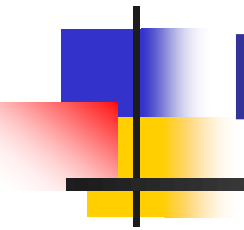
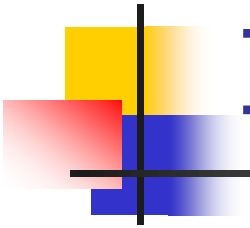


Programmation Parallèle et Distribuée

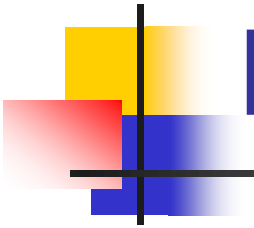
Cours 8 : Introduction à OpenMP

Patrick Carribault
David Dureau
Marc Pérache (marc.perache@cea.fr)



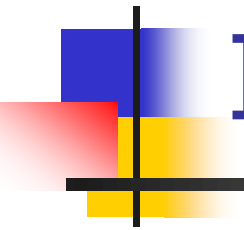
Introduction

- Programmation à mémoire partagée
  - Basée sur des threads
  - API POSIX étudiée
- Modèle de programmation OpenMP
  - Basée sur des directives
  - Utilisation implicite de threads
  - Permet de paralléliser un code séquentiel

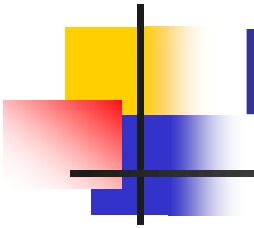


Plan du cours 8

- Introduction à OpenMP
  - Histoire
  - Concept générale
- Région parallèle
  - Modèle fork/join
  - Premiers exemples
- Flot de données
  - Gestion de la visibilité des données
- Synchronisations
  - Verrou
  - Sections critique
  - Barrière

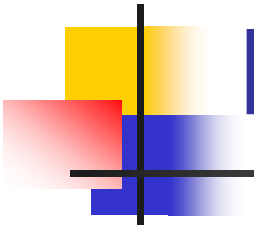


Introduction à OpenMP



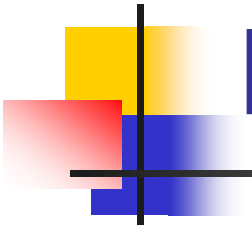
Définition

- 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
- La programmation parallèle hybride peut être réalisée par exemple en utilisant à la fois OpenMP et MPI.



Historique

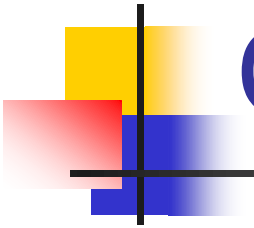
- Parallélisation multitâches existante depuis longtemps chez certains constructeurs (ex. CRAY, NEC, IBM, ...)
  - Chacun avait son propre jeu de directives.
- Retour en force des machines multiprocesseurs à mémoire partagée
  - Définition un standard.
- Tentative de standardisation de PCF (Parallel Computing Forum)
  - Jamais été adoptée par les instances officielles de normalisation.
- Le 28 octobre 1997, une majorité importante d'industriels et de constructeurs ont adopté OpenMP (Open Multi Processing) comme un standard dit « industriel ».



Historique

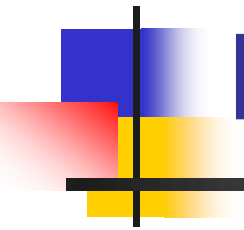
- Les spécifications d'OpenMP appartiennent aujourd'hui à l'ARB (Architecture Review Board), seul organisme chargé de son évolution.
  - http://www.openmp.org
  - http://www.compunity.org

- Une version OpenMP-2 a été finalisée en novembre 2000. Elle apporte surtout des extensions relatives à la parallélisation de certaines constructions Fortran 95.
- La version OpenMP-3 datant de mai 2008 introduit essentiellement le concept de tâche.
- La version OpenMP-4 de juillet 2013 apporte de nombreuses nouveautés, avec notamment le support des accélérateurs, des dépendances entre les tâches, la programmation SIMD (vectorisation) et l'optimisation du placement des threads.



