Programmation Parallèle Mémoire partagée: Notion de threads - Présentation d'OpenMP

Imane YOUKANA

Université Mohamed Khider - Biskra

Mémoire partagée

Deux principaux modèles de parallélisme

- Mémoire distribuée/Passage de message
- Mémoire partagée

Mémoire distribuée (Rappel)

- Chaque processus est indépendant en mémoire
- Il faut gérer la distribution des données aux différents processus
- Il faut éventuellement introduire des communications pour échanger des informations utiles entre processus
- **MPI**

Mémoire partagée

Deux principaux modèles de parallélisme

- Mémoire distribuée/Passage de message
- Mémoire partagée

Mémoire partagée

- Tous les processus partagent le même espace mémoire
- Cela semble facile à programmer car on n'a pas à se soucier de ce que chaque processus peut voir
- Cependant, il faut gérer dans le code les accès concurrents en mémoire ce qui est complexe (synchronisations)
- En pratique, on a de grandes pertes de performances si les données ne sont pas bien gérées, c'est en fait très compliqué.
- Exemple : thread C++, multithreading, **OpenMP**, Cilk, ...

API pour la programmation du parallélisme à mémoire partagée

OpenMP (Open Multi Processing)

- **Un ensemble de directives**
- Une bibliothèque de fonctions
- Des variables d'environnement

* Fortran, **C**, C++

Quelques définitions

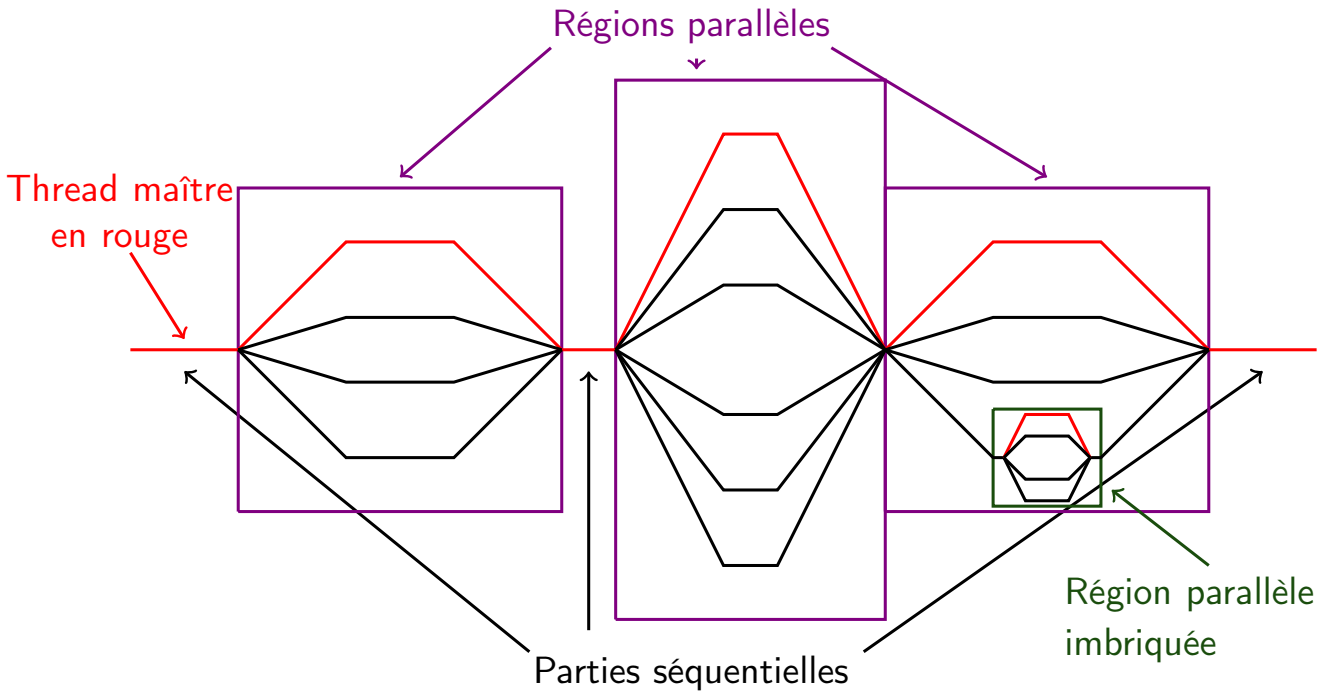
cœurs, processus, ...

