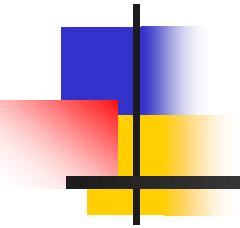
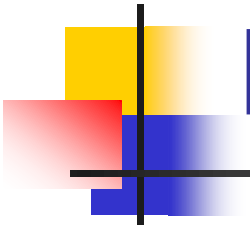


OpenMP

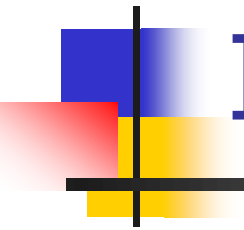
Introduction à OpenMP

Julien Jaeger
julien.jaeger@cea.fr

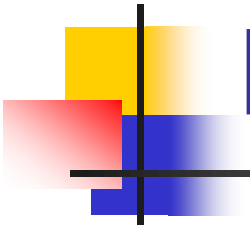


Plan du cours

- Introduction à OpenMP
- Région parallèle
- Partage ou non des données
- Synchronisations
- Partage de travail
- Exécution exclusive



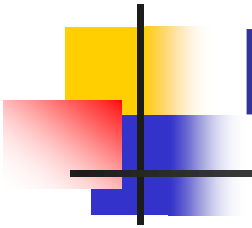
Introduction à OpenMP



Définition

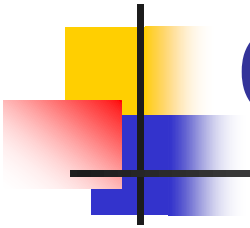
- Définition : OpenMP (Open Multi-Processing)
  - Interface de programmation pour le calcul parallèle sur architecture à mémoire partagée.
  - Supportée sur de nombreuses plateformes
    - Unix, Windows
  - Multi-langages de programmation
    - C/C++ et Fortran.
  - Ensemble de directives + bibliothèque logicielle + variables d'environnement.
- OpenMP est portable et dimensionnable.
  - Développement rapide d'applications parallèles
  - Granularité restant proche du code séquentiel

- Le 28 octobre 1997, une majorité importante d'industriels et de constructeurs ont adopté OpenMP (Open Multi Processing) comme un standard dit « industriel ».



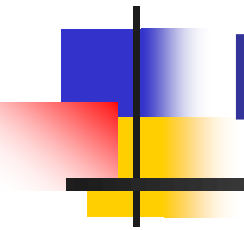
Historique

- Les spécifications d'OpenMP appartiennent aujourd'hui à l'ARB (Architecture Review Board), seul organisme chargé de son évolution.
  - http://www.openmp.org
  - http://www.compunity.org
- Version OpenMP 2.0 a été finalisée en novembre 2000
  - Extensions relatives à la parallélisation de certaines constructions Fortran 95.
  - Version 2.5 Mai 2005
- Version 3.0 en Mai 2008
  - Parallélisme de tâches
  - Version 3.1 pour 2011
    - Draft disponible
    - Finalisation lors de la conférence IWOMP 2011 (http://www.iwomp.org)
- Version 4.5 en Novembre 2015
  - Support des accélérateurs



Concepts généraux

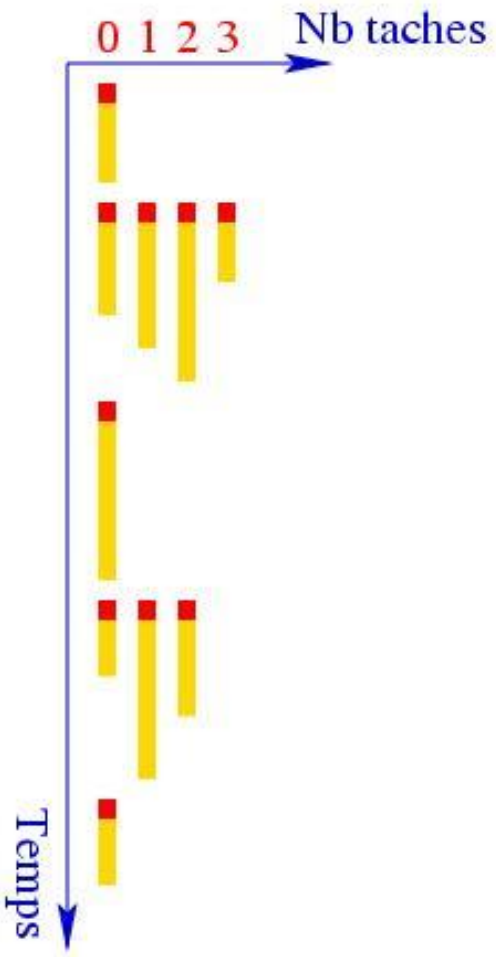
- Un programme OpenMP est exécuté par un processus unique (processus maître).
  - Ce processus active des processus légers (threads) à l'entrée d'une **région parallèle**
- Chaque processus léger exécute une tâche implicite composée d'un ensemble d'instructions.
  - Notion de rang
- Pendant l'exécution d'une tâche, une variable peut être lue et/ou modifiée en mémoire.
  - Elle peut être définie dans la pile (stack) (espace mémoire local) d'un processus léger ; on parle alors de variable privée.
  - Elle peut être définie dans un espace mémoire partagé



