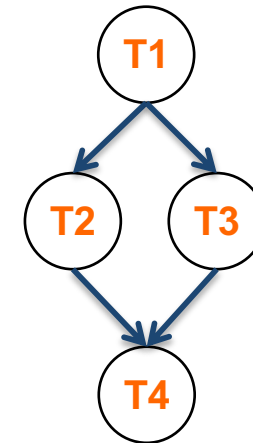
% ex2

$x + 2 = 4$

$x + 1 = 3$ .

## 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

$x + 2 = 4$

$x + 1 = 3$ .  $x + 4 = 6$

# Terminaison

## **#pragma omp taskwait**

- spécifie une attente sur la terminaison des tâches produites dans la tâche actuelle ; ne s'applique 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
```

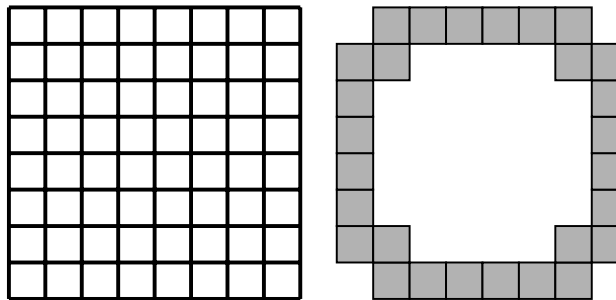
# Dépendances

## **NOTIONS AVANCÉES**

# La clause depend (nouveau)

## Quelques limitations

Les dépendances doivent être connues à la compilation  
Impossible si le nombre de dépendances est dynamique



L'ordre de soumission implique un ordre d'exécution  
pas d'ordre dans les tâches impliquées dans réduction

# 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** (*[depend-modifier]*, *dependence-type* : *list*)

*dependence-type* = in, out, inout, mutexinoutset, inoutset, depobj

- *list* : une variable ou une section de tableau

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

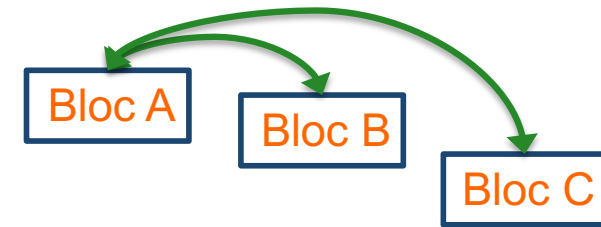
- *depend-modifier* = iterator(definition)

# La clause depend

*dependence-type* = in, out, inout, mutexinoutset, inoutset, depobj

- in : la tâche dépend de toutes les autres tâches de même parent ayant la variable en out, inout, mutexinoutset ou inoutset
- 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, inout, mutexinoutset ou inoutset
- mutexinoutset : même que inout, out. De plus, si deux tâches sont générées avec une dépendance mutexinoutset, les tâches sont mutuellement exclusives.
- inoutset : un ensemble de tâches mutexinoutset
- depobj : les dépendances de la tâche sont dérivées des dépendances représentées par les objets

# Exécution mutuellement exclusive avec dépendances (1)



Exemple :

Bloc A interagit avec les blocs B, C, D

en mode RW et mutuel

1 tâche sur chaque bloc  $T_A$ ,  $T_B$ ,  $T_C$

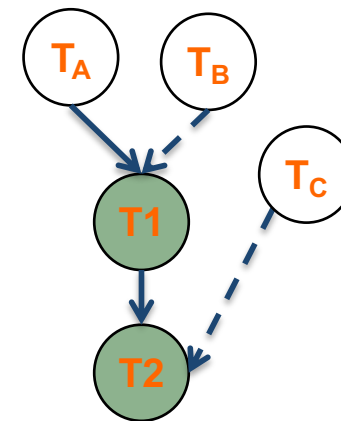
Interaction = 1 tâche

Pas d'ordre dans les tâches  $T_1$ ,  $T_2$

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(inout: A, B)

    // task T1
    #pragma omp task depend(inout: A, C)

    // task T2
}
```