- **Thread** : processus léger avec sa propre pile d'exécution (mémoire locale)
- **Team** : ensemble de threads qui participent à l'exécution d'une région parallèle
- **Thread maître** : processus créant la région parallèle (identifiant 0).

Le principe général

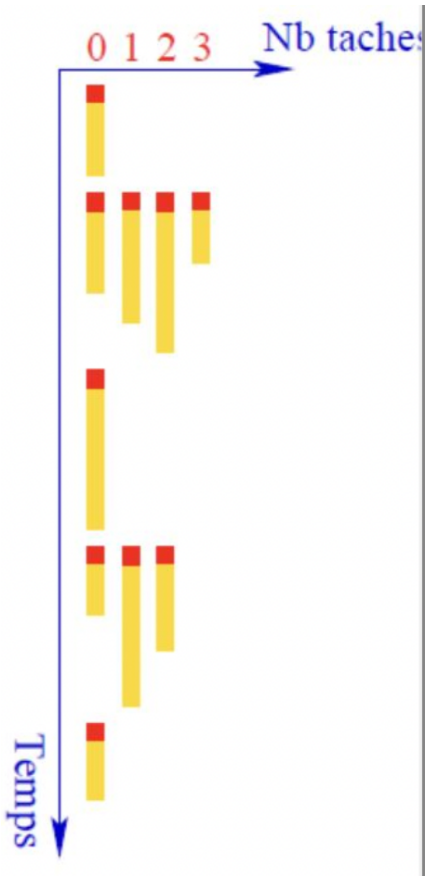
Modèle d'exécution - **Fork-Join**

- Le thread maître créé une équipe de threads
- Le parallélisme est ajouté incrémentalement jusqu'à ce que les objectifs de performance soient atteints. Le programme séquentiel évolue en programme parallèle.



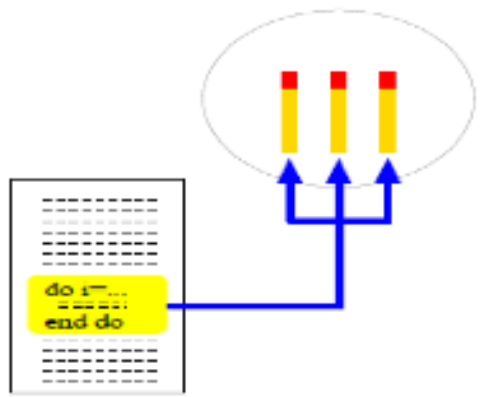
Le principe général

- Un programme OpenMP est une alternance de régions séquentielles et de régions parallèles.
- Une région séquentielle est toujours exécutée par la tâche maître (celle dont le rang vaut 0).
- Une région parallèle peut être exécutée par plusieurs tâches à la fois. Où, les tâches peuvent se partager le travail contenu dans la région parallèle.

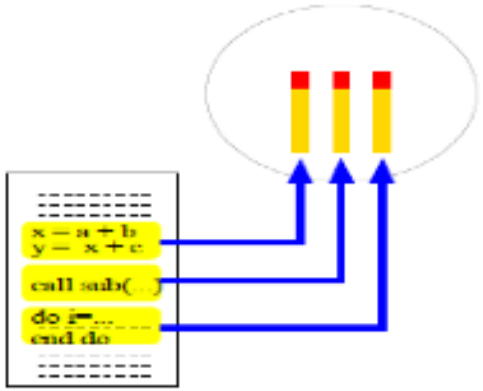


Le principe général

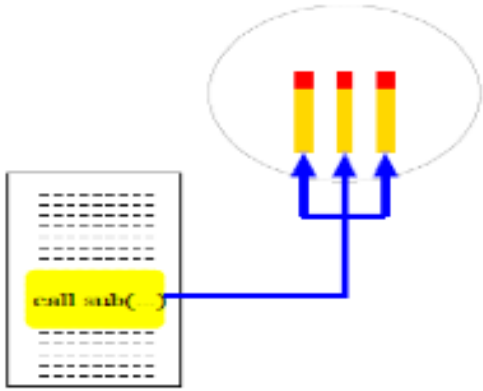
- Le partage du travail consiste essentiellement à :
  - exécuter une boucle par répartition des itérations entre les tâches;
  - exécuter plusieurs sections de code mais une seule par tâche,
  - exécuter plusieurs occurrences d'une même procédure par différentes tâches.