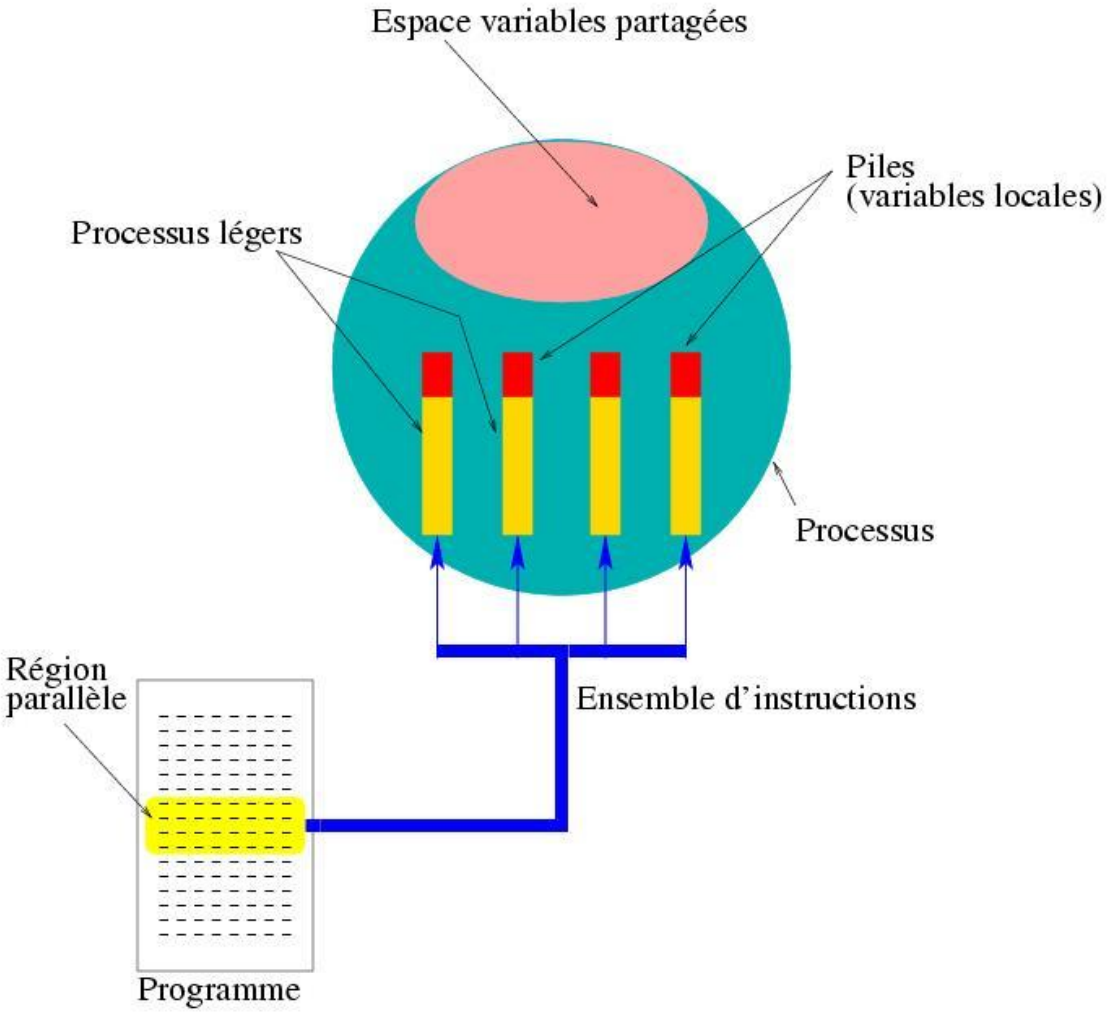
Région parallèle

Région parallèle

- 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.
- Les tâches peuvent se partager le travail contenu dans la région parallèle.

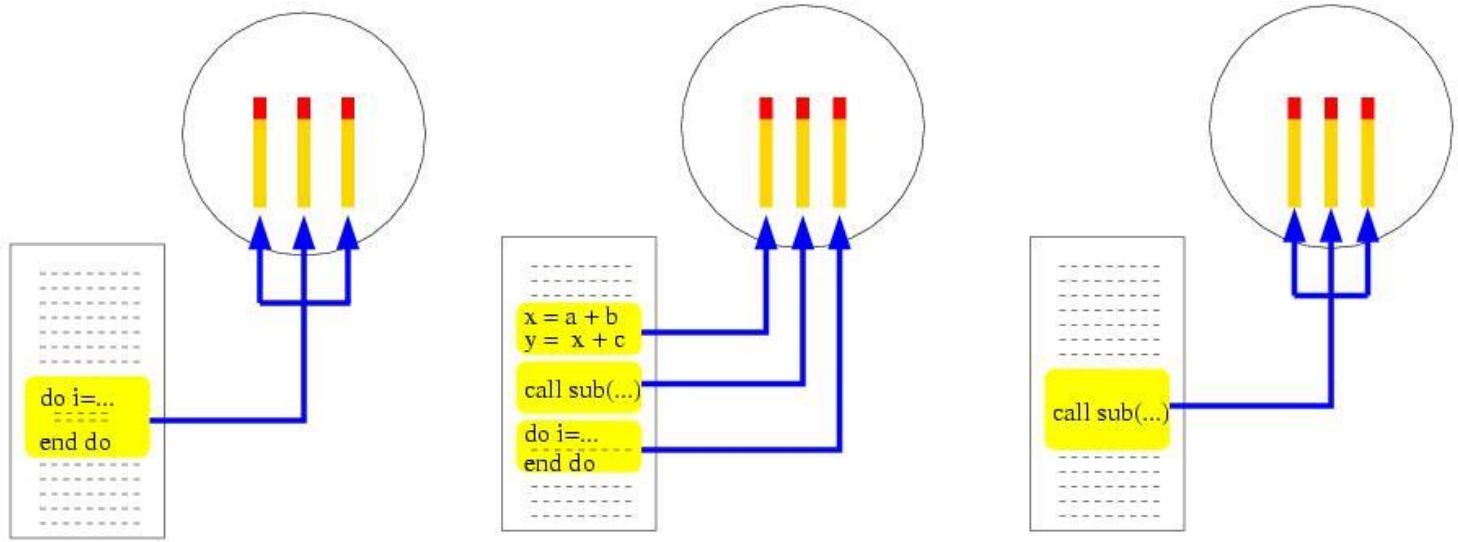


Région parallèle



Partage du travail

- 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 des tâches explicites différentes



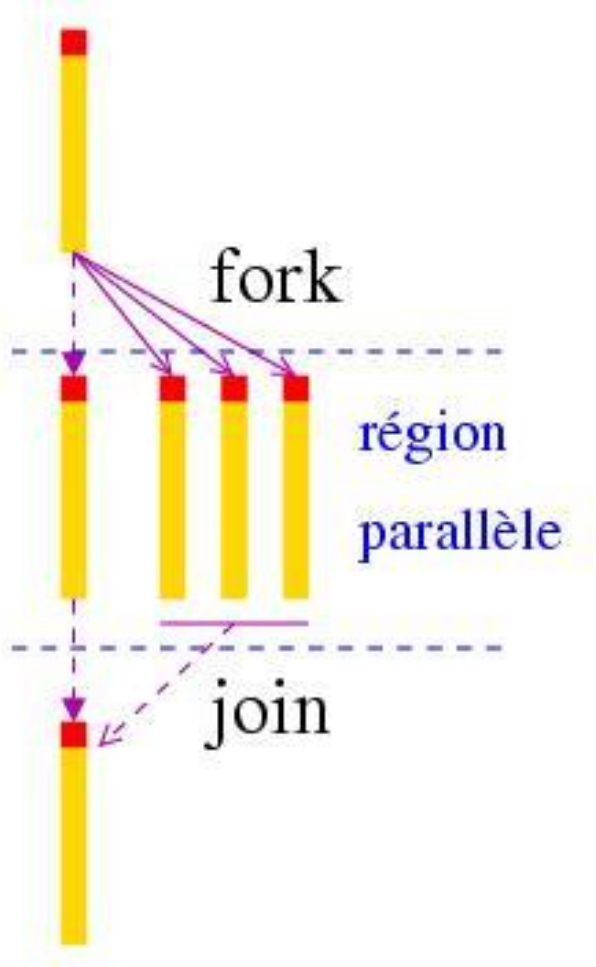
Boucle parallèle (Looplevel parallelism)

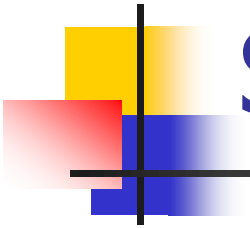
Sections parallèles

Procédure parallèle (orphaning)

Principe d'OpenMP

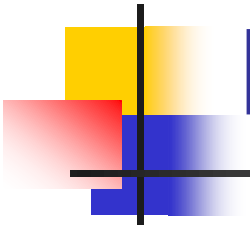
- Ajout de directives OpenMP
  - Travail du programmeur
  - Possibilité d'ajout automatique dans certains cas précis
- À l'exécution du programme, le système d'exploitation construit une région parallèle sur le modèle « fork/join ».
- À l'entrée d'une région parallèle, la tâche maître crée/active (fork) des processus « fils » (processus légers) qui disparaissent (ou s'assoupissent) en fin de région parallèle (join) pendant que la tâche maître poursuit seule l'exécution du programme jusqu'à l'entrée de la région parallèle suivante.





