



Mineure CalHau1

## L'essentiel d'OpenMP

Stéphane Vialle



Stephane.Vialle@centralesupelec.fr  
<http://www.metz.supelec.fr/~vialle>

## Programmation OpenMP (par partage de mémoire)

1. Principes d'OpenMP
2. Détails de syntaxe et de sémantique
3. Optimisations
4. Exemples de parallélisation
5. Bilan d'OpenMP
6. Mesures et représentation de performances

## Principes de base

### Développer un code séquentiel :

- conception
- implantation
- mise au point

```
Initialisation();
for (i=0; i<N; i++)
    Calcul(i);
Autre_calcul();
```



### Ajouter des directives de compilation parallèles

- **parallélisation incrémentale**
- peu de code supplémentaire
- limité au parallélisme présent dans le code initial

Exemple  
utilisant une  
« directive »  
OpenMP

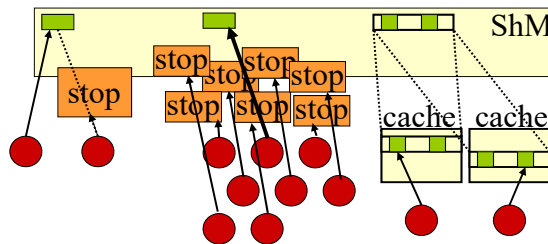
```
#include <omp.h>
Initialisation();
#pragma omp parallel for private(i)
for (i=0; i<N; i++)
    Calcul(i);
Autre_calcul();
```

## Principes de base

### OpenMP : basé sur des *threads* et soumis aux mêmes limites

Exploite une mémoire partagée, mais :

- pb de synchronisation
- pb de contention
- pb de *false sharing*

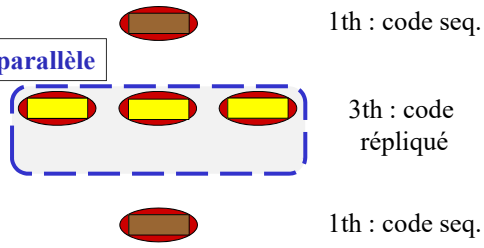


→ Utiliser OpenMP facilite (souvent) l'implantation, mais ne dispense pas de résoudre les problèmes d'algorithmique parallèle en mémoire partagée.

## Programmation OpenMP - principes

# Régions parallèles

```
main() {
  .....
  #pragma omp parallel
  {
    .....
  }
  .....
}
```



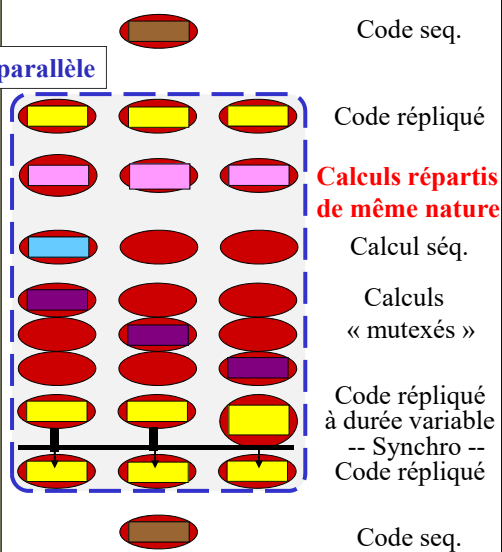
### Ex sur une machine à 3 cœurs (!!):

- Création et destruction de 3 threads en début et fin d'une région parallèle
- « Join » (synchro) automatique sur la mort des 3 threads en sortie de région parallèle
- Syntaxe très simple et concise