Boucle parallèle
(Looplevel parallelism)



Sections parallèles



Procédure parallèle (orphaning)

La syntaxe

- La plupart des constructions OpenMP sont des directives de compilateur.

```
#pragma omp construct [clause [clause]...]
```

Par exemple

```
#pragma omp parallel num_threads(4)
```

- Il y a aussi quelques fonctions et types spécifiques :

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

- La plupart des constructions OpenMP s'appliquent à un bloc structuré.
  - Un bloc structuré est un bloc d'une ou plusieurs instructions avec un point d'entrée au début du bloc et une sortie à la fin du bloc.
  - Il est possible d'avoir un `exit()` dans un bloc structuré.

Création de threads

Les régions parallèles

- En OpenMP, les threads sont créés avec la construction `parallel`
- Par exemple, pour créer une région parallèle de 4 threads :

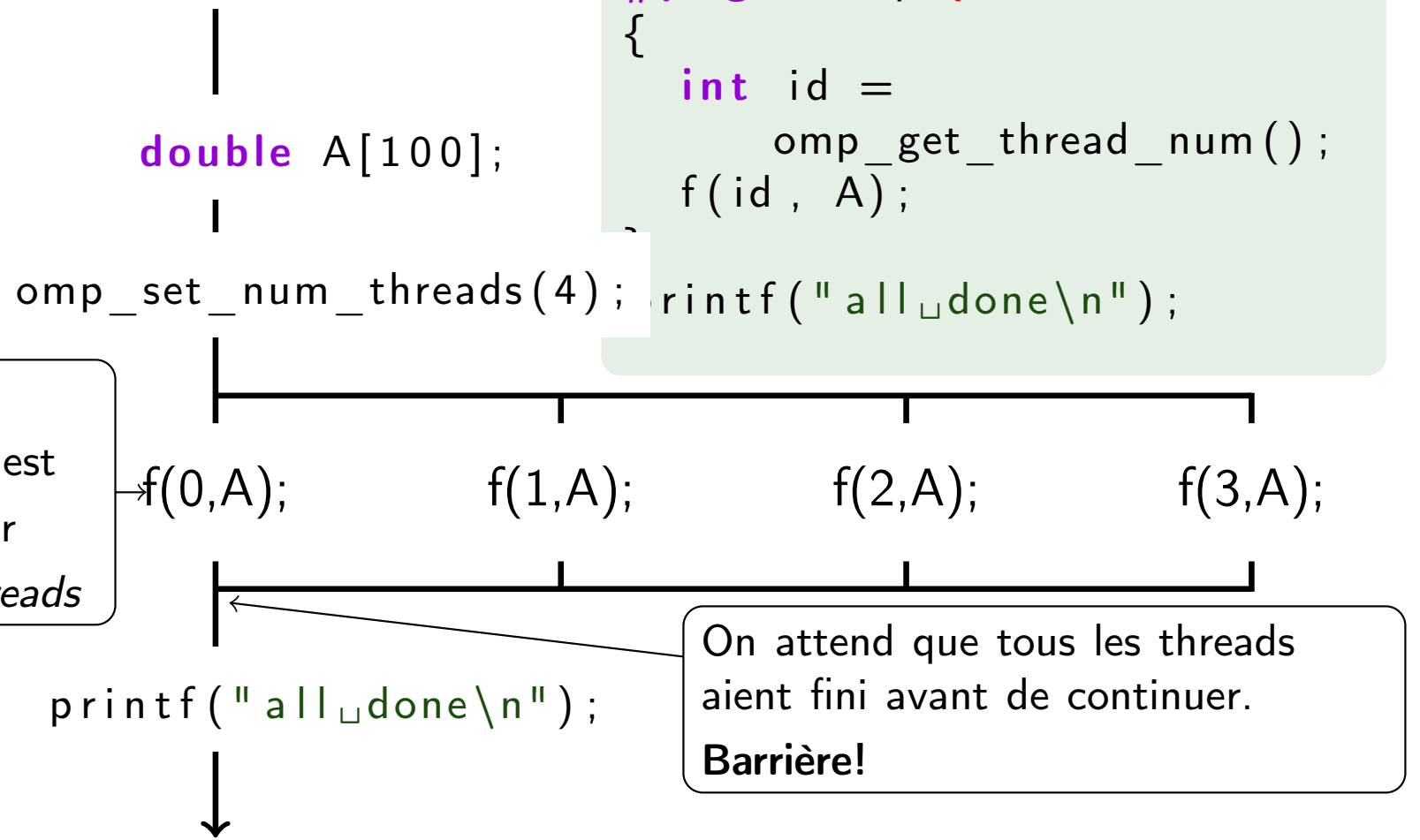
```
double A[100];

#pragma omp parallel num_threads(4)
{
  int id = omp_get_thread_num();
  f(id, A);
}
```