Syntaxe générale

- Une directive OpenMP possède la forme générale suivante :
  - **sentinelle directive [clause[clause]…]**
- C'est une ligne qui doit être ignorée par le compilateur si l'option permettant l'interprétation des directives OpenMP n'est pas spécifiée.
- La sentinelle est une chaîne de caractères dont la valeur dépend du langage utilisé.
  - C/C++ : #pragma
  - Fortran : !$
- Il existe un module Fortran 95 OMP_LIB et un fichier d'inclusion C/C++ omp.h qui définissent le prototype de toutes les fonctions OpenMP.
  - Il est indispensable de les inclure dans toute unité de programme OpenMP utilisant ces fonctions.



Directives & Environnement

- Directives et clauses de compilation :
  - Définition de région parallèle, de partage de travail, de synchronisation, du flot de données, …
  - Elles sont ignorées par le compilateur à moins de spécifier une option adéquate de compilation pour qu'elles soient interprétées.
- Fonctions et sous-programmes : ils font partie d'une bibliothèque chargée à l'édition de liens du programme.
- Variables d'environnement : une fois positionnées, leurs valeurs sont prises en compte à l'exécution.

Premier programme

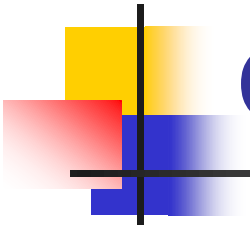
```
#include <omp.h>
#include <stdio.h>

void main() {
  #pragma omp parallel
  {
    printf( "Hello from thread %d\n",
      omp_get_thread_num() ) ;
  }
}
```

Header

Directive

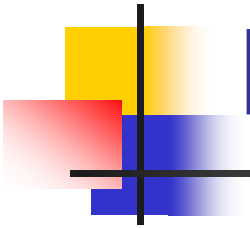
Région parallèle

```
$ gcc –o test –fopenmp test.c

$ export OMP_NUM_THREADS=4
$ ./test
Hello from thread 1
Hello from thread 2
Hello from thread 3
Hello from thread 0
```



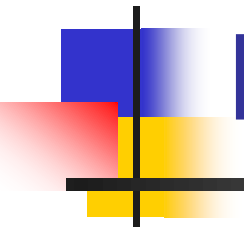
Construction d’une région parallèle

- Au sein d'une même région parallèle, toutes les tâches concurrentes exécutent le même code.
- Il existe une barrière implicite de synchronisation en fin de région parallèle.
- Il est interdit d'effectuer des branchements (ex. GOTO, CYCLE, etc.) vers l'intérieur ou vers l'extérieur d'une région parallèle ou de toute autre construction OpenMP.



Étendue d’une région parallèle

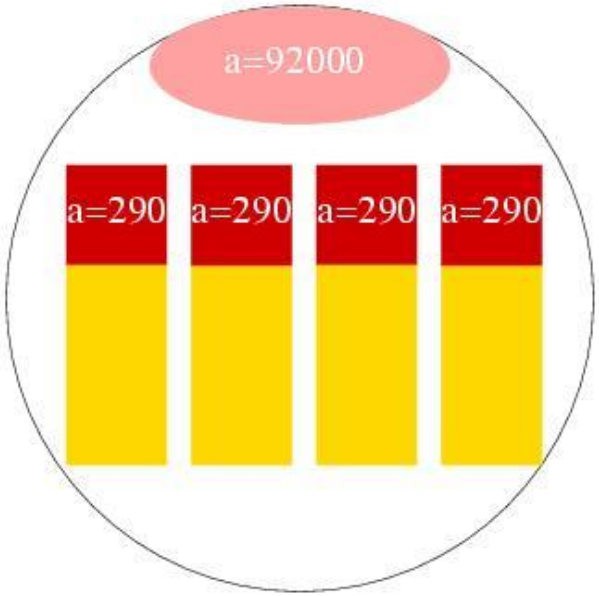
- L'étendue d'une construction OpenMP représente le champ d'influence de celle-ci dans le programme.
- L'influence (ou la portée) d'une région parallèle s'étend aussi bien au code contenu lexicalement dans cette région (étendue statique), qu'au code des sous-programmes appelés.
  - L'union des deux représente « l'étendue dynamique ».



Flôt de données

Flot de données

- Dans une région parallèle, par défaut, le statut des variables est partagé.
- Il est possible, grâce à la clause DEFAULT, de changer le statut par défaut des variables dans une région parallèle.
- Si une variable possède un statut privé (PRIVATE), elle se trouve dans la pile de chaque tâche. Sa valeur est alors indéfinie à l'entrée d'une région parallèle (dans l'exemple ci-contre, la variable a vaut 0 à l'entrée de la région parallèle).



Données privées

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

int main() {
  float a;
  a = 92000.;
  printf( "Out region: %p\n", &a);
#pragma omp parallel
  {
    printf("In region: %p thread %d\n",
      &a, omp_get_thread_num() );
  }
  return 0;
}
```

```
$ gcc –fopenmp –o test test.c

$ OMP_NUM_THREADS=4
$ ./test
Out region: 0xbf8d9a9c
In region: 0xbf8d9a9c thread 1
In region: 0xbf8d9a9c thread 2
In region: 0xbf8d9a9c thread 3
In region: 0xbf8d9a9c thread 0
```

Données privées

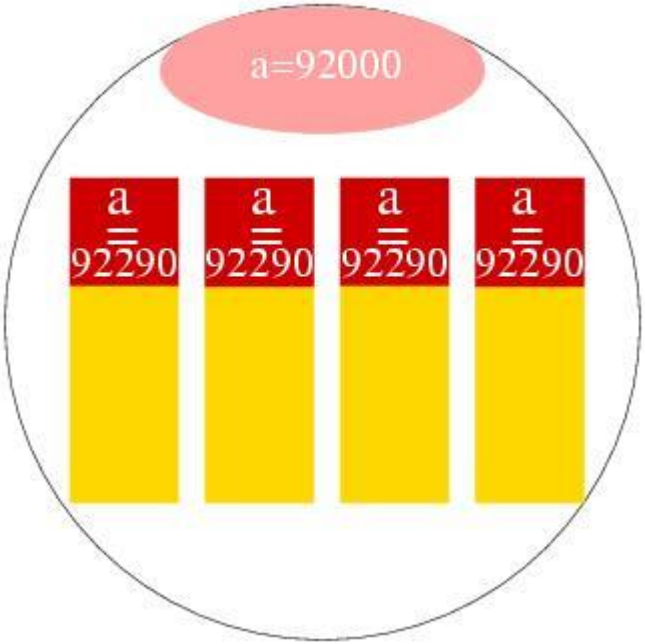
```
#include <stdio.h>
#include <omp.h>

int main() {
  float a;
  a = 92000.;
  printf( "Out region: %p\n", &a);
#pragma omp parallel private(a)
  {
    printf("In region: %p thread %d\n",
      &a, omp_get_thread_num() );
  }
  return 0;
}
```