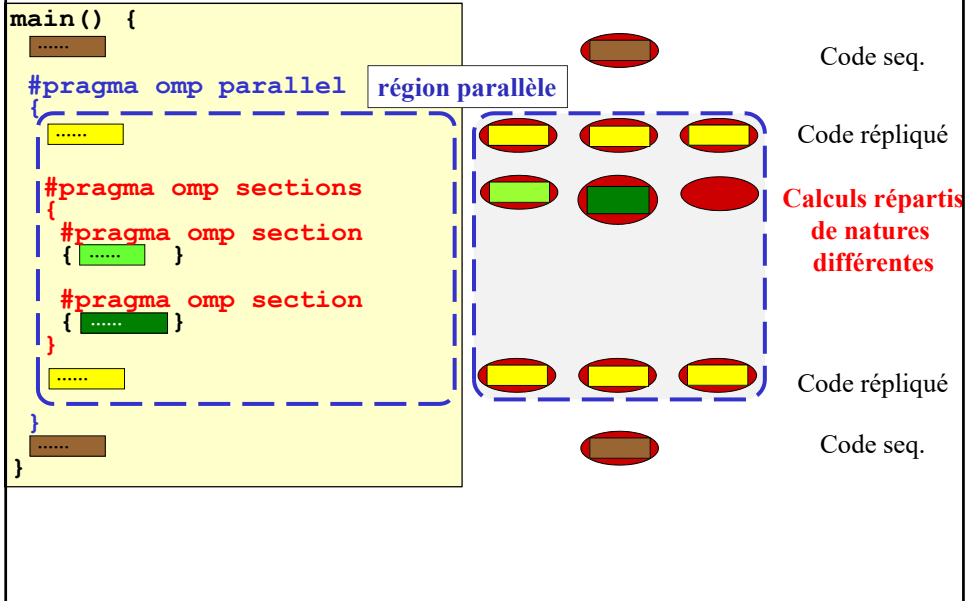
## Programmation OpenMP - principes

# Ctrl du parallélisme par directives

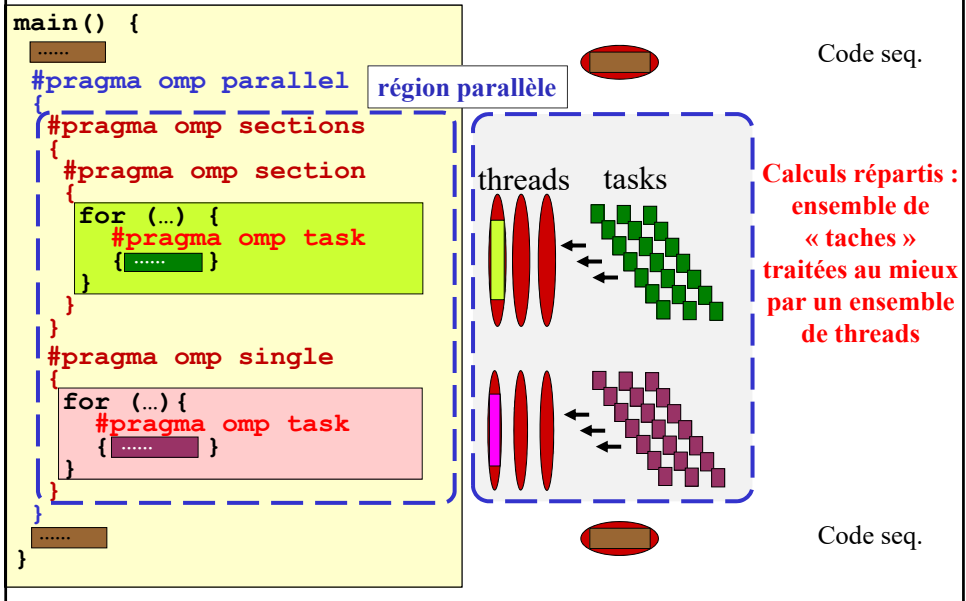
```
main() {
  .....
  #pragma omp parallel
  {
    .....
    #pragma omp for private(i)
    for (i=0; i<N; i++){
      .....
    }
    #pragma omp single
    { ..... }
    #pragma omp critical
    { ..... }
    .....
    #pragma omp barrier
    .....
  }
  .....
}
```



# Ctrl du parallélisme par directives



# Ctrl du parallélisme par directives



# Ctrl explicite du parallélisme

Parallélisation d'un appel de fonction séquentielle :

```

Code parallèle ←
main() {
    .....
    #pragma omp parallel
    {
        f_lib(N/omp_get_num_threads()* // Borne inférieure
              omp_get_thread_num(),
              N/omp_get_num_threads()* // Borne supérieure
              (omp_get_thread_num()+1), // (exclue)
              SharedTable);           // Table à traiter
    }
    .....
}

Code séquentiel
main() {
    .....
    f_lib(0, N, SharedTable);
    .....
}
    
```

→ code répliqué MAIS avec des paramètres spécifiques à chaque thread

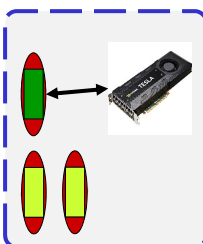
Rmq : il faut que le code de la fonction soit « ré-entrant »

# Ctrl explicite du parallélisme

```

main() {
    .....
    #pragma omp parallel
    {
        switch (omp_get_thread_num()) {
        case 0 :
            ..... // calcul sur // le GPU
            break;
        default :
            ..... // calcul sur // les cœurs CPU
            break;
        }
    }
    .....
}
    
```

Hyp : 3 threads OpenMP créés sur une machine à 3 cœurs CPU...



...et l'un des cœurs CPU est dédié au pilotage du GPU

Faire accomplir au thread 0 (qui existe toujours) une tâche spéciale

Ex : pilotage d'un GPU et calculs sur les autres cœurs CPU...

recouvrement d'IO et de calculs sur les autres cœurs CPU...

## Région parallèles de directives orphelines

```
main() {
  int i, j;
  #pragma omp parallel
  {
    #pragma omp for private(i)
    for (i=0; i<N; i++)
    .....
    #pragma omp for private(j)
    for (j=0; j<Q; j++)
    .....
  }
}
```

Région parallèle explicite

Région parallèle d'une directive « orpheline »

```
main() {
  int i, j;
  #pragma omp parallel for private(i)
  for (i=0; i<N; i++)
  .....
  #pragma omp parallel for private(j)
  for (j=0; j<Q; j++)
  .....
}
```

### Directive orpheline :

- plus simple, plus compacte,
- amène sa propre **région parallèle minimale**,
- provoque la création et la mort automatique de threads à chaque exécution (possible pb de perf, voir plus loin)...

