

# Introduction à OpenMP

(basé sur diapos de P. Fortin)

Charles Bouillaguet

`charles.bouillaguet@lip6.fr`

11 mars 2021

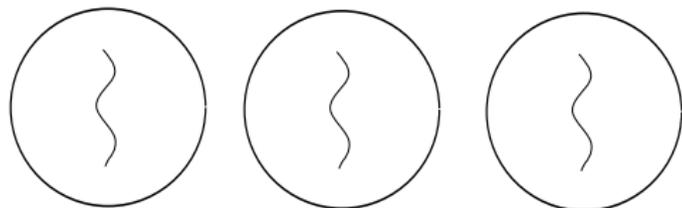
# Threads et processus (rappel)

## Notions de processus et de thread

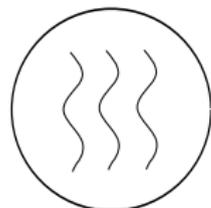
Processus : « flot d'exécution » + « espace mémoire »

Thread : « flot d'exécution »

Éléments propres à chaque processus	Éléments propres à chaque thread
Espace d'adressage	Compteur ordinal
Variables globales	Registres
Fichiers ouverts	Pile (dont variables locales)
Processus enfant, signaux...	Etat



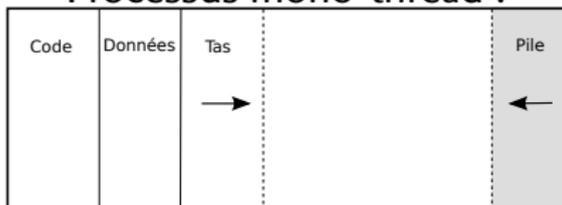
Mode multi-processus



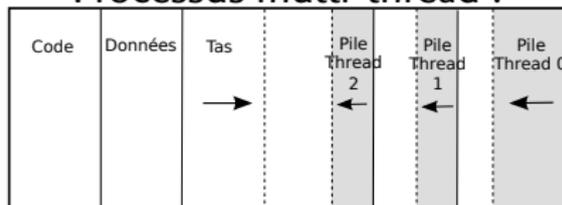
Mode multi-thread

# Threads et processus (rappel)

Processus mono-thread :



Processus multi-thread :



## La mémoire du processus est accessible à tous les threads

- ▶ Variables **partagées**
  - ▶ variables *globales* ou *statiques* (segment de données)
  - ▶ variables du tas avec pointeur connu de tous les threads
- ▶ Variables **privées** (à un thread)
  - ▶ Variables locales (pile),
  - ▶ variables du tas avec pointeur privé

## OpenMP

- 😊 plus facile à programmer / mettre au point que MPI
- 😊 préserve le code séquentiel original
- 😊 code plus facile à comprendre/maintenir
- 😊 permet une parallélisation progressive
- 🙄 machines à **mémoire partagée** uniquement
- 🙄 + adapté à certains types de codes (boucles parallèles ...)

## MPI

- 😊 s'exécute sur machines à mémoire partagée ou distribuée
- 😊 applicable plus largement qu'OpenMP
- 😊 chaque processus a ses propres variables (pas de conflits)
- 🙄 modifications algorithmiques souvent nécessaires
- 🙄 peut être plus dur à mettre au point
- 🙄 performance dépend du réseau de communication

# Historique d'OpenMP

- ▶ 1997, des industriels et des constructeurs adoptent OpenMP (*Open Multi Processing*) comme un *standard industriel*. Interface en Fortran, C et C++
- ▶ Version 2.5 (2000, gcc 4.2) : *lean and mean* (boucles for)
- ▶ Version 3.0 (2008, gcc 4.4) : notion de tâche (*task*)
- ▶ Version 3.1 (2008, gcc 4.7) : raffinements tâche, `atomic`
- ▶ Version 4.0 (2013, gcc 4.9) : SIMD et *devices* (GPU, ...)
- ▶ Version 4.5 (2013, gcc 6) : raffinement SIMD, GPU, ...
- ▶ Version 5.1 (2020, gcc 10) : `atomic compare` + ...
  - ▶ La spec v5.1 est plus dure à lire que la spec v4.5

## Principe de base

- ▶ Un processus unique est exécuté sur une machine parallèle à mémoire partagée. Le thread correspondant est le thread « maître » (de numéro 0).
- ▶ Des parties de programme sont exécutées en parallèle par des threads selon le modèle *fork and join* :



- ▶ La déclaration des zones parallèles se fait à l'aide de directives OpenMP.
- ▶ Modèle mémoire OpenMP particulier : le programmeur peut choisir si une variable est *privée* ou *partagée*.