```
$ gcc –fopenmp –o test test.c

$ OMP_NUM_THREADS=4
$ ./test
Out region: 0xbf887e2c
In region: 0xbf887dfc thread 0
In region: 0xb67cf2ec thread 3
In region: 0xb6fd02ec thread 2
In region: 0xb77d12ec thread 1
```

Clause

Données privées initialisées

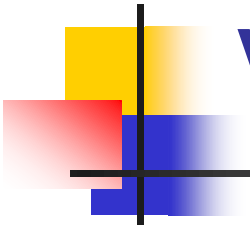
- Cependant, grâce à la clause FIRSTPRIVATE, il est possible de forcer l'initialisation de cette variable privée à la dernière valeur qu'elle avait avant l'entrée dans la région parallèle.



Données privées initialisées

```
#include <stdio.h>

int main() {
  float a;
  a = 92000.;
#pragma omp parallel default(none) \
    firstprivate(a)
  {
    a = a + 290.;
    printf("a vaut : %f\n",a);
  }
  printf("Hors region, a vaut : %f\n",
    a);
  return 0;
}
```

```
$ gcc –o prog -fopenmp prog.c

$ export OMP_NUM_THREADS=4
$ ./prog
a vaut : 92290.
a vaut : 92290.
a vaut : 92290.
a vaut : 92290.
Hors region, a vaut : 92000.
```



Variables statiques

- Une variable est statique si son emplacement en mémoire est défini à la déclaration par le compilateur.
- En Fortran, c'est le cas des variables apparaissant en COMMON ou contenues dans un MODULE ou déclarées SAVE ou initialisées à la déclaration (ex. PARAMETER, DATA, etc.).
- En C, c'est le cas des variables externes ou déclarées STATIC.

Variables statiques

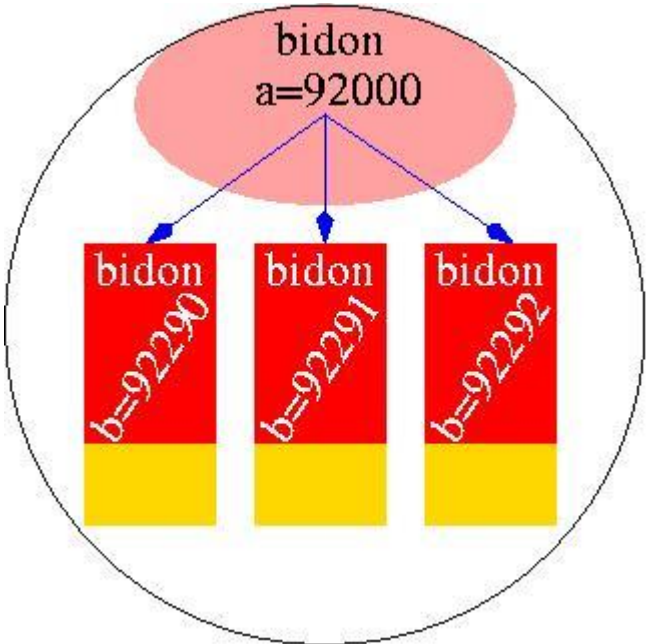
```
#include <omp.h>
float a;

int main() {
  a = 92000;
#pragma omp parallel
  {
    sub();
  }
  return 0;
}
```

```
#include <stdio.h>
float a;

void sub(void) {
  float b;

  b = a + 290.;
  printf(
    "b vaut : %f\n",b);
}
```

```
$ export OMP_NUM_THREADS=2
$ ./prog
B vaut : 92290
B vaut : 92290
```

Variables statiques

- L'utilisation de la directive THREADPRIVATE permet de privatiser une instance statique et faire que celle-ci soit persistante d'une région parallèle à une autre.
- Si, en outre, la clause COPYIN est spécifiée alors la valeur des instances statiques est transmise à toutes les tâches.

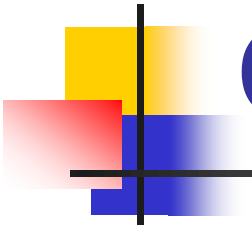


Variables statiques

```
#include <stdio.h>
#include <omp.h>
int a;
#pragma omp threadprivate(a)

int main() {
  a = 92000;
  #pragma omp parallel copyin(a)
  {
    a = a + omp_get_thread_num();
    sub();
  }
  printf(
   "Hors region, A vaut: %d\n",a);
  return 0;
}
```

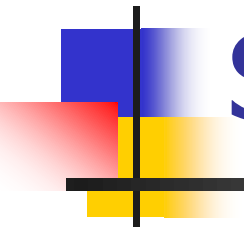
```
#include <stdio.h>

int a;
#pragma omp threadprivate(a)