## Programmation OpenMP (par partage de mémoire)

1. Principes d'OpenMP
2. **Détails de syntaxe et de sémantique**
3. Optimisations
4. Exemples de parallélisation
5. Bilan d'OpenMP
6. Mesures et représentation de performances

## Parallel for

```
main() {
    .....
    #pragma omp parallel
    {
        .....
        #pragma omp for private(i)
        for (i=0; i<N; i++){
            .....
        }
        .....
    }
    .....
}
```

OpenMP peut paralléliser des boucles de calculs **ssi les itérations sont indépendantes.**

Sinon, l'algorithme n'est pas parallélisable, et il faut en changer !

Algorithme et code séquentiels

```
for (i = 1; i < N; i++) {
    tab[i] = 1 + (tab[i-1])2
}
```

Algorithme et code parallélisables

```
for (i = 1; i < N; i++) {
    tab2[i] = 1 + (tab1[i-1])2
}
```

## Parallel for

**Création de variables privées aux threads : clause « private »**

```
#define N 1000
double Tab1[N], Tab2[N];
main() {
    int i;
    double alpha = 2.0;
    .....
    #pragma omp parallel for private(i) \
                             shared(Tab1, Tab2) \
                             shared(alpha)
    for (i = 0; i < N; i++) {
        double coef; // var locale (privée)
        coef = pow(Tab1[i], alpha);
        Tab2[i] *= coef;
    }
}
```

Les compteurs de boucles déclarés en amont des boucles des threads doivent être transformés en variables privées.

*Rmq : fait par le compilateur en théorie...*

## Parallel for

### Création de variables privées aux threads : def. de variables locales

```
#define N 1000
double Tab1[N], Tab2[N];
main() {
    double alpha = 2.0;
    .....
    #pragma omp parallel for shared(Tab1,Tab2) \
                             shared(alpha)
    for (int i = 0; i < N; i++) {
        double coef; // var locale : privée
        coef = pow(Tab1[i],alpha);
        Tab2[i] *= coef;
    }
}
```

Les compteurs de boucles déclarés dans les boucles sont automatiquement privés à chaque thread

## Parallel for

### Propagation de valeur aux variables privées : clause « firstprivate »

```
#define N 1000
double Tab[2][N];
main() {
    int i, c;
    int idx = 0;
    double a = 2.0;
    #pragma omp parallel \
        private(i,c,idx) \
        shared(Tab) \
        shared(a)
    {
        idx = 0; // NE PAS OUBLIER
        for (c = 0; c < NbC; c++){
            #pragma omp for
            for (i = 0; i < N; i++){
                double coef =
                    pow(Tab[idx][i],a);
                Tab[1-idx][i] *= coef;
            }
            idx = 1 - idx;
        }
    }
    printf("%d",idx); // → 0
}
```

```
#define N 1000
double Tab[2][N];
main() {
    int i, c;
    int idx = 0;
    double a = 2.0;
    #pragma omp parallel \
        private(i,c) \
        firstprivate(idx) \
        shared(Tab) \
        shared(a)
    {
        for (c = 0; c < NbC; c++){
            #pragma omp for
            for (i = 0; i < N; i++){
                double coef =
                    pow(Tab[idx][i],a);
                Tab[1-idx][i] *= coef;
            }
            idx = 1 - idx;
        }
    }
    printf("%d",idx); // → 0
}
```

Voir aussi  
« lastprivate »



## Parallel reduce

**Def :** une « réduction » passe d'un vecteur de dimension  $n$  à un vecteur de dimension  $m$  ( $< n$ ). Ex : un vecteur  $\rightarrow$  un scalaire.

Le « `parallel for` » d'OpenMP permet aussi de faire des réductions :

```
double x[N], y[N];
double Sx = 0.0, Sy = 0.0;
double MoyX, MoyY;
int i;

#pragma omp parallel for private(i)
                        shared(x, y)
                        reduction(+: Sx, Sy)
for (i = 0; i < N; i++) {
    Sx = Sx + x[i];      // local Sx
    Sy = Sy + y[i];      // local Sy
}
// global Sx = sum of all local Sx (idem Sy)
MoyX = Sx/N;           // global Sx
MoyY = Sy/N;           // global Sy
```

Cohérence de l'opérateur et des opérations : à la charge du développeur

## Contrôle de l'ampleur du parallélisme

Par défaut OpenMP *inonde* le nœud : un thread par cœur logique !  
Mais on peut spécifier le nbr de threads créés

**Sol 1 :** spécifier le nbr de threads à chaque entrée en région parallèle

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

.....

#pragma omp parallel for num_threads(10)
for ...

.....

#pragma omp parallel for num_threads(2)
for ...
```

Impose le nombre de threads créés **juste** sur une **portion de code** OpenMP

Commande à **re-spécifier à chaque** région parallèle (non robuste)

## Contrôle de l'ampleur du parallélisme

Par défaut OpenMP *inonde* le nœud : un thread par cœur logique !  
 Mais on peut spécifier le nbr de threads créés

**Sol 2 : spécifier le nbr de threads jusqu'à une nouvelle spécification**

```

omp_set_num_threads(10);
#pragma omp parallel {
    .....
}

.....

#pragma omp parallel for
for ...

.....
omp_set_num_threads(2);
#pragma omp parallel for
for ...
    