Concepts généraux

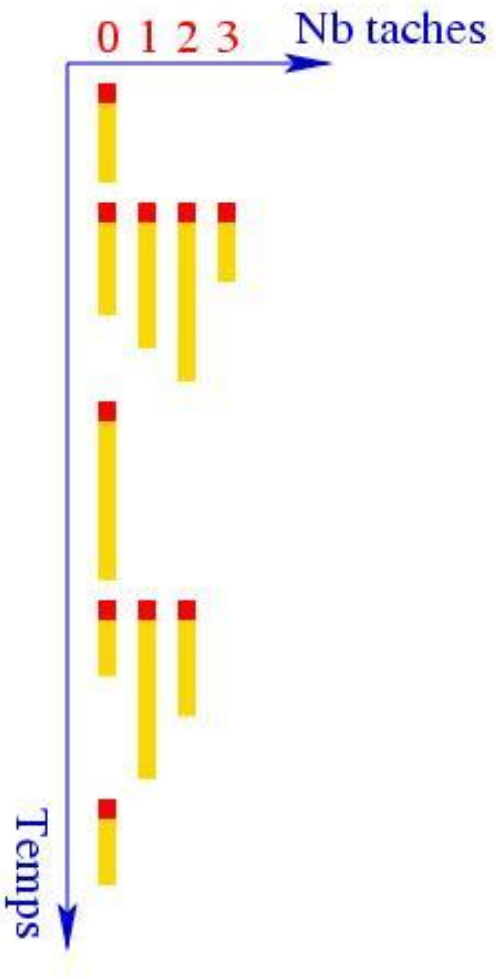
- Un programme OpenMP est exécuté par un processus unique.
  - Ce processus active des processus légers (threads) à l'entrée d'une région parallèle
- Chaque processus léger exécute une tâche implicite composée d'un ensemble d'instructions.
  - Notion de rang
- Pendant l'exécution d'une tâche, une variable 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