Interprétation

- Chaque thread appelle `f(id, A)` pour `id = 0` à 3.
- Le tableau `A` est une variable globale et partagée par tous les threads

Création de threads : régions parallèles

```
double A[100];
omp_set_num_threads(4);
#pragma omp parallel
{
   int id =
       omp_get_thread_num();
   f(id, A);
}
printf("all done\n");
```



Une seule copie de A est partagée par tous les *threads*

On attend que tous les threads aient fini avant de continuer.

Barrière!

Quelques fonctions OpenMP

Nombre de threads, identifiant

```
int omp_get_num_threads();

int omp_get_thread_num();
```

1. nombre de threads dans l'équipe
2. id (ou rang) du thread

Un premier exemple (1)

Hello world

```
void main (){



    int id = 0;
    printf ("Hello(%d) ", id);
    printf ("world(%d)\n", id);

}
```

Un premier exemple (2)

Hello world

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

#pragma omp parallel
  {
    int id = 0;
    printf ("Hello(%d) ", id);
    printf ("world(%d)\n", id);
  }
}
```

Compilation

- avec gcc

  `gcc -fopenmp hello.c -o hello`

Exécution

- `OMP_NUM_THREADS=4 ./hello`

Un premier exemple (3)

Hello world

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

#pragma omp parallel
  {
    int id = omp_get_thread_num();
    printf ("Hello(%d) ", id);
    printf ("world(%d)\n", id);
  }
}
```

Résultat

hello(0) hello(1) world(1)
world(0)
hello(3) world(3)
hello(2) world(2)

SPMD vs partage de travail/collaboration

- La construction `parallel` crée un programme SPMD (*Single Program Multiple Data*), c-à-d où chaque thread exécute le même code.
- Comment diviser un chemin d'exécution entre plusieurs threads formant une équipe?
  C'est de la collaboration (*Worksharing*)
  - Boucles
  - Sections
  - Tasks

La directive `for`

Collaboration dans une boucle

La construction d'une collaboration sur une boucle répartit les itérations de la boucle en plusieurs threads dans une équipe.

Exemple

```
#pragma omp parallel
{

  #pragma omp for
  for (i = 0; i < N; i++)
    calcul_savant(i);
}
```

Attention

La variable `i` est privée à chaque thread. Ceci peut être fait explicitement avec la clause `private(i)`.

Comment partager le travail d'une boucle ?

Séquentiel

```
for (i=0, i < N; i++) a[i] = a[i] + b[i];
```

OpenMP : région parallel

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

Comment partager le travail d'une boucle ?

Séquentiel

```
for (i=0, i < N; i++) a[i] = a[i] + b[i];
```

OpenMP : région parallel + construction for de collaboration

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

OpenMP : construction combinée

```
#pragma omp parallel for
for (i=0; i < N; i++) a[i] = a[i] + b[i];
```

Paralléliser les boucles

Approche de base

- Trouver les boucles de calcul intensif
- Rendre les itérations de la boucle indépendantes afin qu'elles puissent s'exécuter dans n'importe quel ordre.
- Placer la directive OpenMP appropriée et c'est terminé !

Réduction

Comment gérer ce cas?

```
double ave=0.0, A[max];
int i;
for (i = 0; i < max; i++)
  ave += A[i];
ave /= max;
```

Situation très fréquente

- Toutes les valeurs sont accumulées dans une seule variable. Il y a une vraie dépendance entre les itérations de la boucle qui ne peut pas être supprimée trivialement.
- C'est une situation connue qu'on appelle une réduction.
- Comme déjà vu avec MPI, des opérations de réduction sont souvent incluses dans les environnements de programmation parallèle.

Réduction avec OpenMP

La clause `reduction`

```
reduction (op :  list)
```

Dans une construction de collaboration ou `parallel`

- Une copie locale de chaque variable de `list` est créée et initialisée en fonction de l'opérateur `op`(0 pour + par ex.).
- Le compilateur trouve les expressions de réduction standard qui contiennent `op` et les utilise pour mettre à jour la copie.
- Les copies locales sont réduites à une valeur unique et combinées dans la valeur globale d'origine.