void sub(void) {
  int b;
  b = a + 290;
  printf("b vaut : %d\n",b);
}
```

```
$ OMP_NUM_THREADS=4 ./prog
B vaut : 92290
B vaut : 92291
B vaut : 92292
B vaut : 92293
Hors region, A vaut : 92000
```

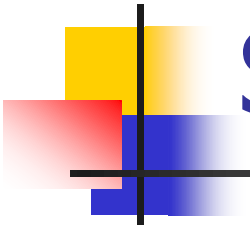


Compléments

- La construction d'une région parallèle admet deux autres clauses :
  - REDUCTION : pour les opérations de réduction avec synchronisation implicite entre les tâches ;
  - NUM_THREADS : elle permet de spécifier le nombre de tâches souhaité à l'entrée d'une région parallèle de la même manière que le ferait le sous-programme OMP_SET_NUM_THREADS.
    - → Exemple précédent avec l'utilisation de cette clause/fonction
- D'une région parallèle à l'autre, le nombre de tâches concurrentes peut être variable si on le souhaite. Pour cela, il suffit d'utiliser le sous-programme OMP_SET_DYNAMIC ou de positionner la variable d'environnememt OMP_DYNAMIC à true.
- Il est possible d'imbriquer (nesting) des régions parallèles, mais cela n'a d'effet que si ce mode a été activé à l'appel du sous-programme OMP_SET_NESTED ou en positionnant la variable d'environnement OMP_NESTED.

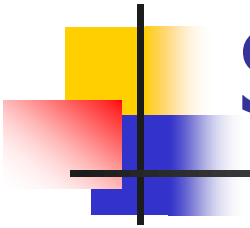


Synchronisations



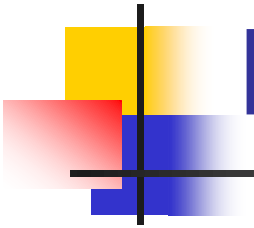
Synchronisations

- La synchronisation devient nécessaire dans les situations suivantes :
  1. pour s'assurer que toutes les tâches concurrentes aient atteint un même niveau d'instruction dans le programme (barrière globale)
  2. pour ordonner l'exécution de toutes les tâches concurrentes quand celles-ci doivent exécuter une même portion de code affectant une ou plusieurs variables partagées dont la cohérence (en lecture ou en écriture) en mémoire doit être garantie (exclusion mutuelle).
  3. pour synchroniser au moins deux tâches concurrentes parmi l'ensemble (mécanisme de verrous).



Synchronisations

- Il est possible d'imposer explicitement une barrière globale de synchronisation grâce à la directive BARRIER.
- Le mécanisme d'exclusion mutuelle (une tâche à la fois) se trouve, par exemple, dans les opérations de réduction (clause REDUCTION) ou dans l'exécution ordonnée d'une boucle (directive DO ORDERED). Dans le même but, ce mécanisme est aussi mis en place dans les directives ATOMIC et CRITICAL.
- Des synchronisations plus fines peuvent être réalisées soit par la mise en place des mécanismes de verrous (cela nécessite l'appel à des sous-programmes de la bibliothèque OpenMP), soit par l'utilisation de la directive FLUSH.



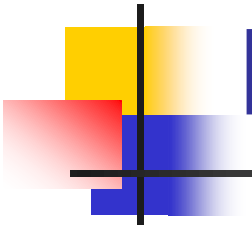
Barrière

- Principe implicite
  - Chaque construction OpenMP (région parallèle, partage de travail, …) implique une barrière implicite à la fin
  - Cette barrière concerne tous les threads de la même équipe (*team*)
- La directive BARRIER synchronise l'ensemble des tâches concurrentes dans une région parallèle.

```
#pragma omp barrier
```

- Chacune des tâches attend que toutes les autres soient arrivées à ce point de synchronisation pour poursuivre, ensemble, l'exécution du programme.

Mise à jour atomique



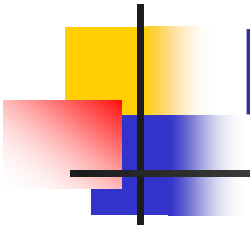
- La directive ATOMIC assure qu'une variable partagée est lue et modifiée en mémoire par une seule tâche à la fois.

```
#pragma omp atomic
```

- Son effet est local à l'instruction qui suit immédiatement la directive.

Mise à jour atomique

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

int main() {
    int compteur, rang;

    compteur = 92290;
    #pragma omp parallel private(rang)
    {
        rang=omp_get_thread_num();

        #pragma omp atomic
          compteur++;

        printf("Rang : %d ; compteur vaut : %d\n",rang,compteur);
    }
    printf("Au total, compteur vaut : %d\n",compteur);
    return 0;
}
```



Régions critiques

- Une région critique peut être vue comme une généralisation de la directive ATOMIC bien que les mécanismes sous-jacents soient distincts.
  - Les tâches exécutent cette région dans un ordre non-déterministe mais une à la fois.
  - Son étendue est dynamique.

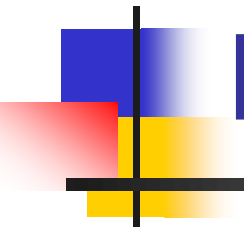
- Une région critique est définie grâce à la directive CRITICAL et s'applique à une portion de code

```
#pragma omp critical
```

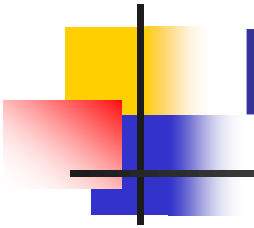
- Possibilité de créer des région critiques *nommées*
- Pour des raisons de performances, il est déconseillé d'émuler une instruction atomique par une région critique.

Régions critiques

```
#include <stdio.h>

int main()
{
    int s, p;

    s = 0, p = 1;
    #pragma omp parallel
    {
        #pragma omp critical
        {
            s++;
            p*=2;
        }
    }
        printf("Somme et produit finaux : %d, %d\n",s,p);
    return 0;
}
```

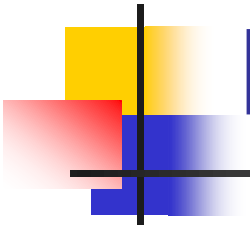


Partage de travail



Partage du travail

- En principe, la construction d'une région parallèle et l'utilisation de quelques fonctions OpenMP suffisent à eux seuls pour paralléliser une portion de code.
  - Mais il est, dans ce cas, à la charge du programmeur de répartir aussi bien le travail que les données et d'assurer la synchronisation des tâches.
  - Exemple : addition de deux vecteurs en Pthreads
- Heureusement, OpenMP propose deux directives (FOR, SECTIONS) qui permettent aisément de contrôler assez finement la répartition du travail et des données en même temps que la synchronisation au sein d'une région parallèle.
- Par ailleurs, il existe d'autres constructions OpenMP qui permettent l'exclusion de toutes les tâches à l'exception d'une seule pour exécuter une portion de code située dans une région parallèle.
- ATTENTION : le partage du travail concerne des parties indépendantes de code. Ni le compilateur, ni le runtime ne vérifie que ces parties sont bien parallélisables !



Boucle parallèle

- C'est un parallélisme par répartition du domaine d'itérations d'un nid de boucle
  - Concerne un ensemble de boucles parfaitement imbriquées
  - Restrictions sur la structure des boucles (pas de boucle *while* ni de boucles irrégulières)
  - Les indices de boucles sont des variables entières privées.
- Le mode de répartition des itérations peut être spécifié dans la clause SCHEDULE.
  - Le choix de l'ordonnancement par défaut n'est pas imposé
- Le choix du mode de répartition permet de mieux contrôler l'équilibrage de la charge de travail entre les tâches.
- Par défaut, une synchronisation globale est effectuée en fin de construction à moins d'avoir spécifié la clause NOWAIT.