# Utilisation de OpenMP

- ▶ **Directives de compilation** (#pragma en C)
  - ▶ Interprétées si le compilateur les reconnaît
  - ▶ Ignorées **silencieusement** sinon
  - ▶ Indiquent au compilateur comment paralléliser le code.
  - ▶ gcc : option `-fopenmp`
- ▶ Édition de liens : bibliothèques particulières OpenMP
  - ▶ gcc : option `-fopenmp`
- ▶ Exécution : des **variables d'environnement** peuvent être utilisées pour paramétrer l'exécution

# OpenMP résumé en une diapo

```
// prologue séquentiel  
#pragma omp parallel for  
for (int i = 0 ; i < n ; i++) {  
    /*  
     * ici une boucle où toutes les itérations  
     * peuvent être exécutées en parallèle.  
     */  
}  
// épilogue séquentiel
```

gcc -fopenmp prog\_omp.c -o prog\_omp

- ▶ Programme séquentiel « normal » jusqu'à #pragma omp
- ▶ Une **équipe de threads** est créée
- ▶ Les itérations de la boucle sont **réparties** entre eux
- ▶ **Barrière** à la fin de la boucle for
- ▶ Programme séquentiel « normal » ensuite

# Compilation conditionnelle et fonctions OpenMP

## Compilation conditionnelle

```
#ifdef _OPENMP  
    // Code inclu uniquement si le compilateur supporte OpenMP  
    // Dans gcc, seulement si -fopenmp a été utilisé  
#endif
```

## Fonctions OpenMP

Avec `#include <omp.h>`

- ▶ Permettent un mode de programmation SPMD
- ▶ `omp_get_num_threads()`
- ▶ `omp_get_thread_num()`
- ▶ `omp_set_num_threads()`
- ▶ ...

# Hello world

## Programme

```
#ifdef _OPENMP
#include <omp.h>
#endif

int main()
{
    #pragma omp parallel
    {
        #ifdef _OPENMP
        printf("Hello world thread %d/%d\n",
            omp_get_thread_num(),
            omp_get_num_threads());
        #else
        printf("Hello world\n");
        #endif
    }
}
```

## Exécution

```
$ gcc hello.c -o hello
$ ./hello
Hello world
$ gcc -fopenmp hello.c \
  -o hello
$ export OMP_NUM_THREADS=4
$ ./hello
Hello world thread 0/4
Hello world thread 3/4
Hello world thread 1/4
Hello world thread 2/4
```

# Les directives OpenMP

Elles sont délimitées par une sentinelle :

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

Par défaut, il y a une barrière de synchronisation à la fin.

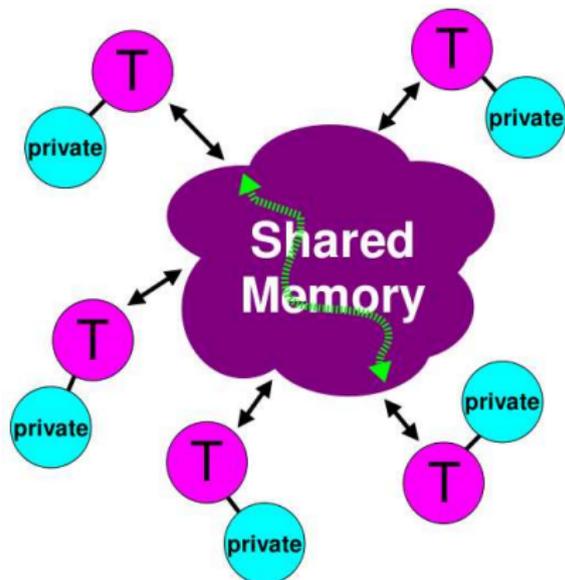
## Utilisation des directives OpenMP :

- ▶ débranchement externe interdit! (`goto`, `setjmp/longjmp`, ...)
- ▶ une seule directive par `#pragma omp`
- ▶ majuscule/minuscule importante
- ▶ les directives sont : `parallel`, `for`, `sections`, `section`, `single`, `master`, `critical`, `barrier`, `atomic`, `flush`, `ordered`, `threadprivate`, ...

# Le modèle mémoire OpenMP

Les variables du code source séquentiel original peuvent être partagées (*shared*) ou privées (*private*) en OpenMP.

- ▶ **Variable partagée** : chaque thread accède à la même et unique variable originale.
- ▶ **Variable privée** : chaque thread a sa propre copie de la variable originale.



(d'après *An Overview of OpenMP 3.0*, R. van der Pas,

IWOMP2009)

## Le modèle mémoire OpenMP (suite)

- ▶ Les variables déclarées **avant** la région parallèle sont **partagées par défaut**
- ▶ On peut modifier leur statut avec `private`, `shared`, `firstprivate`, `lastprivate`, `default(shared)`, `default(none)`, `reduction`, `copyin`
- ▶ Les variables locales à un thread sont privées (cf. infra)

# Directive `parallel`

```
#pragma omp parallel [clause[[, ]clause]...]  
bloc structuré
```

- ▶ Assemble une **équipe de threads** (création/recyclage)
- ▶ Ils exécutent **tous** le *bloc structuré*.