```

Impose le nombre de threads pour tout le reste du programme  
 Mais une nouvelle spécification écrase la précédente

## Contrôle de l'ampleur du parallélisme

Par défaut OpenMP *inonde* le nœud : un thread par cœur logique !  
 Mais on peut spécifier le nbr de threads créés

**Sol 3 : les deux types de spécifications sont compatibles**

```

omp_set_num_threads(10);
#pragma omp parallel {
    .....
}

.....

#pragma omp parallel for
for ...

.....

#pragma omp parallel for num_threads(2)
for ...
    
```

Impose le nombre de threads pour tout le reste du programme  
 Mais une nouvelle spécification écrase la précédente

# Qu'imprime ce programme ?

```

omp_set_num_threads(10);
printf("Thread %d/%d\n",
      omp_get_thread_num(),
      omp_get_num_threads());

#pragma omp parallel num_threads(2)
{
  printf("Thread %d/%d\n",
        omp_get_thread_num(),
        omp_get_num_threads());

  #pragma omp for private(i,j)
  for (i = 0; i < 4; i++)
    for (j = 0; j < 2; j++)
      print("%d-%d-%d\n",
            omp_get_thread_num(), i, j);
}

printf("Thread %d/%d\n",
      omp_get_thread_num(),
      omp_get_num_threads());

```

Thread 0/1

Thread 0/2 Thread 1/2  
ou bien

Thread 1/2 Thread 0/2

th = 0 :      th = 1

0-0-0	1-2-0
0-0-1	1-2-1
0-1-0	1-3-0
0-1-1	1-3-1

Thread 0/1

# Mesure du temps écoulé

```

start = omp_get_wtime();
#pragma omp parallel \
  private(cycle,i,j)
{
  ..... // PARALLEL COMPUTATIONS
}
finish = omp_get_wtime();
duration = finish - start;
gigaflops = NbrOPs/duration/1E9;
.....

```

Début de la mesure de temps

Fin de la mesure de temps

Déduction de la vitesse de calcul

**omp\_get\_wtime() :**

- Portable sous Linux et Windows
- Mesure le temps qui passe (le « wall clock »)
- Est assez précise
- Est très facile à utiliser

## Synchronisation par « locks » explicites

Exemple d'utilisation des fonctions d'OpenMP :

<code>int compteur = 0;</code>	Déclaration d'un verrou
<code>omp_lock_t Lock</code>	
<code>omp_init_lock(&amp;Lock);</code>	Initialisation du verrou
<code>#pragma omp parallel</code>	
<code>{</code>	
<code>.....</code>	
<code>omp_set_lock(&amp;Lock);</code>	Utilisation du verrou
<code>compteur++ // accès à une rsrc critique</code>	
<code>omp_unset_lock(&amp;Lock);</code>	
<code>.....</code>	
<code>}</code>	Libération du verrou
<code>.....</code>	
<code>omp_destroy_lock(&amp;Lock);</code>	

Permet de réaliser une synchronisation par verrous, qui est portable entre Windows et Linux.

## Fonctions disponibles en OpenMP

### Execution Environment Functions

```
omp_set_num_threads
omp_get_num_threads
omp_get_max_threads
omp_get_thread_num
omp_get_num_procs
```

```
omp_in_parallel
omp_set_dynamic
omp_get_dynamic
omp_set_nested
omp_get_nested
```

### Lock Functions

```
omp_init_nest_lock
omp_destroy_nest_lock
omp_set_nest_lock
omp_unset_nest_lock
omp_test_nest_lock
```

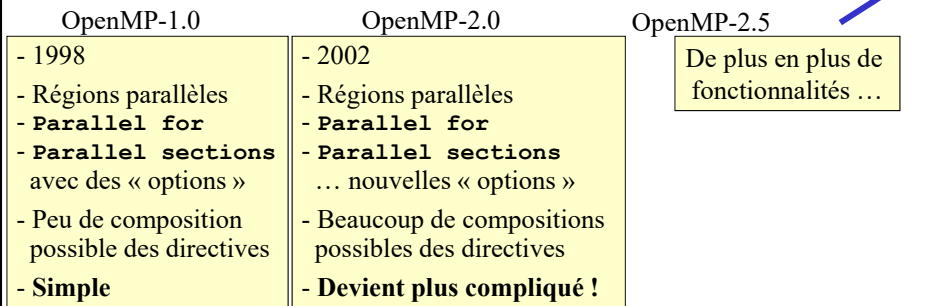
```
omp_init_lock
omp_destroy_lock
omp_set_lock
omp_unset_lock
omp_test_lock
```

### Timing Routines

```
omp_get_wtime
omp_get_wtick
```

## Evolution d'OpenMP

### OpenMP-1 vs OpenMP-2 (ou 2.5 ou 3.1 ou 4.0)



Evolution de la documentation de l'API :

OpenMP -1.0	OpenMP -2.0	OpenMP -2.5	OpenMP -3.0	OpenMP -3.1	OpenMP-4.0
1998	2002	2005	2008	2011	2012
85p	106p	250p	326p	354p	380p

OpenMP utilisé au maximum est-il plus simple que les P-Threads ?

## Programmation OpenMP (par partage de mémoire)

1. Principes d'OpenMP
2. Détails de syntaxe et de sémantique
3. **Optimisations**
4. Exemples de parallélisation
5. Bilan d'OpenMP
6. Mesures et représentation de performances

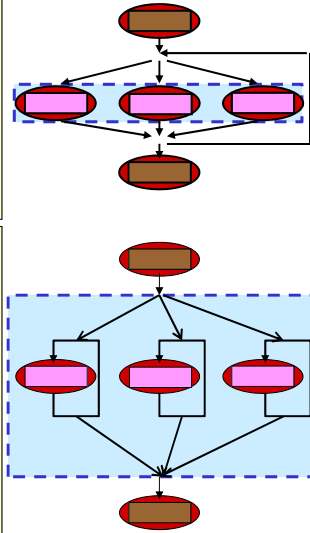
## Réutilisation des threads

Définition d'une région parallèle qui évite de re-créeer les threads