Réduction avec OpenMP

La clause `reduction`

```
reduction (op :  list)
```

Dans une construction de collaboration ou `parallel`

- Les variables de `list` doivent être partagées dans la région parallèle englobante

Exemple

```
double ave=0.0, A[max];
#pragma omp parallel for reduction(+:ave)
for (int i = 0; i < max; i++)
  ave += A[i];
ave /= max;
```

Les directives de synchronisation

Les constructions de haut niveau

- Critical
- Atomic
- Barrière
- Ordonnée

Les sections critiques

Définition

Un seul thread à la fois peut entrer dans une région `critical`

Exemple

```
float res;
#pragma omp parallel
{
  float B; int i, id, nthrds;
  id = omp_get_thread_num();
  nthrds = omp_get_num_threads();

  for(i=id;i<niters;i+nthrds){
    B = big_job(i);

#pragma omp critical
    consume (B, res);
  }
}
```

Interprétation

Les threads attendent leur tour. Un seul appel à `consume()` à la fois.

Barrière simple

La directive `barrier`

Chaque thread attend tous les autres threads.

Détails

```
#pragma omp parallel shared(A,B,C) private(id)
{
  id=omp_get_thread_num();
  A[id] = big_calc1(id);

  #pragma omp barrier

  #pragma omp for
  for (i = 0; i < N; i++){C[i] = big_calc3(i,A);}

  #pragma omp for nowait
  for (i = 0; i < N; i++){B[i] = big_calc2(C,i);}
  A[id] =  big_calc4(id);
}
```

Barrière implicite

La barrière implicite de la construction `for`

- par défaut il y a une barrière à la fin d'une construction `for`
- La clause `nowait` supprime cette barrière

```
#pragma omp parallel shared(A,B,C) private(id)
{
  id=omp_get_thread_num();
  A[id] = big_calc1(id);

  #pragma omp barrier

  #pragma omp for
  for (i = 0; i < N; i++){C[i] = big_calc3(i,A);}

  #pragma omp for nowait
  for (i = 0; i < N; i++){B[i] = big_calc2(C,i);}
  A[id] =  big_calc4(id);
}
```

Barrière finale

Synchronisation de la région parallèle

- Barrière implicite à la fin de la région `parallel`

```
#pragma omp parallel shared(A,B,C) private(id)
{
  id=omp_get_thread_num();
  A[id] = big_calc1(id);

  #pragma omp barrier

  #pragma omp for
  for (i = 0; i < N; i++){C[i] = big_calc3(i,A);}

  #pragma omp for nowait
  for (i = 0; i < N; i++){B[i] = big_calc2(C,i);}
  A[id] =  big_calc4(id);
}
```

Synchronisation : Construction `master`

`#pragma omp master`

- La construction `master` annonce un bloc structuré qui est uniquement exécuté par le thread maître
- Les autres threads l'ignorent (aucune synchronisation)

```
#pragma omp parallel
{
  do_many_things();
  #pragma omp master
  {
    exchange_boundaries();
  }
  #pragma omp barrier
  do_many_other_things
}
```