## Liste des clauses associées à cette directive :

- ▶ `if(cond)` : `cond == False` → pas de threads
  - ▶ Par ex. ne pas lancer toute la machine si problème trop petit
- ▶ `private (var_list)`, `firstprivate (var_list)`
- ▶ `reduction` (cf. infra)
- ▶ `num_threads(int)` : force le nombre de threads de l'équipe

```

int main()
{
    ...
    initialisation();
    #pragma omp parallel ...
    {
        calcul()
    }
    post_calcul();
}

```

## Possibilités pour définir le nombre de threads

Par ordre de priorité décroissante :

#pragma	: #pragma omp parallel num_threads(16)
au cours de l'exécution	: <i>omp_set_num_threads(4)</i>
variable d'environnement	: export OMP_NUM_THREADS=4

# Les variables prédéfinies sont partagées par défaut

## Programme

```
#include <omp.h>
#include <stdio.h>

int main()
{
    int c=0;

    #pragma omp parallel
    {
        c++;
        printf("c=%d thread %d\n",
            c, omp_get_thread_num());
    }
}
```

## Exécution

```
$ export OMP_NUM_THREADS=4
$ ./a.out
c=1 thread 3
c=2 thread 0
c=3 thread 1
c=4 thread 2
```

# Attention aux conflits !

## Programme

```
#include <omp.h>
#include <stdio.h>

int main()
{
    int c = 0;

    #pragma omp parallel
    {
        for (int i=0; i<100000; i++)
            c++;
        printf("c=%d thread %d\n",
            c, omp_get_thread_num());
    }
}
```

## Exécution

```
$ export OMP_NUM_THREADS=4
$ ./a.out
c=100000 thread 0
c=200000 thread 3
c=270620 thread 2
c=286162 thread 1
```

# Clause private

Variable private :

- ▶ Chaque thread possède une copie locale (privée)
- ▶ **Non initialisée**

Un exemple de BUG :

## Programme

```
int main()
{
    int a = 100;
    #pragma omp parallel private(a)
    {
        /* Ce "a" n'est pas le
           même qu'au dessus */
        a = a + 10;
        printf("a=%d\n", a);
    }
    printf("Après a=%d\n", a);
}
```

## Exécution

```
$ export OMP_NUM_THREADS=4
$ ./a.out
a=-1208433038
a=-22
a=-22
a=-22
Après a=100
```

# Clause firstprivate

Variable firstprivate :

- ▶ Chaque thread possède une copie locale (privée)
- ▶ Initialisée avec la valeur préexistante

## Programme

```
int main()
{
  int a = 100;
  #pragma omp parallel \
    firstprivate(a)
  {
    a = a + 10;      // idem...
    printf("a=%d\n", a);
  }
  printf("Après a=%d\n", a);
}
```

## Exécution

```
$ export OMP_NUM_THREADS=4
$ ./a.out
a=110
a=110
a=110
a=110
Après a=100
```

# Variables locales

Toutes les variables locales de fonctions appelées depuis une partie parallèle sont locales (privées) aux threads.

Idem pour les variables déclarées dans le bloc parallèle { . . . }.

## Programme

```
void func()
{
    int a = 10;
    a += omp_get_thread_num();
    printf("a=%d\n", a);
}

int main()
{
    #pragma omp parallel
    func();
}
```

## Exécution

```
$ export OMP_NUM_THREADS=4
$ test2
10
11
12
13
```

# Variables locales : mon opinion

## Compliqué (à éviter)

```
int main()
{
    int a;
    #pragma omp parallel private(a)
    {
        a = ...;
        ...
    }
}

/*****/

int main()
{
    int a;
    #pragma omp parallel firstprivate(a)
    {
        ... a ...
    }
}
```

## Simple (à privilégier)

```
int main()
{
    #pragma omp parallel
    {
        int a = ...;
        ...
    }
}

/*****/

int main()
{
    #pragma omp parallel
    {
        int b = a;
        ... b ...
    }
}
```

# Rappels / Précisions

## `#pragma omp parallel`

- ▶ Une **équipe de threads** est créée
- ▶ Celui qui a rencontré la directive en fait partie (**maître**)
- ▶ Tous les threads de l'équipe exécutent le code qui suit
- ▶ Identifiants 0 (maître), 1, 2, ..., #threads - 1
  - ▶ `omp_get_num_threads()`, `omp_get_thread_num()`
- ▶ **Barrière** à la fin
- ▶ Le thread de départ reprend seul l'exécution après

## La directive for

```
#pragma omp for [clause[[, ]clause]...]
< boucle for >
```

### Directive de **partage de travail**

Les threads de l'équipe coopèrent et se **répartissent** le travail

Clauses associées :

- ▶ `private (liste_de_variables), firstprivate (liste_de_variables), lastprivate (liste_de_variables)`
- ▶ `reduction(opérateur : liste_de_variables)`
- ▶ `ordered`
- ▶ `collapse(n)`
- ▶ `schedule(type, taille)`
- ▶ `nowait`

## La directive for (2)

### Forme canonique des boucles

```
for ( expr_init ; expr_logique ; increment )
```

- ▶ L'indice est de type entier.
- ▶ L'incrément est de la forme ++, -- , +=, -=, var=var+inc, var=inc+var, var=var-inc, avec un incrément entier.
- ▶ Test : <, >, <=, >=. La borne est une expression invariante.
- ▶ Pas de sortie prématurée de la boucle (break, return, exit).