```
main() {
    .....
    for (c = 0; c < NbCycle; c++){
        #pragma omp parallel for private(i)
        for (i = 0; i < N; i++)
            .....
    }
    .....
}
```



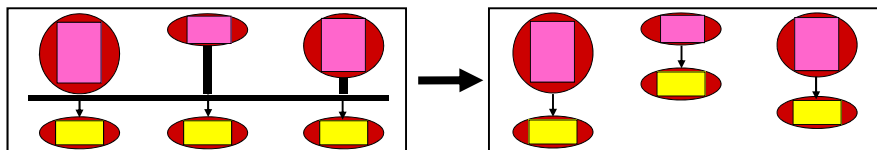
```
main() {
    .....
    #pragma omp parallel private(c)
    {
        for (c = 0; c < NbCycle; c++){
            #pragma omp for private(i)
            for (i = 0; i < N; i++)
                .....
        }
    }
    .....
}
```



## Suppression des synchro inutiles

Suppression de barrières de synchronisation systématiques :

```
#pragma omp sections nowait
#pragma omp for nowait
```



Par défaut : un `wait` est présent à la fin de chaque directive OpenMP

On peut l'enlever (`nowait`) :

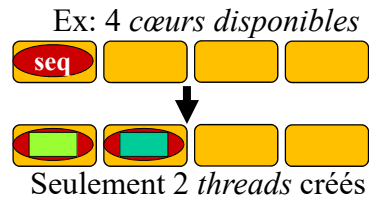
- si les threads n'ont pas besoin d'être resynchronisés à cet endroit du code !
- si il y a matière à gagner du temps : enchaîner avec des calculs de tailles différentes et espérer gagner au final

## Optimisation de l'ampleur du parallélisme

### Eviter les créations de threads inutiles :

1 - Limitation au nombre de tâches à exécuter (< nbr de rsrce)

```
..... // code séquentiel
omp_set_num_threads(2);
#pragma omp sections
{
  #pragma omp section
  { ..... }
  #pragma omp section
  { ..... }
}
```



2 – Optimisation du parallélisme selon le couple calcul/machine

```
#pragma omp parallel for num_threads(nth)
for (int i = 0; i < N; i++) {
  ..... // Accès données et calculs
}
```

Selon les calculs et les accès aux données, le nbr optimal de tâches peut varier d'une architecture à l'autre

## Distribution de boucle équilibrée

### On veille à l'équilibrage de charge

Quelle boucle paralléliser dans ce « nid de boucles » ?