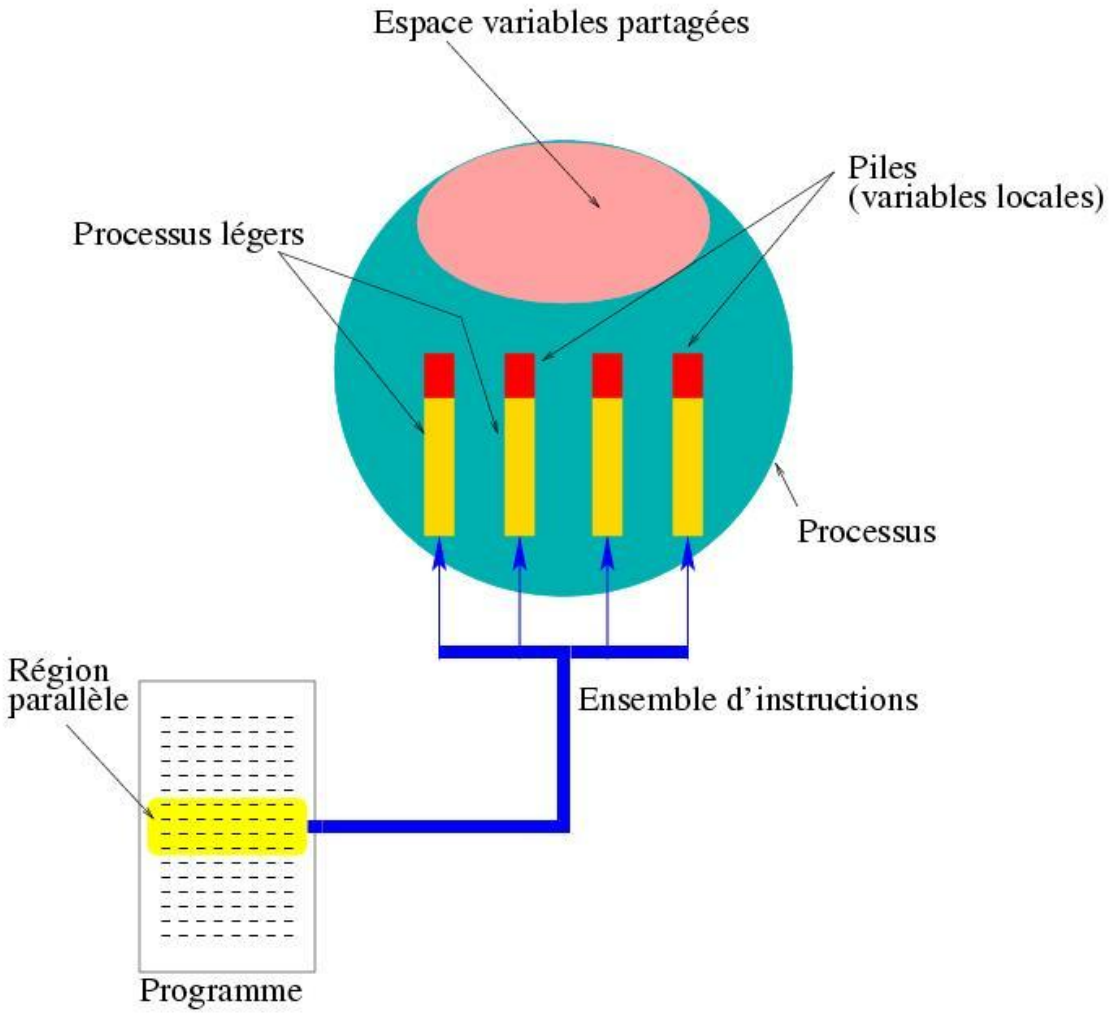
Région parallèle

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.

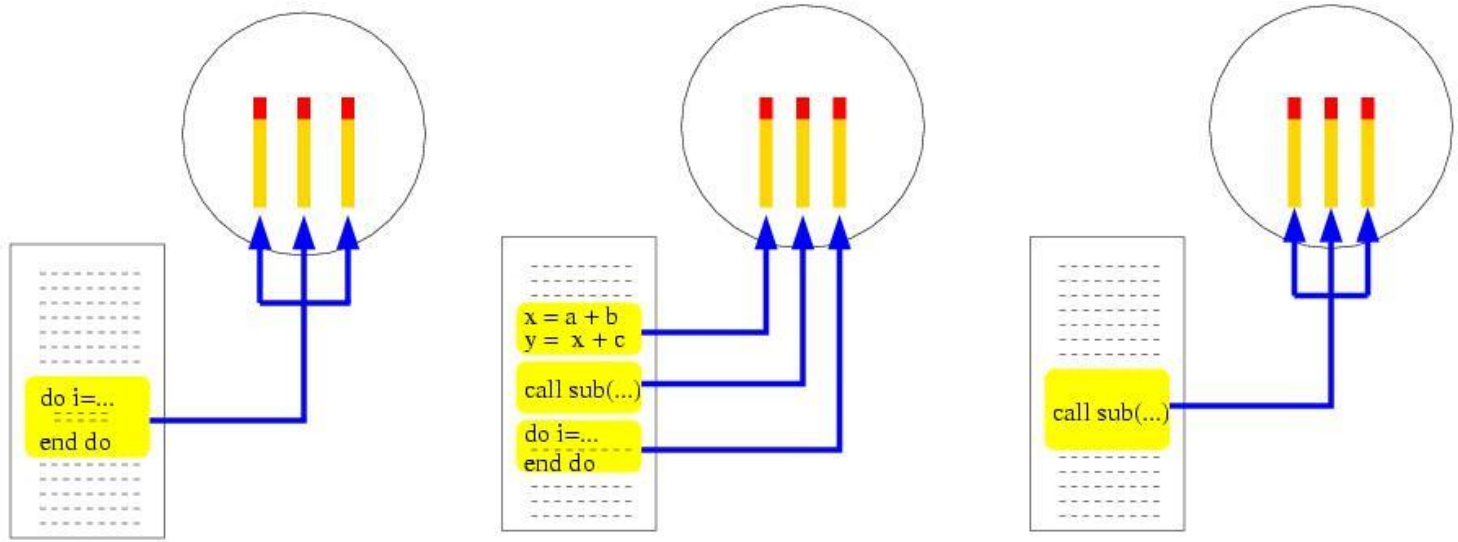


Région parallèle



Partage du travail

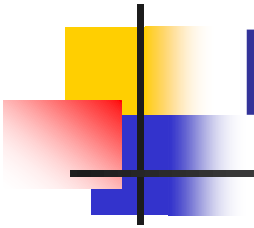
- Le partage du travail consiste essentiellement à :
  - Exécuter une boucle par répartition des itérations entre les tâches;
  - Exécuter plusieurs sections de code mais une seule par tâche;
  - Exécuter des tâches explicites différentes



Boucle parallèle (Looplevel parallelism)