Conséquences de la directive for :

- ▶ Barrière implicite à la fin du for (sauf si nowait)
- ▶ Pas de barrière au début
- ▶ **L'indice est une variable privée**

## Exemple

```
int main()
{
    int t[100];
    #pragma omp parallel
    {
        #pragma omp for
        for (int i = 0; i < 100; i++)
            t[i] = i;
    }
}
```

Avec 4 threads, le premier peut **par exemple** calculer les  $t[i]$  de 0 à 24, le second de 25 à 49, ...

# Variable lastprivate

- ▶ Chaque thread possède une copie locale (privée)
- ▶ **Non initialisée**
- ▶ La valeurs obtenue lors la **dernière** itération ( $N - 1$ ) est copiée dans la variable préexistante à la fin de la boucle.

## Programme

```
int a;  
  
#pragma omp parallel  
#pragma omp for lastprivate(a)  
for(int i = 0; i < 4; i++) {  
    a = i * 10;  
    printf("PAR a=%d thread %d \n",  
          a, omp_get_thread_num());  
}  
printf("SEQ a=%d %d\n", a);
```

## Exécution

```
$ export OMP_NUM_THREADS=4  
$ ./a.out  
PAR a=0 thread 0  
PAR a=10 thread 1  
PAR a=20 thread 2  
PAR a=30 thread 3  
SEQ a=30
```

## Forme raccourcie pour la directive for

```
#pragma omp parallel for [clause[[, ]clause]...]
boucle for
```

Cette directive admet toutes les clauses de `parallel` et de `for` à l'exception de `nowait`.

```
#pragma omp parallel
#pragma omp for
for (int i = 0 ; i < 4 ; i++) {
    ....
}
```

```
#pragma omp parallel for
for (int i = 0 ; i < 4 ; i++) {
    ....
}
```

# La clause reduction

## Programme

```
int main()
{
    int a[4][4], s=0;
    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 4; j++)
            a[i][j] = i * 4 + j;
    #pragma omp parallel for reduction(+:s)
    for (int i = 0 ; i < 4 ; i++) {
        for (int j = 0; j < 4; j++)
            s += a[i][j];
        printf("PAR=%d : i=%3d s=%d\n",
            omp_get_thread_num(), i, s);
    }
}
printf("SEQ s=%d\n", s);
}
```

## Exécution

```
$ export OMP_NUM_THREADS=4
$ ./a.out
PAR=0 : i= 0 s=6
PAR=1 : i= 1 s=22
PAR=2 : i= 2 s=38
PAR=3 : i= 3 s=54
SEQ s=120
```

## La clause reduction (suite)

Opérateurs : +, -, \* , &, |, ^, &&, ||, min, max

Autorisé sur omp for, omp parallel, ...

```
int main()
{
    int m = 0;
    #pragma omp parallel reduction(max:m)
    {
        int tid = omp_get_thread_num();
        m = f(tid);
    }
    ...
}
```

## Nouveauté dans réduction



- ▶ réduction sur des *tableaux*
- ▶ ... ou des portions de tableau
- ▶ Depuis OpenMP 4.5 (novembre 2015, gcc  $\geq$  6.1)

```
double *A = malloc(n * sizeof(*A));
...
#pragma omp parallel reduction(+:A[0:n])
{
    // chaque thread possède sa propre version de A[0:n]
    for (int i = 0; i < n; i++) {
        A[i]= ....
    }
    ....
}
// A[i] contient la somme des A[i] privés de chaque thread
```

# Répartition de charge dans les boucles

`omp for` admet des clauses : `schedule` et `nowait`

- ▶ clause `nowait` :  
Par défaut, il y a une synchronisation à la fin de la boucle.
- ▶ clause `schedule(mode, chunk_size)` :  
4 modes : `static`, `dynamic`, `guided`, `runtime`

Par défaut, le choix dépend de l'implémentation d'OpenMP utilisée.