```
void ActStock(double sqrttdt)
{
  int StkIdx, yIdx, xIdx;      // Loop indexes
  #pragma omp parallel private(StkIdx,yIdx,xIdx)
  {
    #pragma omp for
    for (StkIdx = 0; StkIdx < NbStocks; StkIdx++) {
      Parameters t *parPt = &par[StkIdx];
      for (yIdx = 0; yIdx < Ny; yIdx++)
        for (xIdx = 0; xIdx < Nx; xIdx++) { // Process each "trajectory"
          float call;
          // - First pass
          call = .....; // Calcul de Monte Carlo
          // - The passes that remain
          for (int stock = 1; stock <= StkIdx; stock++)
            call = .....; // Calcul de Monte Carlo
          // Copy result in the global GPU memory
          TabStockCPU[StkIdx][yIdx][xIdx] = call;
        }
      }
    }
}
```

On tente de paralléliser la boucle la plus externe (corps plus conséquent)  
 Mais la boucle interne à une taille fonction de l'itération de la boucle externe  
 → les premières itérations de la boucle externe durent moins longtemps

## Distribution de boucle équilibrée

On veille à l'équilibrage de charge

Quelle boucle paralléliser dans ce « nid de boucles » ?

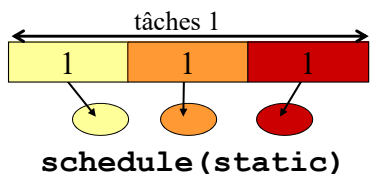
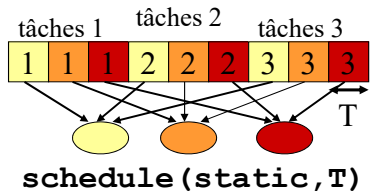
```
void ActStock(double sqrttdt)
{
    int StkIdx, yIdx, xIdx; // Loop indexes
    #pragma omp parallel private(StkIdx,yIdx,xIdx)
    {
        for (StkIdx = 0; StkIdx < NbStocks; StkIdx++) {
            Parameters t *parPt = &par[StkIdx];
            #pragma omp for
            for (yIdx = 0; yIdx < Ny; yIdx++) // Process each
                for (xIdx = 0; xIdx < Nx; xIdx++) { // "trajectory"
                    float call;
                    // - First pass
                    call = .....; // Calcul de Monte Carlo
                    // - The passes that remain
                    for (int stock = 1; stock <= StkIdx; stock++)
                        call = .....; // Calcul de Monte Carlo
                    // Copy result in the global GPU memory
                    TabStockCPU[StkIdx][yIdx][xIdx] = call;
                }
            }
        }
    }
}
```

On tente de paralléliser la boucle la plus externe (corps + conséquent)  
 Mais ici, pour un bon équilibrage de charge statique : il faut paralléliser la 2<sup>ème</sup> boucle (et non pas la première)

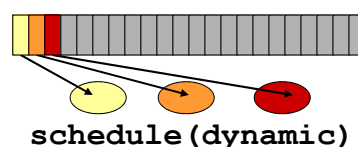
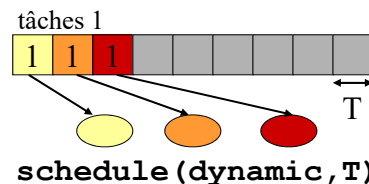
## Distribution de boucle équilibrée

Réglage du *scheduling* des `#pragma omp for` :

- taille des tâches : nombre d'itérations de boucle prises en charge
- type de répartition des tâches : `static`, `dynamic`, `guided`, `auto`, `runtime`



Calculs réguliers équilibrés



Calculs irréguliers déséquilibrés



## Distribution de boucle équilibrée

### Réglage du *scheduling* des `#pragma omp for` :

- taille des tâches : nombre d'itérations de boucle prises en charge
- type de répartition des tâches : `static`, `dynamic`, `guided`,  
`auto`, `runtime`

#### `schedule(guided, chunk_size)` :

le nombre d'itérations de chaque tâche est fixé (sauf pour la dernière), mais par le mode d'affectation des tâches (`static/dyna.`)

#### `schedule(auto)` :

le découpage des boucles en tâches est délégué au compilateur

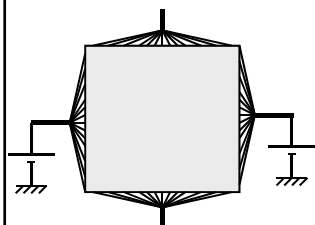
#### `schedule(runtime)` :

le découpage des boucles en tâches, et leur affectation aux threads se fera à l'exécution seulement

## Programmation OpenMP (par partage de mémoire)

1. Principes d'OpenMP
2. Détails de syntaxe et de sémantique
3. Optimisations
4. Exemples de parallélisation
5. Bilan d'OpenMP
6. Mesures et représentation de performances

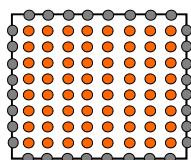
# Parallélisation de la relaxation de Jacobi



Calcul des lignes de potentiel dans une plaque diélectrique :

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = 0$$

- Discrétisation et équation aux différences
- Itération jusqu'à la convergence ( $V^{n+1}_{ij} - V^n_{ij} < \epsilon$ )



- Condition aux limites : V fixé

$$V^{n+1}_{ij} = \frac{V^n_{i-1,j} + V^n_{i+1,j} + V^n_{i,j-1} + V^n_{i,j+1}}{4}$$

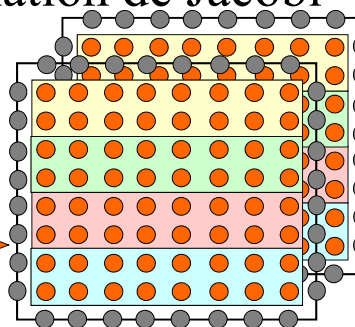
« stencil »

# Parallélisation de la relaxation de Jacobi

## Principe de parallélisation :

- Partitionnement des données et des calculs
- 2 tables ( $V^n$  et  $V^{n+1}$ ) en ShM

Ex : avec 4 *threads* de calcul



## Implantation OpenMP :

- Parallélisation de la boucle sur les lignes
- Utilisation d'une région parallèle plus globale (plus efficace)

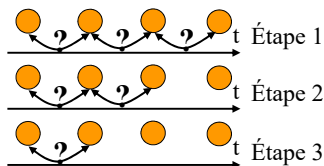
```
#pragma omp parallel private(c,...)
{
  for (c = 0; c < NbCycle; c++) {
    int current = c%2;
    int futur = (c+1)%2;
    #pragma omp for private(i,j)
    for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
        V[futur][i][j] = ... ..
  }
}
```

## Parallélisation du bubble-sort

### Principe du bubble-sort « *odd-even* » :

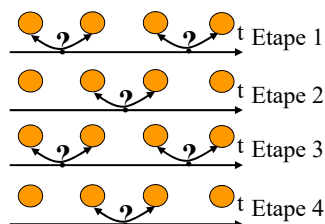
Pb : l'algorithme de base du *bubble-sort* est fortement séquentiel !

→ Modifier l'algorithme pour qu'il contienne du parallélisme potentiel !



→ Bubble-sort « *odd-even* » :

- Même complexité
- Comparaisons indépendantes

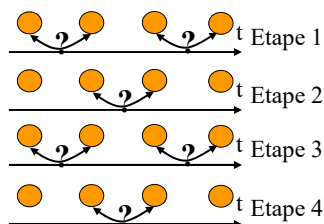


## Parallélisation du bubble-sort

### Implantation du *bubble-sort* « *odd-even* »

- Définition d'une région globale (+ efficace)

- Parallélisations distinctes des 2 boucles de tri.