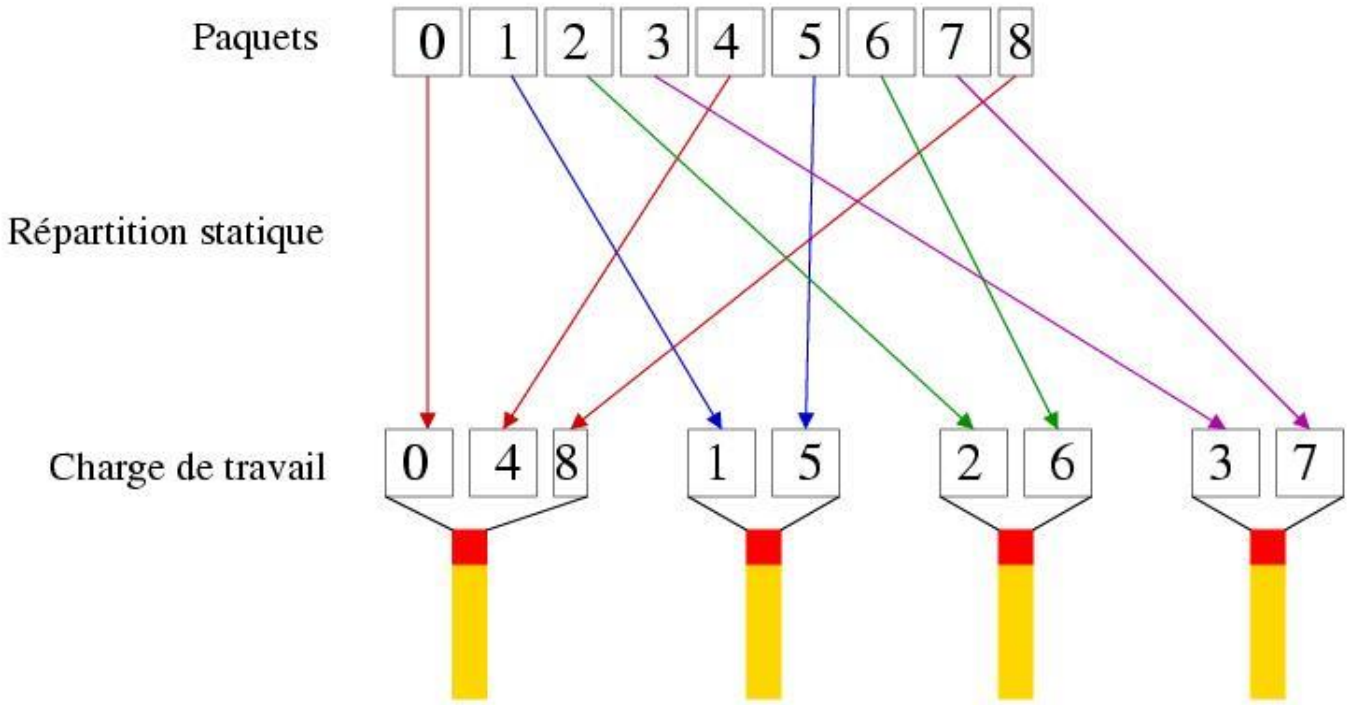
Boucle parallèle

```
#include <stdio.h>

int main( int argc, char ** argv ) {
  int N ;

  N = 10 ;

#pragma omp parallel
  {
    int i ;
#pragma omp for schedule(static)
    for ( i = 0 ; i < N ; i++ ) {
      printf( "Thread %d running iteration %d\n",
         omp_get_thread_num(), i ) ;
    }
  }
  return 0 ;
}
```

```
$ gcc –fopenmp –o prog prog.c

$ OMP_NUM_THREADS=4 ./a.out
Thread 0 running iteration 0
Thread 0 running iteration 1
Thread 0 running iteration 2
Thread 3 running iteration 9
Thread 2 running iteration 6
Thread 2 running iteration 7
Thread 2 running iteration 8
Thread 1 running iteration 3
Thread 1 running iteration 4
Thread 1 running iteration 5
```

Ordonnancement statique

- L'ordonnancement STATIC consiste à diviser les itérations en paquets d'une taille donnée appelé *chunk* (sauf peut-être pour le dernier).
  - Par défaut, le taille du *chunk* est maximale
  - Il est ensuite attribué, d'une façon cyclique à chacune des tâches, un ensemble de paquets suivant l'ordre des tâches jusqu'à concurrence du nombre total de paquets.



Ordonnancement statique

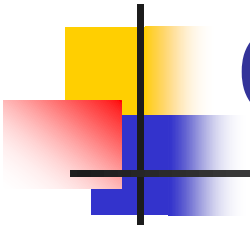
```
#include <stdio.h>

int main( int argc, char ** argv ) {
  int N ;

  N = 10 ;

#pragma omp parallel
  {
    int i ;
#pragma omp for schedule(static,1)
    for ( i = 0 ; i < N ; i++ ) {
      printf( "Thread %d running iteration %d\n",
         omp_get_thread_num(), i ) ;
    }
  }
  return 0 ;
}
```

```
$ gcc –fopenmp –o prog prog.c

$ OMP_NUM_THREADS=4 ./a.out
Thread 0 running iteration 0
Thread 0 running iteration 4
Thread 0 running iteration 8
Thread 3 running iteration 3
Thread 3 running iteration 7
Thread 2 running iteration 2
Thread 2 running iteration 6
Thread 1 running iteration 1
Thread 1 running iteration 5
Thread 1 running iteration 9
```

Ordonnancement statique

```
#include <stdio.h>

int main( int argc, char ** argv ) {
  int N ;

  N = 10 ;

#pragma omp parallel
  {
    int i ;
#pragma omp for schedule(static,2)
    for ( i = 0 ; i < N ; i++ ) {
      printf( "Thread %d running iteration %d\n",
         omp_get_thread_num(), i ) ;
    }
  }
  return 0 ;
}
```

```
$ gcc –fopenmp –o prog prog.c

$ OMP_NUM_THREADS=4 ./a.out
Thread 0 running iteration 0
Thread 0 running iteration 1
Thread 0 running iteration 8
Thread 0 running iteration 9
Thread 3 running iteration 6
Thread 3 running iteration 7
Thread 1 running iteration 2
Thread 1 running iteration 3
Thread 2 running iteration 4
Thread 2 running iteration 5
```