# Exemple schedule

```
#define MAX 10
int main()
{
    int a[MAX];
    #pragma omp parallel
    {
        int imax = 0;
        int imin = MAX;
        #pragma omp for schedule(static)
        for (int i = 0; i < MAX; i++) {
            imin = (i < imin) ? i : imin;
            imax = (i > imax) ? i : imax;
            a[i] = 1;
            sleep(1); /* pour augmenter la charge */
            printf("%3d:%3d\n", omp_get_thread_num(), i);
        }
        printf("T%d imin=%d imax=%d\n", omp_get_thread_num(), imin, imax);
    }
}
```

# Clauses static et dynamic

schedule(...)

static

```
1: 3
3: 8
2: 6
0: 0
1: 4
3: 9
2: 7
0: 1
1: 5
0: 2
T1 imin=3 imax=5
T0 imin=0 imax=2
T2 imin=6 imax=7
T3 imin=8 imax=9
```

static, 2

```
1: 2
3: 6
2: 4
0: 0
1: 3
3: 7
0: 1
2: 5
0: 8
0: 9
T0 imin=0 imax=9
T1 imin=2 imax=3
T3 imin=6 imax=7
T2 imin=4 imax=5
```

dynamic, 2

```
1: 4
0: 0
2: 6
3: 2
1: 5
2: 7
0: 1
3: 3
1: 8
1: 9
T1 imin=4 imax=9
T3 imin=2 imax=3
T2 imin=6 imax=7
T0 imin=0 imax=1
```

## schedule

- ▶ `schedule(static)` : chaque thread a un bloc de la même taille.
- ▶ `schedule(static, n)` :  $n$  indique la taille des paquets (*chunk*). Distribution bloc-cyclique.
- ▶ `schedule(dynamic, n)`, les paquets sont affectés aux premiers threads disponibles.  
Valeur de  $n$  par défaut : 1.
- ▶ `guided` : équilibrage de charge dynamique avec une taille de paquet proportionnelle au nombre d'itérations encore non attribuées divisé par le nombre de threads (taille décroissante vers 1)
- ▶ `auto` : OpenMP se débrouille.
- ▶ `runtime` : le choix est reporté à l'exécution  
Exemple : `export OMP_SCHEDULE="static,1"`

# Directive single

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

## But

définir, dans une région parallèle, une portion de code qui sera exécutée par **un seul** thread

- ▶ Rien n'est dit sur le thread qui exécute la directive
- ▶ Barrière implicite à la fin de `single`
- ▶ `nowait` et `copyprivate` sont incompatibles

Listes des clauses possibles :

- ▶ `private` (*liste\_de\_variables*), `firstprivate` (*liste\_de\_variables*), `copyprivate` (*liste\_de\_variables*)
- ▶ `nowait`

# Directive single

## Programme

```
int main()
{
    int a = 10;
    #pragma omp parallel firstprivate(a)
    {
        #pragma omp single
        a = 20;

        printf("thread %d a=%d\n",
              omp_get_thread_num(), a);
    }
}
```

## Exécution

```
$ export OMP_NUM_THREADS=4
$ ./a.out
thread 0 a=20
thread 1 a=10
thread 3 a=10
thread 2 a=10
```

## Directive master

```
#pragma omp master  
bloc structuré
```

- ▶ Pas de clause
- ▶ **Pas de barrière implicite**
- ▶ Seul le thread 0 (master) exécute le code associé

# Synchronisations en OpenMP

Plusieurs possibilités :

- ▶ barrières
- ▶ directives `atomic` et `critical`
- ▶ verrous via fonctions OpenMP (non traitées ici) :
  - `omp_init_lock()`
  - `omp_{set,test}_lock()`
  - `omp_unset_lock()`
  - `omp_destroy_lock()`

## Directive barrier

```
#pragma barrier
```

Lorsqu'un thread rencontre une barrière, il attend que tous les autres soient arrivés à ce même point.

### Problèmes de syntaxe C :

```
if (n != 0)
    #pragma omp barrier // syntaxiquement incorrect

if (n != 0) {
    #pragma omp barrier // OK
}
```

## Directive atomic

```
#pragma omp atomic  
expression-maj
```

- ▶ La mise à jour est **atomique**
- ▶ L'effet ne concerne que l'instruction suivante
- ▶ *expression-maj* est de la forme :
  - ▶ ++x, x++, --x, x--
  - ▶ x binop= expr;  
x = x binop expr;  
x = expr binop x;
    - ▶ binop ∈ {+, \*, -, /, &, &&, ^, |, ||, >>, <<},
    - ▶ l'expression *expr* ne doit pas faire référence à x.
- ▶ Seul le chargement et la mise à jour de la variable forment une opération atomique, l'évaluation de l'expression *expr* ne l'est pas.