```
#pragma omp parallel private(step)
{
  for (step = N; step > 0; step--) {
    if (step % 2 == 0) {
      #pragma omp for private(i,buff)
      for (i = 0; i < N-1; i += 2)
        if (Tab[i] > Tab[i+1]) {
          buff = Tab[i];
          Tab[i] = Tab[i+1];
          Tab[i+1] = buff;
        }
    } else {
      #pragma omp for private(i,buff)
      for (i = 1; i < N-1; i += 2)
        if (Tab[i] > Tab[i+1]) {
          buff = Tab[i];
          Tab[i] = Tab[i+1];
          Tab[i+1] = buff;
        }
    }
  }
}
```

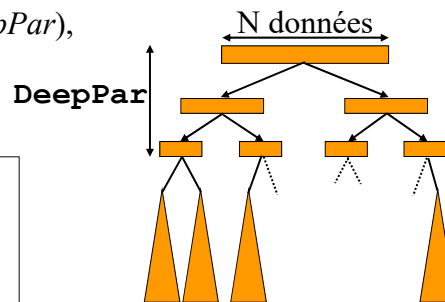
# Parallélisation du quick-sort

## Principe de parallélisation simple du Quick-Sort

Appels récursifs → Créations récursives de *sections* parallèles

→ « *nested parallelism* » nécessaire

→ limiter la profondeur de la parallélisation (variable *DeepPar*), sinon trop de *threads*!



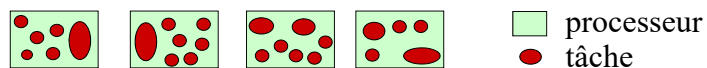
Algorithme parallèle peu efficace !  
 → Il existe des algorithmes de quick-sort parallèle beaucoup plus performants

# Parallélisation du quick-sort

## Optimisation :

- Pb d'équilibrage de charge :
  - selon les pivots .... les *sections* sont plus ou moins importantes

- solution possible : équilibre statistique !  
Nb Sections >> Nb Processeurs

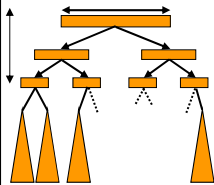


- Il faut donc créer beaucoup plus de *threads* qu'il n'y a de processeurs mais sans saturer les processeurs de *threads* !

# Parallélisation du quick-sort

## Implantation d'une parallélisation simple du Quick-Sort

Test &  
Région parallèle &  
Sections  
(OpenMP-1)



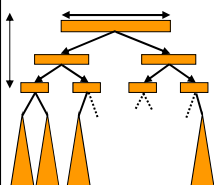
```
void quicksort(int q, int r, int deep)
{
    ...
    if (q < r) {
        ..... // split the table ...
        // classic quick-sort op

        if (deep < DEEPPAR) {
            #pragma omp parallel
            {
                #pragma omp sections nowait
                {
                    #pragma omp section
                    { quicksort(q,s-1,deep+1); }
                    #pragma omp section
                    { quicksort(s+1,r,deep+1); }
                }
            }
        } else {
            quicksort(q,s-1,deep+1);
            quicksort(s+1,r,deep+1);
        }
    }
}
```

# Parallélisation du quick-sort

## Implantation d'une parallélisation simple du Quick-Sort

Région parallèle &  
Clause  
conditionnelle &  
Sections  
(OpenMP-2)



```
void quicksort(int q, int r, int deep)
{
    ...
    if (q < r) {
        ..... // split the table ...
        // classic quick-sort op

        #pragma omp parallel if(deep < DEEPPAR)
        {
            #pragma omp sections nowait
            {
                #pragma omp section
                { quicksort(q,s-1,deep+1); }
                #pragma omp section
                { quicksort(s+1,r,deep+1); }
            }
        }
    }
}
```

# Programmation OpenMP (par partage de mémoire)

1. Principes d'OpenMP
2. Détails de syntaxe et de sémantique
3. Optimisations
4. Exemples de parallélisation
- 5. Bilan d'OpenMP**
6. Mesures et représentation de performances

CentraleSupélec

Programmation OpenMP

## Bilan

**Stratégies :**

I - Algorithmes réguliers contenant un fort parallélisme naturel :

- conserver l'algorithme classique
- compléter l'implantation séquentielle

→  $T_{dev}$  faible, Performances élevées

**OK**

II - Algorithmes irréguliers ou contenant peu de parallélisme naturel :

- conservation de l'algorithme  
ajouts au code séquentiel  
→  $T_{dev}$  faible, Perfs moyennes
- conception d'un nouvel algo  
nouvelle implantation  
→  $T_{dev}$  élevé, Perfs élevées

**OpenMP ne dispense pas de connaître l'algorithmique parallèle !**

# Programmation OpenMP (par partage de mémoire)

1. Principes d'OpenMP
2. Détails de syntaxe et de sémantique
3. Optimisations
4. Exemples de parallélisation
5. Bilan d'OpenMP
6. **Mesures et représentation de performances**

## Méthodologie de mesures

### Mesures externes :