Sections parallèles

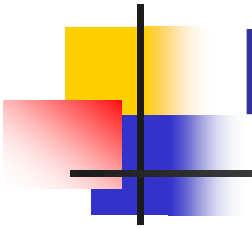
Procédure parallèle (orphaning)



Placement des threads

- Généralement, les threads sont affectés aux processeurs par le système d'exploitation. Différents cas peuvent se produire :
  - Au mieux, à chaque instant, il existe une tâche par processeur avec autant de tâches que de processeurs dédiés pendant toute la durée du travail
  - Au pire, toutes les tâches sont traitées séquentiellement par un et un seul processeur ;
  - En réalité, pour des raisons essentiellement d'exploitation sur une machine dont les processeurs ne sont pas dédiés, la situation est en général intermédiaire.

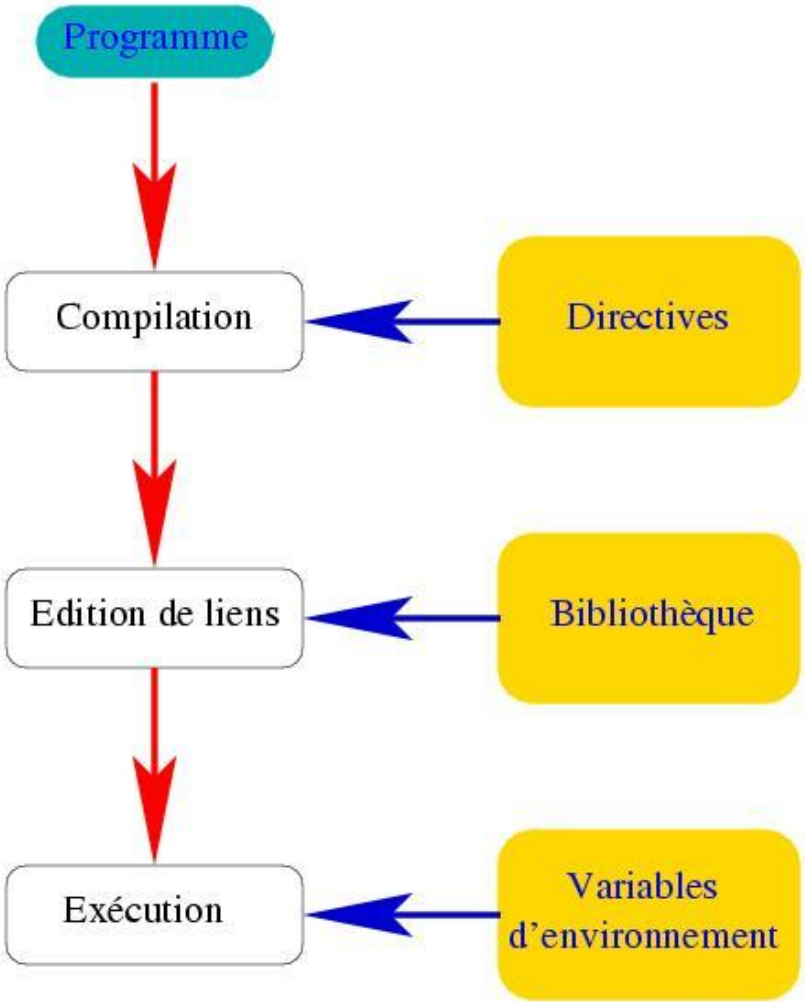
- Pour palier à ces problèmes, il est possible de construire le *runtime* OpenMP sur une bibliothèque de threads mixtes et ainsi contrôler l'ordonnancement des tâches.

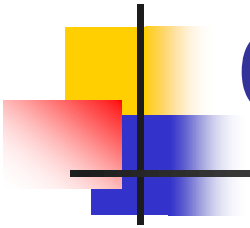


Directives & Environnement

- 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.