## Programme

```
#include <omp.h>

int main()
{
    int c = 0;
    #pragma omp parallel
    {
        for (int i = 0; i < 100000; i++) {
            #pragma omp atomic
            c++;
        }
        printf("c=%d thread %d\n",
            c, omp_get_thread_num());
    }
}
```

## Exécution

```
$ export OMP_NUM_THREADS=4
$ ./a.out
c=100000 thread 0
c=294308 thread 2
c=394308 thread 3
c=400000 thread 1
```

# Directive atomic avancée

depuis OpenMP 3.0 (2008)

```
#pragma omp atomic [ read | write | update | capture ]  
expression-maj  
ou :  
#pragma omp atomic capture  
bloc-structuré
```

- ▶ La clause `update` est équivalente à l'absence de clause
- ▶ `Capture` = mise à jour + récupérer l'ancienne valeur
- ▶ Pour la clause `capture`, `expression-maj` peut être :

`v = x++;`

`v = x--;`

`v = ++x;`

`v = --x;`

`v = x binop= expr;`

`v = x = x binop expr;`

`v = x = expr binop x;`

où  $x$ ,  $v$  sont des expressions *l-values* de type scalaire, et `expr` est une expression de type scalaire (voir autres contraintes sur slide précédent et dans la norme).

## Directive `atomic` avancée (suite) (depuis OpenMP 3.0)

- Pour la clause `capture` (suite), bloc-structuré peut être :

```
{v = x; x binop= expr;}
```

```
{x binop= expr; v = x;}
```

```
{v = x; x = x binop expr;}
```

```
{v = x; x = expr binop x;}
```

```
{x = x binop expr; v = x;}
```

```
{x = expr binop x; v = x;}
```

```
{v = x; x = expr;}
```

```
{v = x; x++;}
```

```
{v = x; ++x;}
```

```
{++x; v = x;}
```

```
{x++; v = x;}
```

```
{v = x; x--;}
```

```
{v = x; --x;}
```

```
{--x; v = x;}
```

```
{x--; v = x;}
```

## Directive critical

```
#pragma omp critical [nom]  
bloc structuré
```

- ▶ Un seul thread à la fois peut exécuter le *bloc structuré*
- ▶ « **section critique** »
- ▶ Le thread est bloqué à l'entrée du bloc structuré tant qu'un autre thread exécute un bloc portant le même nom.
- ▶ Le nom est utile pour l'édition multi-fichiers et pour distinguer des sections critiques indépendantes.

# Exemple de directive critical

Ajout d'un nouvel élément au début d'une liste chaînée

```
struct item_t {
    int val;
    struct item_t *next;
};
...
struct item_t *list = (struct item_t *) malloc(sizeof(*list));
...
#pragma omp parallel
{
    ...
    (struct item_t *) nouv = malloc(sizeof(*nouv));
    nouv->val = ...
    #pragma omp critical
    {
        nouv->next = list;
        list = nouv;
    }
    ...
}
```

## Différence atomic, critical

- `atomic` :
- ▶ Destinée à la mise à jour de variables
  - ▶ Dépend du matériel et de l'OS
    - ▶ Instructions atomiques du processeur

- `critical` :
- ▶ Destinée à englober une partie plus importante de code
  - ▶ Implémentation avec des verrous

## Différence atomic, critical

- `atomic` :
- ▶ Destinée à la mise à jour de variables
  - ▶ Dépend du matériel et de l'OS
    - ▶ Instructions atomiques du processeur
  - ▶ Plus léger a priori

- `critical` :
- ▶ Destinée à englober une partie plus importante de code
  - ▶ Implémentation avec des verrous
  - ▶ Plus lourd

## Différence atomic, critical

- `atomic` :
- ▶ Destinée à la mise à jour de variables
  - ▶ Dépend du matériel et de l'OS
    - ▶ Instructions atomiques du processeur
  - ▶ Plus léger a priori
  - ▶ À privilégier si possible.

- `critical` :
- ▶ Destinée à englober une partie plus importante de code
  - ▶ Implémentation avec des verrous
  - ▶ Plus lourd

# Règle d'or de la programmation multithreads

**Tous** les accès potentiellement conflictuels\* aux variables **partagées** doivent être protégés (`atomic`, `critical`, ...).

\* au moins l'un d'entre eux est une écriture.

## Directive `threadprivate`

Permet de définir le statut des variables globales ou statiques dans les threads (usage pas très fréquent).

Une variable `threadprivate` ne doit pas apparaître dans une autre clause sauf pour `copyin`, `copyprivate`, `schedule`, `num_thread`, `if`.

Une variable `threadprivate` ne doit pas être une référence (C++).

# Directive threadprivate

## Programme

```
int a = 10; // variable globale
#pragma omp threadprivate(a)

int main()
{
    #pragma omp parallel
    {
        int rang = omp_get_thread_num();
        a += rang;
        printf("thd=%d a=%d\n", rang, a);
    }
    printf("SEQ a=%d\n",a);
}
```

## Exécution

```
$ export OMP_NUM_THREADS=4
$ ./a.out
thd=0 a=10
thd=3 a=13
thd=1 a=11
thd=2 a=12
SEQ a=10
```

## Parallélisme imbriqué (nesting)

- ▶ une directive `parallel` dans une directive `parallel`
- ▶ un certain nombre de règles sont à respecter (cf. spec.)
- ▶ une variable d'environnement : `OMP_NESTED` doit prendre la valeur `TRUE` (ou `FALSE`) pour autoriser (ou non) le parallélisme imbriqué (non autorisé par défaut).