Construction single

#pragma omp single

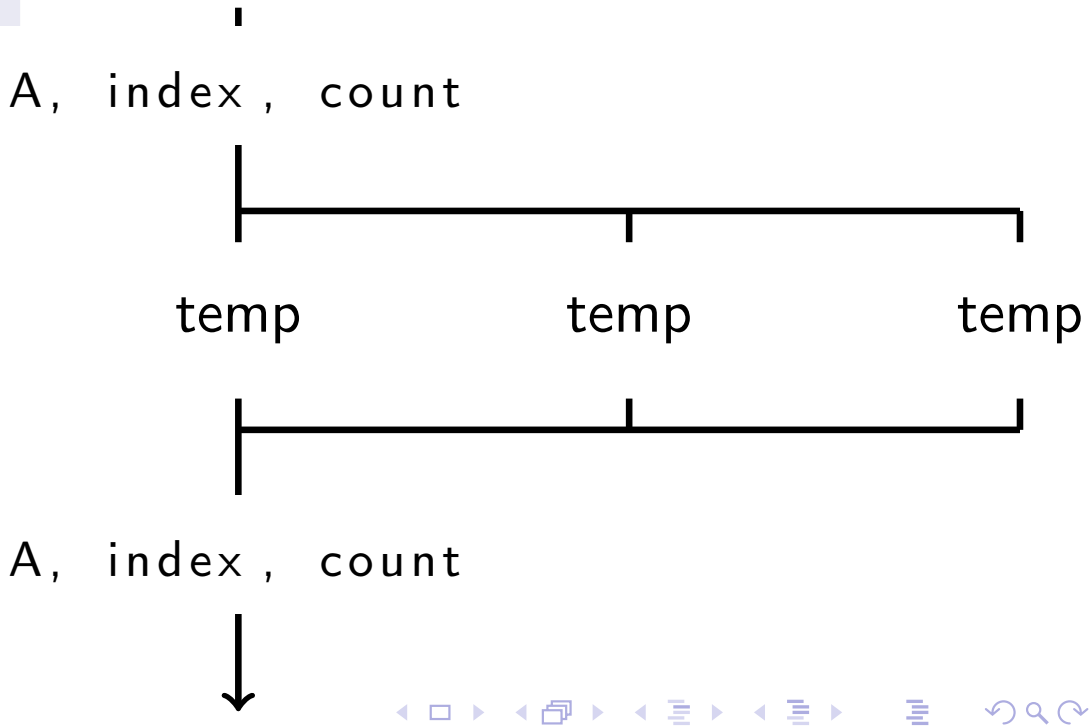
- La construction single annonce un bloc structuré qui sera exécuté par un seul thread (pas nécessairement le thread maître)
- Il y a une barrière implicite à la fin d'un bloc single
- On peut enlever la barrière avec la clause nowait.

```
#pragma omp parallel
{
  do_many_things();
  #pragma omp single
  {
    exchange_boundaries();
  }
  do_many_other_things
}
```

Partage de données

```c
double A[10];
int main(){
  int index[10];
  #pragma omp parallel
    work(index);
  printf("%d\n", index
      [0]);
}
```

```c
extern double A[10];
void work(int* index){
  double temp[10];
  static int count;
  ...
  ...
}
```



`A, index` et `count` sont partagés par tous les threads.

`temp` est local à chaque thread

Partage de données

Changer les attributs

- Il est possible de changer les attributs de stockage des données pour une construction omp en utilisant les clauses suivantes :
  - **shared** :
    partagée par tous les threads.
  - **private** :
    locale à chaque thread (non partagée $\rightarrow$ privée).
  - **firstprivate** :
    comme `private` mais initialisée, pour chaque thread avec sa valeur avant la construction OpenMP.
  - **lastprivate** :
    comme `private` mais la dernière valeur de la donnée dans une boucle parallèle est transmise à la variable partagée en dehors de la boucle.

Attention!

Ces clauses s'appliquent à la construction OpenMP, PAS à la région entière.

Clause private

Exemple

```
void wrong(){
  int tmp = 0;
  #pragma omp for private(tmp)
  for (int j = 0; j < 1000; j++)
    tmp += j;

  printf ("tmp = %d\n", tmp);
}
```

private(var)

- Créé une nouvelle copie locale de var pour chaque thread.

Attention

tmp n'est pas initialisée. Valeur de sortie ??

Clause `firstprivate`

Exemple

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

`firstprivate`

- Cas particulier de `private`
- Initialise chaque copie privée avec la valeur correspondante du thread maître.

Interprétation

- Chaque thread obtient son propre `tmp` (ici $tmp = 0$)
- A la sortie de la construction *tmp* n'a pas changé de valeur !

Clause lastprivate

Exemple

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

lastprivate

- Cas particulier de private
- Passe la valeur d'une variable privée à la dernière itération à la variable globale

Interprétation

tmp=0 initialement et à la sortie tmp=??? tel que défini à la dernière itération pour une des variables privées.

La clause schedule

La clause schedule modifie comment les itérations de la boucle seront projetées sur les threads.

- schedule(static[, chunk])
  Distribue des blocs d'itérations de taille `chunk` à chaque thread
- schedule(dynamic[, chunk])
  Chacun son tour, chaque thread prend `chunk` itérations dans une liste jusqu'à ce que toutes les itérations aient été réalisées.
- schedule(guided[, chunk])
  Les threads récupèrent dynamiquement des blocs d'itérations. La taille des blocs est d'abord grosse puis diminue jusqu'à la taille `chunk` pendant que le calcul progresse.

Quand se servir de quel ordonnancement?

Ordonnancement	Quand s'en servir?
STATIC	Calcul pré-déterminé et prédictible par le programmeur
DYNAMIC	Non prédictible, quantité de travail par itération très variable
GUIDED	Cas spécifique d'ordonnancement dynamique pour réduire la surcharge (*overhead*) introduite