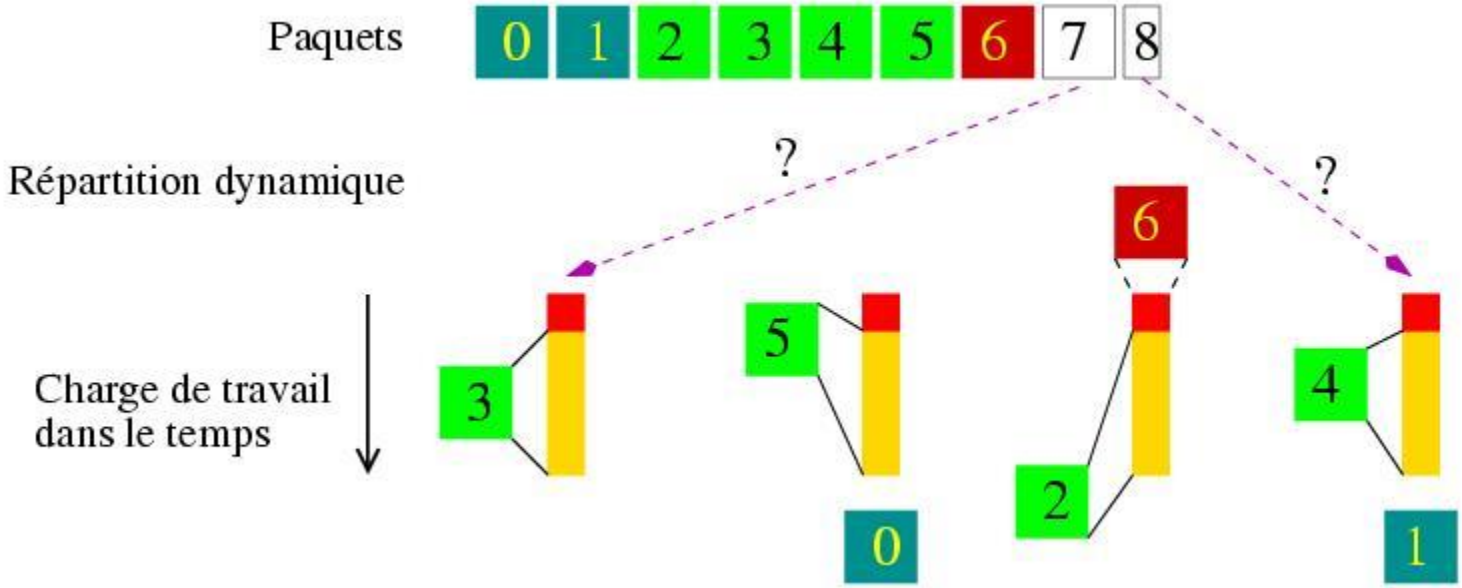
Clause schedule

- Nous aurions pu différer à l'exécution le choix du mode de l'ordonnancement des itérations à l'aide de la variable d'environnement OMP_SCHEDULE.
  - Fonction également disponible `omp_set_schedule()`
- Le choix de l'ordonnancement des itérations d'une boucle peut être un atout majeur pour l'équilibrage de la charge de travail sur une machine dont les processeurs ne sont pas dédiés.
  - La taille du *chunk* joue également un rôle important dans les performances

Clause schedule

- DYNAMIC : les itérations sont divisées en paquets de taille donnée. Sitôt qu'une tâche épuise ses itérations, un autre paquet lui est attribué.

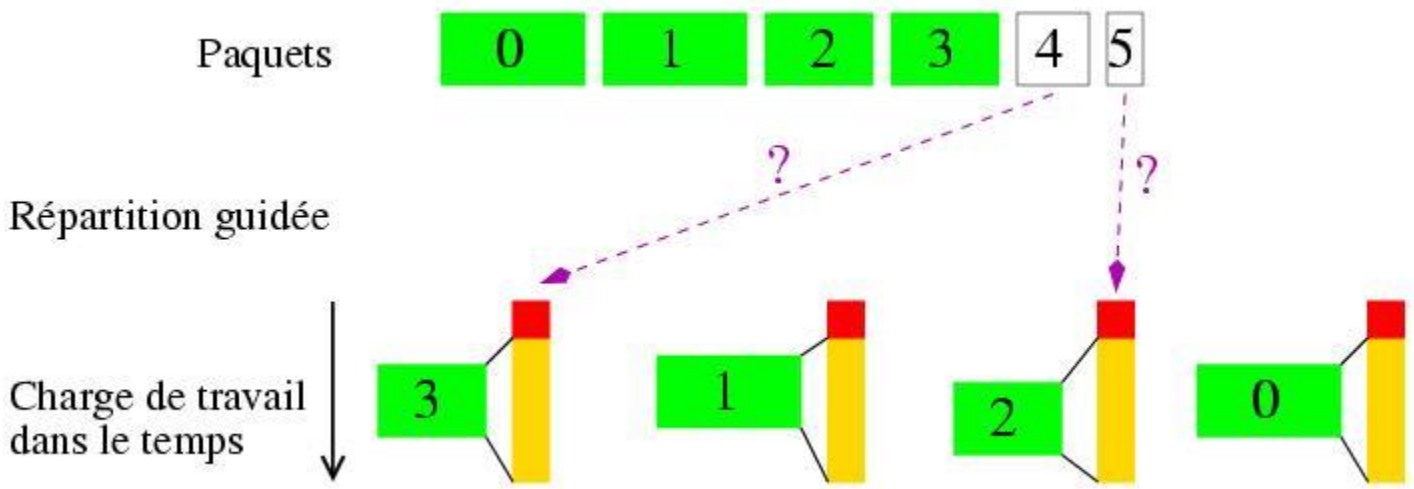
```
$ export OMP_SCHEDULE="DYNAMIC,480"
$ export OMP_NUM_THREADS=4 ; ./prog
```

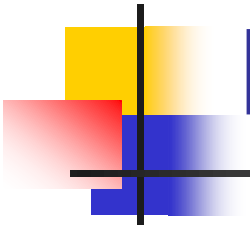


Clause schedule

- GUIDED : les itérations sont divisées en paquets dont la taille décroît exponentiellement. Tous les paquets ont une taille supérieure ou égale à une valeur donnée à l'exception du dernier dont la taille peut être inférieure. Sitôt qu'une tâche finit ses itérations, un autre paquet d'itérations lui est attribué.

```
> export OMP_SCHEDULE="GUIDED,256"
> export OMP_NUM_THREADS=4 ; ./prog
```



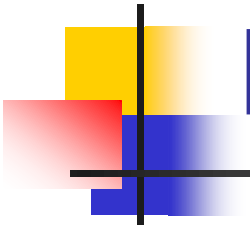


Exécution ordonnée

- Il est parfois utile d'exécuter une partie d’une boucle d'une façon ordonnée.
  - Pour le débuggage
  - Pour les I/O
- L'ordre des itérations sera alors identique à celui correspondant à une exécution séquentielle.
  - Seul le bout de code en question est concerné

Exécution ordonnée

```
#include <stdio.h>
#include <omp.h>
#define N 9
int main() {
    int i, rang;

    #pragma omp parallel default(none) private(rang,i)
    {
        rang=omp_get_thread_num();
        #pragma omp for schedule(runtime) ordered nowait
        for (i=0; i<N; i++) {
            #pragma omp ordered
            {
                printf("Rang : %d ; iteration : %d\n",rang,i);
            }
        }
    }
    return 0;
}
```



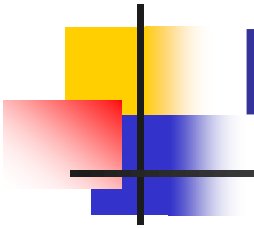
Réduction

- Une réduction est une opération associative appliquée à une variable partagée.
- L'opération peut être :
  - arithmétique : +, --, * ;
  - logique : .AND., .OR., .EQV., .NEQV. ;
  - une fonction intrinsèque : MAX, MIN, IAND, IOR, IEOR.
- Chaque tâche calcule un résultat partiel indépendamment des autres. Elles se synchronisent ensuite pour mettre à jour le résultat final.