### Mon opinion

- ▶ **AUCUN INTÉRÊT** à le faire volontairement
- ▶ Peut arriver si un code parallèle invoque une librairie parallèle

# Bibliothèque OpenMP

Quelques fonctions courantes (avec `#include <omp.h>`) :

- ▶ `void omp_set_num_threads(int num_thread)` : fixe le nombre de threads utilisables par le programme. Ne pas appeler dans une région parallèle.
- ▶ `int omp_get_num_threads(void)`
- ▶ `int omp_get_max_threads(void)` : retourne le nombre de threads maximum qui sera utilisé pour la prochaine région parallèle (valable si la clause `num_threads` n'est pas utilisée).
- ▶ `int omp_get_thread_num(void)`
- ▶ `int omp_get_num_procs(void)`
- ▶ `int omp_in_parallel(void)` : retourne un entier non nul si l'appel a lieu dans une région parallèle.
- ▶ `double omp_get_wtime(void)` la différence entre deux appels permet de calculer le *wall-clock time* en secondes.

# Les variables d'environnement

- ▶ OMP\_NUM\_THREADS
- ▶ OMP\_SCHEDULE :  
export OMP\_SCHEDULE="static,4"  
export OMP\_SCHEDULE="dynamic"
- ▶ OMP\_NESTED
- ▶ ...

# Cheat Sheet

## Le plus courant

- ▶ Le *must* : `#pragma omp parallel for`
- ▶ Directive `atomic` et `critical`
- ▶ Clauses `schedule` et `reduction`

## Moins courant : mode SPMD

- ▶ Région parallèle avec `#pragma omp for` dedans
- ▶ `omp_get_thread_num()` et `omp_get_num_threads()`
- ▶ Directives `barrier`, `master` ou `single`

Plus rare (réservé aux cas durs mais pas seulement...)

Tout le reste !

# Tâches OpenMP (depuis v3.0, 2008)

## Point d'orthographe

**Tache** marque, salissure, souillure. *Une tache d'encre.*

**Tâche** travail à exécuter. *Une tâche ardue.*

# Tâches OpenMP (depuis v3.0, 2008)

## Point d'orthographe

**Tache** marque, salissure, souillure. *Une tache d'encre.*

**Tâche** travail à exécuter. *Une tâche ardue.*

## « Rappel » : `#pragma omp parallel`

- ▶ Une **équipe de threads** est créée.
- ▶ ...
- ▶ Une **tâche implicite** est créée pour chaque thread.
- ▶ Elle lui est **liée** (peut pas migrer sur un autre thread)

**tâche** = code + données associées, le tout exécuté par un thread.

## Directive omp task

```
#pragma omp task [clause], [clause], ...  
bloc structuré
```

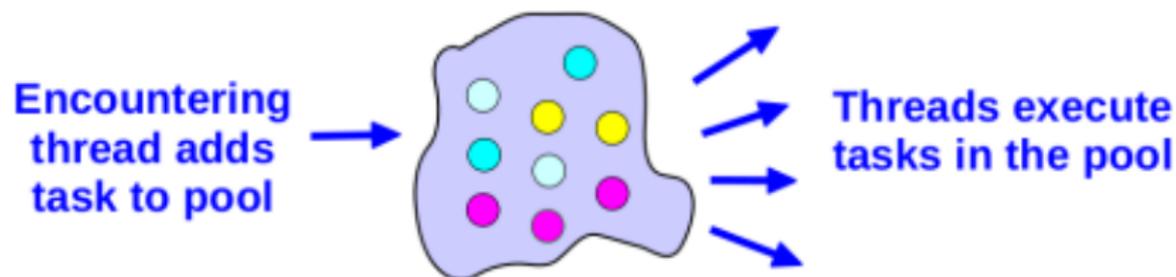
- ▶ Le thread qui rencontre ceci crée une **tâche explicite**.
- ▶ Exécution **immédiate** ou **différée**.
- ▶ Exécution par un des threads de l'équipe... si disponible.

## Directive omp task

```
#pragma omp task [clause], [clause], ...  
bloc structuré
```

- ▶ Le thread qui rencontre ceci crée une **tâche explicite**.
- ▶ Exécution **immédiate** ou **différée**.
- ▶ Exécution par un des threads de l'équipe... si disponible.
- ▶ Un thread peut suspendre l'exécution d'une tâche et en démarrer/reprendre une autre...
- ▶ ... Mais seulement lors d'un *task scheduling points*

## « Task Pool »



***Developer specifies tasks in application  
Run-time system executes tasks***

(An Overview of OpenMP 3.0, R. van der Pas, IWOMP2009)

# Synchronisation des tâches

Modèle *Fork/Join*

## Barrières « normales »

- ▶ Implicites : à la fin d'une région parallèle, de `omp for`, ...
- ▶ Explicites : **#pragma omp barrier**

**Garantie** : Toutes les tâches créées par un thread de l'équipe courante sont terminées à la sortie de la barrière.

## Barrière de tâches : **#pragma omp taskwait**

- ▶ La tâche courante attend la terminaison de ses tâches **filles**.
- ▶ Seulement filles directes, pas descendantes.