```
>time myAppli  
>/usr/bin/time myAppli  
>times myAppli  
>timex myAppli  
.....
```

12.002u user  
0.128s system  
12.150 total

Nom et fonctionnement variables  
selon le système utilisé !!

Fréquemment :  
total > user + system !!



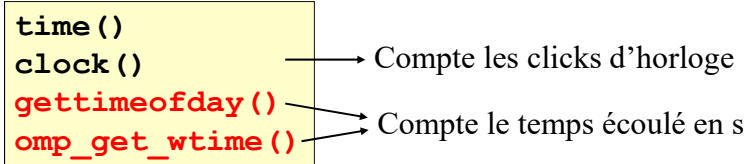
Simple à utiliser  
Pas de modifications des codes sources



Peu précis:  $\pm 0.5s$

# Méthodologie de mesures

## Mesures internes :



- Toutes ces routines ne sont pas toujours disponibles !
- “gettimeofday” est en général une bonne solution.
- Parfois il existe des outils plus précis pour mesurer de petites durées.



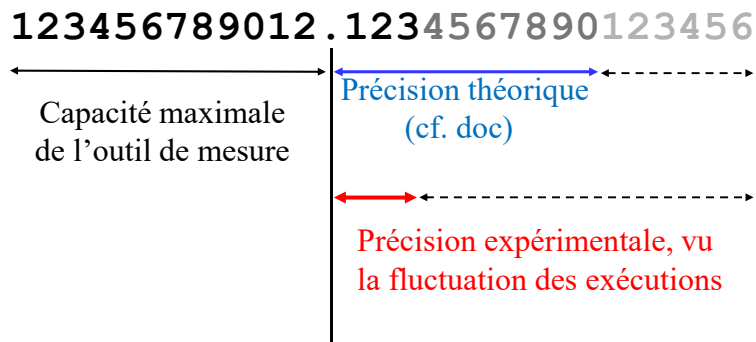
Plus précis que les mesures externes



Besoin de modifier le code source  
Pas toujours totalement portable

# Méthodologie de mesures

## Précision des outils et des mesures :



- Ne pas tenir compte de trop de décimales!
- Faire attention à ne pas déborder la capacité de mesure!



## Méthodologie de mesures

### Problème fréquent :

Test en mode exclusif (mono-user).  
Outil de mesure à 1ms de précision.



Fluctuation de 500ms  
d'une exécution à l'autre !!

Et plus encore avec la  
montée en fréquence  
automatique des procs.  
(effet de "chauffe")

### Démarche conseillée :

- Mesurer les fluctuations, ne pas les ignorer  
(le *warm up* des processeurs peut perturber les premières)
- Ne pas donner que les valeurs moyennes
- Mesurer des temps > 10s si possible

## Méthodologie de mesures

### Stocker des meta-données sur les conditions de mesure :

- **Date de l'exécution**
- **Auteur(s) du test**
- Outil(s) de mesure utilisé(s)
- Caractéristiques de la machine :  
RAM, Cache, Processeurs, ...
- OS utilisé (nom et version)
- Compilateur utilisé (nom et version)
- Options de compilation utilisées
- Test en multi-user/mono-user ?
- Présence d'IO dans le test ?
- Configuration du programme de test : taille des données, ...

On oublie souvent  
(et rapidement) à  
quel benchmark se  
réfère une série de  
mesures !

On manque souvent  
de détail sur les  
conditions de  
réalisation d'une  
série de mesures !

## Choix de la représentation

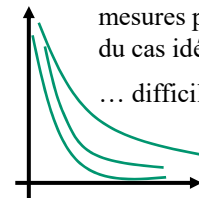
### Quelle courbe présenter ?

- Avoir une idée de l'allure de la courbe attendue et/ou de son expression
- Choisir une représentation qui permet de visualiser "des droites" ou des formes géométriques simples, détectables par l'oeil (droite, cercle, angle droit...)

### Exemple d'un temps d'exécution parallèle :

Cas idéal :  $T(P) = T(1)/P \rightarrow$  **une hyperbole**

- l'hyperbole est mal identifiée par l'œil
- on la confond facilement avec une courbe qui décroît en tendant vers une asymptote



Obtient-on des mesures proches du cas idéal ?  
... difficile à dire!

## Choix de la représentation

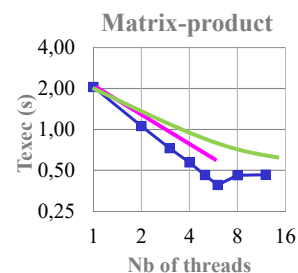
### Quelle courbe présenter ?

- Avoir une idée de l'allure de la courbe attendue et/ou de son expression
- Choisir une représentation qui permet de visualiser "des droites" ou des formes géométriques simples, détectables par l'oeil (droite, cercle, angle droit...)

### Exemple d'un temps d'exécution parallèle :

$\log(T(P)) = \log(T(1)) - \log(P)$

- en **échelle log** une hyperbole est très rapidement identifiée : **droite de pente -1**
- on détecte facilement un écart au cas idéal



## L'essentiel d'OpenMP

**Questions ?**