Réduction

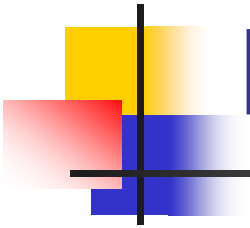
```
#include <stdio.h>
#define N 5
int main()
{
  int i, s=0, p=1, r=1;

  #pragma omp parallel
  {
    #pragma omp for reduction(+:s) reduction(*:p,r)
    for (i=0; i<N; i++) {
      s = s + 1;
      p = p * 2;
      r = r * 3;
    }
  }
  printf("s = %d ; p = %d ; r = %d\n",s,p,r);
  return 0;
}
```

```
$ gcc –o prog –fopenmp prog.c

$ export OMP_NUM_THREADS=4
$ ./prog
s = 5 ; p = 32 ; r = 243
```



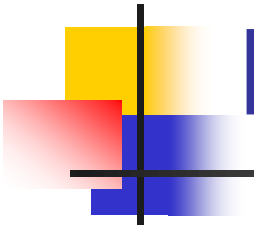
Flot de données

- La construction FOR accepte également des clauses concernant le flot des données
  - PRIVATE : pour attribuer à une variable un statut privé ;
  - FIRSTPRIVATE : privatise une variable partagée dans l'étendue de la construction DO et lui assigne la dernière valeur affectée avant l'entrée dans cette région ;
  - LASTPRIVATE : privatise une variable partagée dans l'étendue de la construction DO et permet de conserver, à la sortie de cette construction, la valeur calculée par la tâche exécutant la dernière itération d'une boucle.



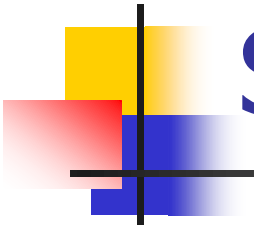
Région combinée

- La directive PARALLEL FOR est une fusion des directives PARALLEL et FOR munie de l'union de leurs clauses respectives.
- La fin du bloc inclut une barrière globale de synchronisation et ne peut admettre la clause NOWAIT.



Nid de boucles

- Depuis la norme 3.0, il est possible de partager le travail d'un ensemble de boucles imbriqués (nid de boucles)
- Utilisation de la clause collapse(int)
  - L'entier en paramètre définit la profondeur du nid de boucles càd le nombre de boucles imbriqués
  - Le nid de boucles doit être parfait

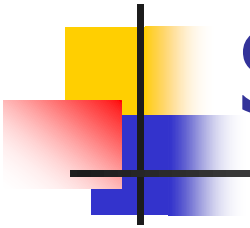


Sections parallèles

- Une section est une portion de code exécutée par une et une seule tâche.
- Plusieurs portions de code peuvent être définies par l'utilisateur à l'aide de la directive SECTION au sein d'une construction SECTIONS.
- Le but est de pouvoir répartir l'exécution de plusieurs portions de code indépendantes sur les différentes tâches.
- La clause NOWAIT est admise en fin de construction pour lever la barrière de synchronisation implicite.

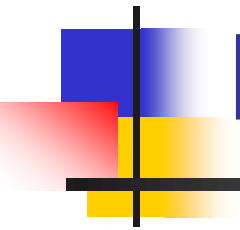
Sections parallèles

```
int main() {
    int i, rang;
    float pas_x, pas_y;
    float coord_x[M],coord_y[N];
    float a[M][N], b[M][N];

    #pragma omp parallel private(rang) num_threads(3)
    {
        rang=omp_get_thread_num();
        #pragma omp sections nowait
        {
            #pragma omp section
            {
                lecture_champ_initial_x(a);
                printf("Tâche numéro %d : init. champ en X\n",rang);
            }
            #pragma omp section
            {
                lecture_champ_initial_y(b);
                printf("Tâche numéro %d : init. champ en Y\n",rang);
            }
        }
    }
    return 0;
}
```

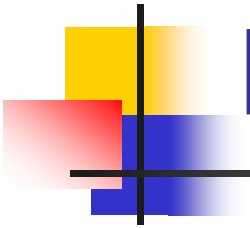


Sections parallèles

- Toutes les directives SECTION doivent apparaître dans l'étendue lexicale de la construction SECTIONS.
- Les clauses admises dans la directive SECTIONS sont celles que nous connaissons déjà :
  - PRIVATE ;
  - FIRSTPRIVATE ;
  - LASTPRIVATE ;
  - REDUCTION.
- La directive PARALLEL SECTIONS est une fusion des directives PARALLEL et SECTIONS munie de l'union de leurs clauses respectives.

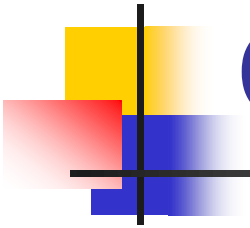


Exécution exclusive



Exécution exclusive

- Il arrive que l'on souhaite exclure toutes les tâches à l'exception d'une seule pour exécuter certaines portions de code incluses dans une région parallèle.
- Pour se faire, OpenMP offre deux directives SINGLE et MASTER.
- Bien que le but recherché soit le même, le comportement induit par ces deux constructions reste assez différent.

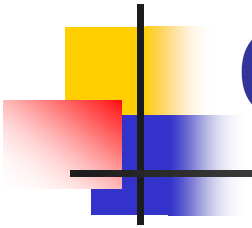


Construction *master*

- La construction MASTER permet de faire exécuter une portion de code par la tâche maître seule.
- Cette construction n'admet aucune clause.
- Il n'existe aucune barrière de synchronisation ni en début (MASTER) ni en fin de construction (END MASTER).

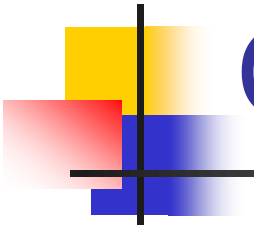
Construction *master*

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

int main()
{
    int rang;
    float a;

    #pragma omp parallel private(a,rang)
    {
        a = 92290.;

        #pragma omp master
        {
            a = -92290.;
        }

        rang=omp_get_thread_num();
        printf("Rang : %d ; A vaut : %f\n",rang,a);
    }
    return 0;
}
```



Construction *single*

- La construction SINGLE permet de faire exécuter une portion de code par une et une seule tâche sans pouvoir indiquer laquelle.
  - Généralisation de MASTER
- En général, c'est la tâche qui arrive la première sur la construction SINGLE mais cela n'est pas spécifié dans la norme.
- Toutes les tâches n'exécutant pas la région SINGLE attendent, en fin de construction, la terminaison de celle qui en a la charge, à moins d'avoir spécifié la clause NOWAIT.



Construction *single*

- Une clause supplémentaire admise est la clause COPYPRIVATE.
- Elle permet à la tâche chargée d'exécuter la région SINGLE, de diffuser aux autres tâches la valeur d'une liste de variables privées avant de sortir de cette région.
- Les autres clauses admises par la directive SINGLE sont PRIVATE et FIRSTPRIVATE.

Quelques pièges

```
#include <stdio.h>

int main()
{
    float s;

    #pragma omp parallel default(none) shared(s)
    {
        #pragma omp single
        {
            s=1.;
        }
        printf(s = %f\n",s);
        s=2.;
    }
    return 0;
}
```