# Exécution mutuellement exclusive avec dépendances (2)

Exemple :

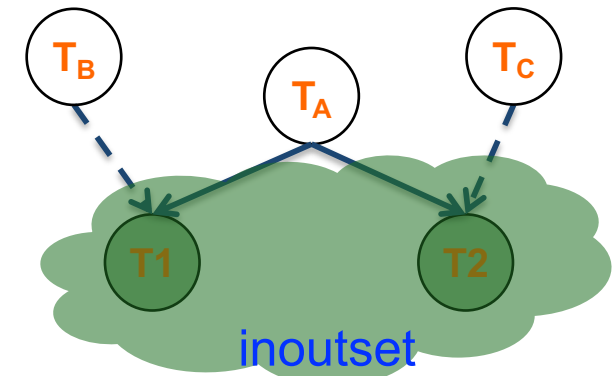
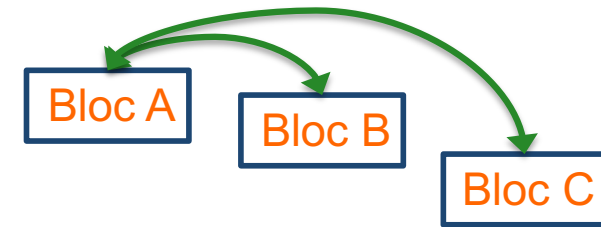
Bloc A interagit avec les blocs B, C, D

en mode RW et mutuel

1 tâche sur chaque bloc  $T_A$ ,  $T_B$ ,  $T_C$

Interaction = 1 tâche

Pas d'ordre dans les tâches  $T_1$ ,  $T_2$



```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(inout: A, B)

    // task T1
    #pragma omp task depend(inout: A, C)

    // task T2
}
```

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(inout: B) \
        depend(mutexinoutset: A)

    // task T1
    #pragma omp task depend(inout: C) \
        depend(mutexinoutset: A)

    // task T2
}
```

# Exécution mutuellement exclusive avec dépendances (3)

## Inouset & mutexinoutset

1. les tâches avec une dépendance `mutexinoutset` créent un nuage de tâches appelé (inouset)
2. Les tâches à l'intérieur de l'ensemble inouset peuvent être exécutées dans n'importe quel ordre mais avec une exclusion mutuelle.



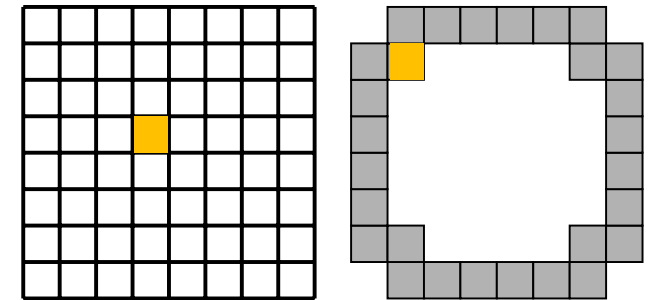
# Nombre dynamique de dépendances

Exemple :

Bloc **A** interagit avec ses blocs voisins

Le nombre de voisin dépend des données initiales

Vecteur de dépendances (variable **list**)




grille dense, grille creuse

```
n = list.size() // longueur de la liste
#pragma omp task depend(iterator(it = 0:n), inout: list[it])
{
    // task sur le bloc A
}
```

équivalent à  
`depend(inout: list[0], list[1], ..., list[n-1])`

# Exemple