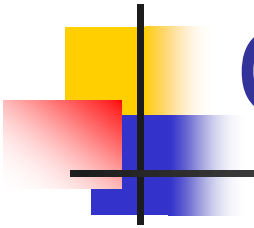
Compilation





Comparaison MPI/OpenMP

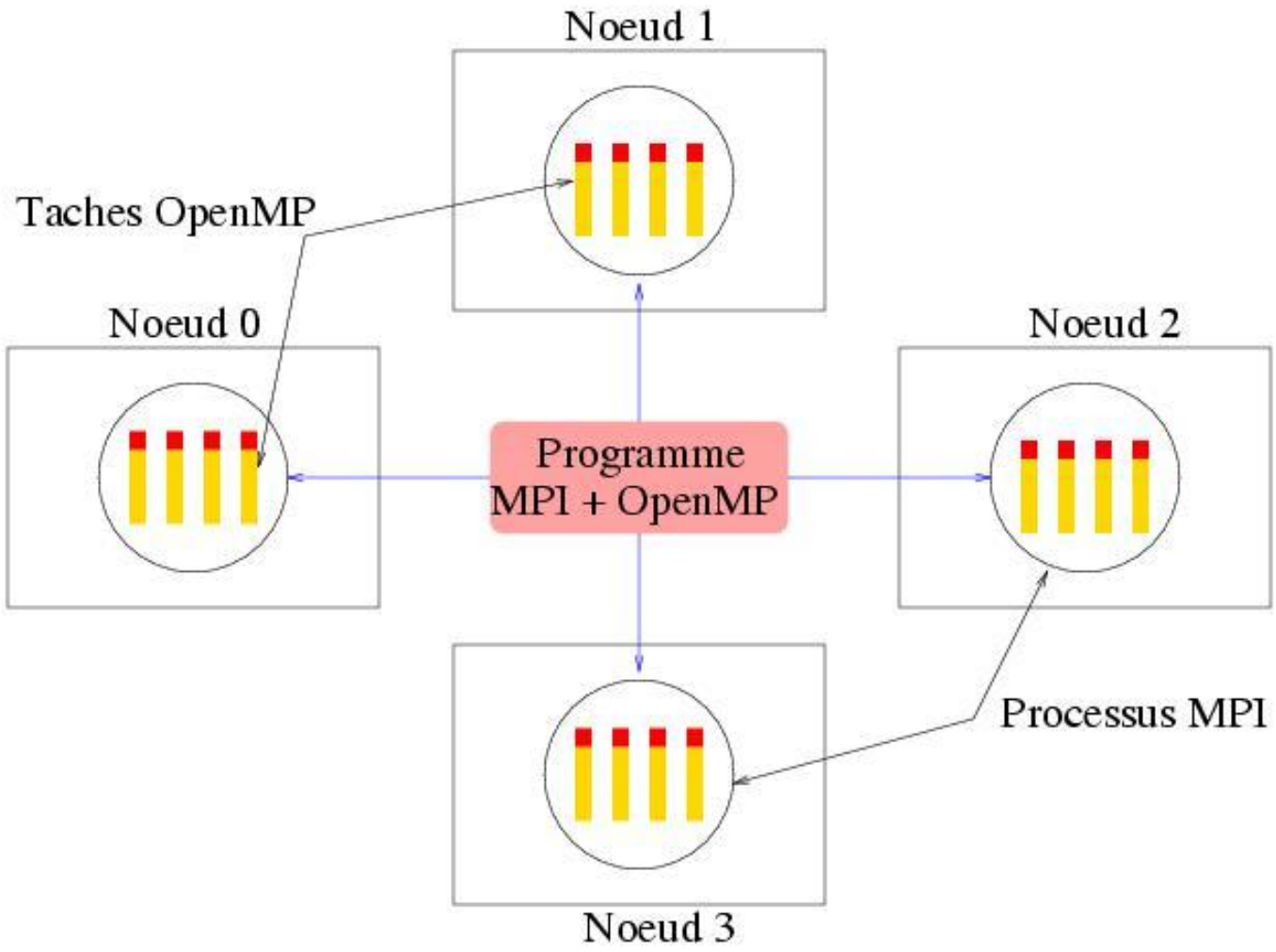
- Modèles complémentaires de parallélisation.
  - OpenMP, comme MPI, possède une interface Fortran, C et C++.
  - MPI est un modèle multiprocessus dont le mode de communication entre les processus est explicite (la gestion des communications est à la charge de l'utilisateur).
  - OpenMP est un modèle multitâches dont le mode de communication entre les tâches est implicite (la gestion des communications est à la charge du compilateur).