## Exemple : parcours d'une liste chaînée

```
struct item_t {  
    void * data;  
    struct item_t * next;  
};  
struct item_t * head;
```

### Séquentiel

```
struct item_t * e = head;  
  
while (e != NULL) {  
    process(e->data);  
    e = e->next;  
}
```

### Avec tâches

```
struct item_t * e = head;  
#pragma omp parallel  
#pragma omp single  
while (e != NULL) {  
    #pragma omp task  
    process(e->data);  
    e = e->next;  
}
```

# Exemple : descente dans un arbre binaire

```
struct tree_t {  
    ...  
    struct tree_t *left, *right;  
};  
struct tree_t *root;
```

## Séquentiel

```
void parcours(struct tree_t *t)  
{  
    ...  
    if (t->left)  
        parcours(t->left);  
    if (t->right)  
        parcours(t->right);  
}  
  
parcours(root);
```

## Avec tâches

```
void parcours(struct tree_t *t)  
{  
    ...  
    if (t->left)  
        #pragma omp task  
        parcours(t->left);  
    if (t->right)  
        #pragma omp task  
        parcours(t->right);  
}  
  
#pragma omp parallel  
#pragma omp single  
parcours(root);
```

## Directive omp task

```
#pragma omp task [clause], [clause], ...  
bloc structuré
```

Clauses associées :

- ▶ private (*liste\_de\_variables*), firstprivate (*liste\_de\_variables*), shared (*liste\_de\_variables*)
- ▶ default(shared | none)
- ▶ untied
- ▶ depend(*dependance-type* : *list*)
- ▶ if(*expression*)

## Tâches : portée des variables

- ▶ Le plus utile avec les tâches : **firstprivate**
- ▶ Attribut **firstprivate** par défaut sur toutes les variables...
- ▶ **sauf** si elles sont déjà considérées comme **shared**
  - ▶ Variable globale
  - ▶ Variable déclarée avant la région parallèle
  - ▶ Variable explicitement marquée comme `shared`

### Attention aux variables partagées sur la pile

```
void f()
{
    int i = 3;
    #pragma omp task shared(i)
    printf("%d\n", i);
}

#pragma omp parallel
#pragma omp single
f();
```

## Tâches : cas où shared est a priori nécessaire

```
struct tree_t {
    ...
    struct tree_t *left, *right;
};
struct tree_t *root;

/* Renvoie le nombre de noeuds de l'arbre. */
int size(struct tree_t *t)
{
    int s_left = 0, s_right = 0;
    if (t->left)
        #pragma omp task shared(s_left)
        s_left = size(t->left);
    if (t->right)
        #pragma omp task shared(s_right)
        s_right = size(t->right);
    #pragma omp taskwait
    return s_left + s_right;
}
#pragma omp parallel
#pragma omp single
printf("%d\n", size(root));
```

# Tâches : ordonnancement

## Liaison tâche ↔ thread

- ▶ Par défaut les tâches sont **tied** (liées).
- toujours exécutées par le même thread (celui qui les a créés).
- ▶ Tâche suspendue seulement aux *task scheduling points* : création/terminaison de tâche, barrière, `taskwait`, `taskyield`.

Problème potentiel : déséquilibre de charge.

## Clause **untied**

- ▶ La tâche peut passer d'un thread à l'autre lors d'un *task scheduling point*.
- ▶ Attention aux variables `threadprivate`.
- ▶ Attention à l'indice du thread.
- ▶ Attention aux sections critiques.

# Tâches : granularité

## Créer une tâche a un coût non-trivial

### Ne pas créer des tâches minuscules

- ▶ *Clause if* de la directive `omp task`
  - ▶ `#pragma omp task if (prof < PROF_MAX)`
  - ▶ La tâche est **quand même créée...**
  - ▶ ...mais exécutée tout de suite par le thread courant

- ▶ *Instruction if :*

```
if (prof < PROF_MAX) {  
    #pragma omp task  
    stuff(...);  
} else {  
    stuff(...);  
}
```

→ à privilégier a priori

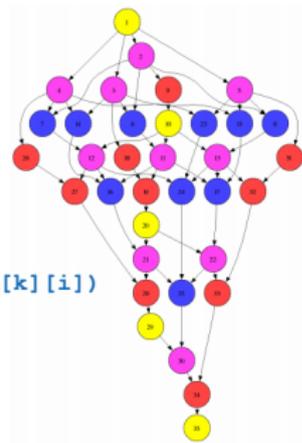
## Tâches : dépendances

TODO...

# Tâches : dépendances

## Exemple de la mort (Christian Terboven)

```
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]);
    }
  }
}
```



\* image from BSC