```
void parallel_computation(int n) {  
    int v[n];  
    #pragma omp parallel  
    #pragma omp single  
    {  
        int i;  
        for (i = 0; i < n; ++i) {  
            #pragma omp task depend(out: v[i])  
            set_an_element(&v[i], i);  
        }  
        #pragma omp task depend(iterator(it = 0:n), in: v[it])  
        print_all_elements(v, n);  
    }  
}
```



équivalent à  
depend(in: v[0], v[1], ..., v[n-1])

# Fonctions avancées : objets dépendants (1)

## Gérer manuellement les dépendances

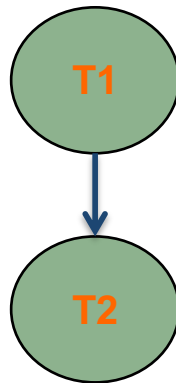
- Utile pour les dépendances de tâches complexes.
- Permet une allocation plus efficace des dépendances de tâches.
- Nouveau type opaque `omp_depend_t`
- 3 nouvelles constructions pour gérer les objets dépendants
  - `#pragma omp depobj(obj) depend(dep-type : list)`
  - `#pragma omp depobj(obj) update(dep-type)`
  - `#pragma omp depobj(obj) destroy`

# Fonctions avancées : objets dépendants (2)

Gérer manuellement les dépendances

```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: x)
  x++; // task T1

  #pragma omp task depend(in: x)
  x+= 2; // task T2
}
```



```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  omp_depend_t obj;
  #pragma omp depobj(obj) depend(inout: x)

  #pragma omp task depend(depobj: obj)
  x++; // task T1
  #pragma omp depobj(obj) update(in)
  #pragma omp task depend(depobj: obj)
  x+= 2; // task T2
  #pragma omp depobj(obj) destroy
}
```

# Réduction

## **NOTIONS AVANCÉES**

# Réduction définie par l'utilisateur

On peut définir sa propre fonction de réduction

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

où :

**identifier** est le nom de la fonction

**typelist** est la liste 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

$$\text{Prod} = v_0 * v_1 * \dots * v_n$$

```
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;
}
```

# Réduction avec les tâches via taskgroup

## Opération de réduction

### 1. La portée de réduction (taskgroup)

```
#pragma omp taskgroup task_reduction(op:list)
```

[1] enregistre une réduction

[3] calcule le résultat finale

### 2. Enrôlement des tâches dans la réduction

```
#pragma omp task in_reduction(op:list)
```

[2] la tache participe à la réduction

```
#node *p = listhead ;
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp taskgroup task_reduction(+: res)
        {
            // [1]
            while (p) {
                #pragma omp task in_reduction(+: res) \
                firstprivate(node)
                {
                    // [2]
                    res += p->value;
                }
                p = p->next; }
            }
        }
    }
}
```

Réduction sur une liste



# Réduction avec les tâches

Les clauses de réduction ont été étendues

La portée de réduction (région parallèle)

```
#pragma omp parallel reduction(task,op:list)
```

[1] enregistre une réduction

[2] les tâches implicites participent à la réduction

[4] calcule le résultat finale

Enrôlement des tâches dans la réduction

```
#pragma omp task in_reduction(op:list)
```

[3] la tache participe à la réduction

```
#node *p = listhead ;  
#pragma omp parallel reduction(task,+: res)  
{ // [1], [2]  
#pragma omp single  
{  
#pragma omp taskgroup  
{  
while (p) {  
#pragma omp task in_reduction(+: res) \  
firstprivate(node)  
{ //[3]  
res += p->value;  
}  
p = p->next; }  
} //[3]  
}}}
```

Réduction sur une liste

# 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-identifiant: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  $\leq 0$  ou  $> 0$
- a, b et c sont peut être des alias

Remarque si a et c sont des alias et  $ioff = 2 \rightarrow$  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**

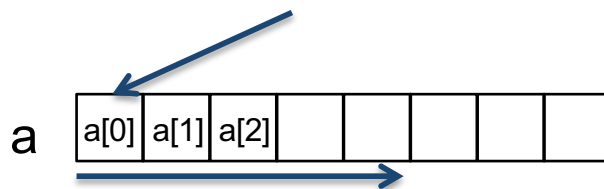
# Clause uniform et linear

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)  $\leftrightarrow$  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);
}
```





# 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

# 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** à **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 explicite
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 1 va sur core 1 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 emplacements 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 (i=0; i<N; i++)  
    { .... Travail sur x[i] ... }
```

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

A blue arrow points from the box to the first loop in the code, indicating that the sequential initialization occurs on the master thread's memory.

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 (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

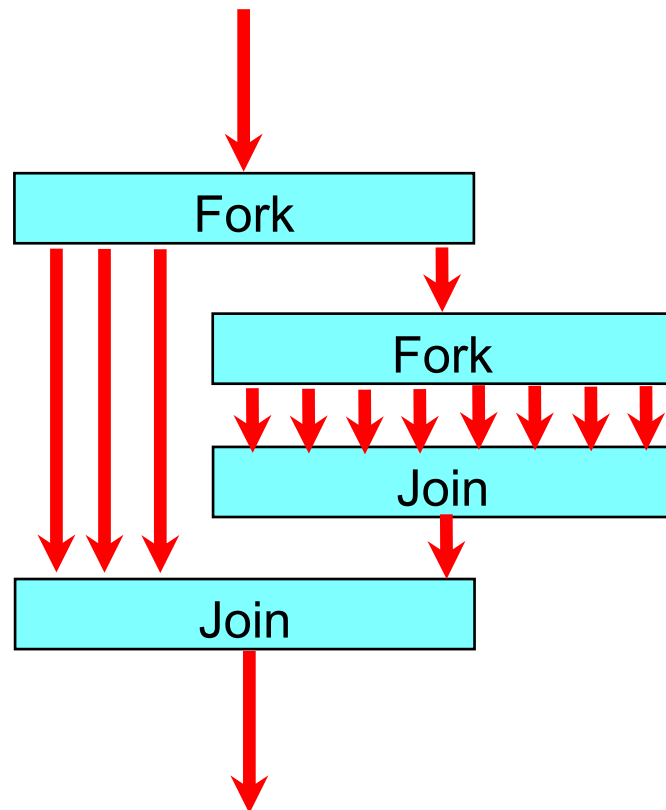
OpenMP a la flexibilité pour permettre les deux sortes de programmation

## Conclusion (2)

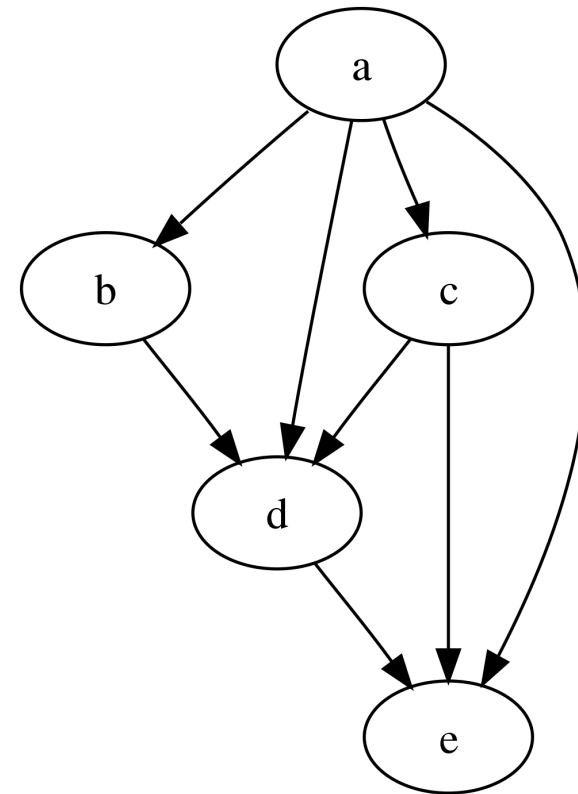
- **Les plus**
  - Facilité de programmation
  - Parallélisation incrémentale
  - Haut niveau d'abstraction
  - Bonne performance (modèle SPMD)
- **Les moins**
  - Des manques (aspect NUMA thread ou data affinity)
  - \* Manque de performances sur les tâches
  - \* Pas d'information sur ce que fait le runtime
- Pour plus de détails : <http://www.openmp.org>

# Deux modèles

Fork and Join



Data flow



**FIN**