Comparaison MPI/OpenMP

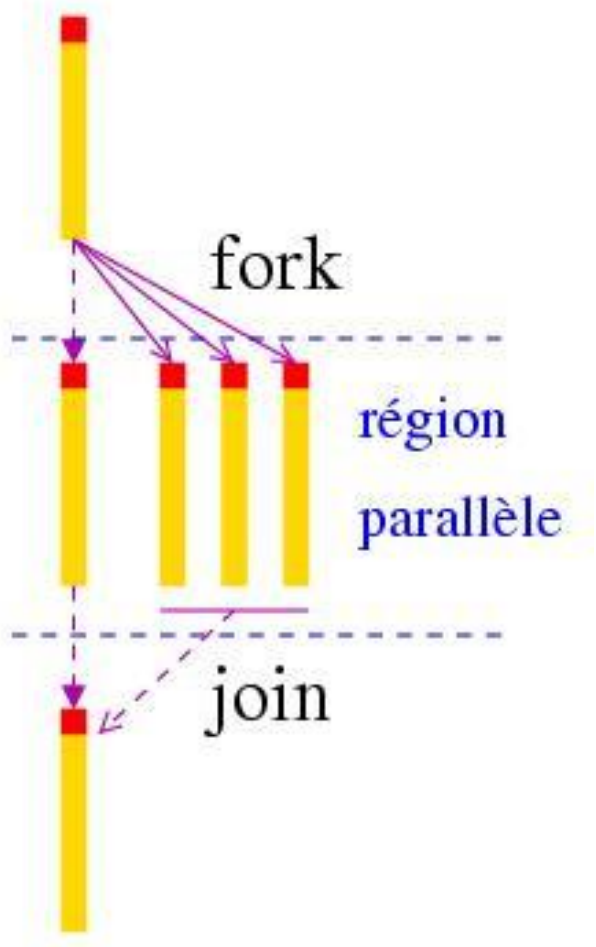
- MPI est utilisé en général sur des machines multiprocesseurs à mémoire distribuée.
- OpenMP est utilisé sur des machines multiprocesseurs à mémoire partagée.
- Sur une grappe de machines indépendantes (noeuds) multiprocesseurs à mémoire partagée, la mise en oeuvre d'une parallélisation à deux niveaux (MPI, OpenMP) dans un même programme peut être un atout majeur pour les performances parallèles du code.

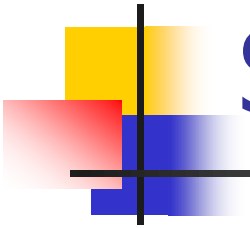
Programmation hybride



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.

Premier programme

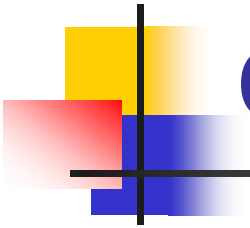
```
#include <omp.h>
#include <stdio.h>

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

Header

Directive

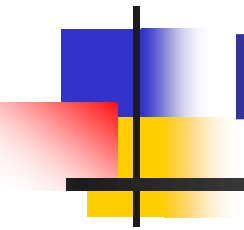
Région parallèle

```
$ 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.



Flôt de données

Second programme

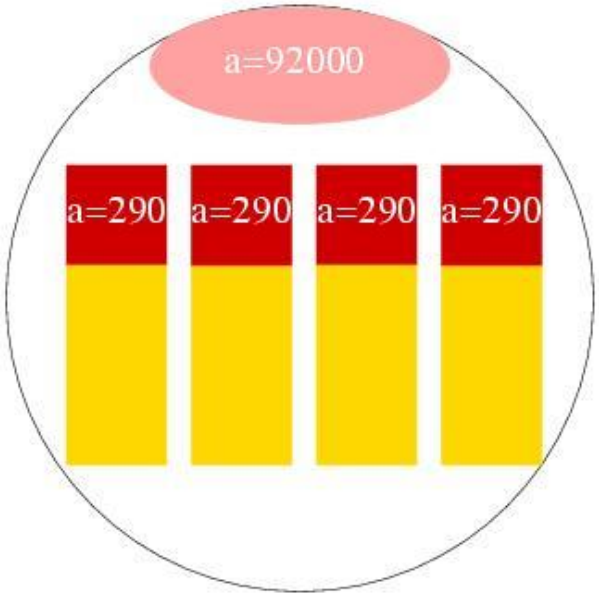
```
#include <stdio.h>
#include <omp.h>

int main() {
  float a;
  int   p;
  a = 92290. ; p = 0;
#pragma omp parallel
  {
#ifdef _OPENMP
    p=omp_in_parallel();
#endif
    printf("a vaut : %f ; p vaut : %d\n",a,p);
  }
  return 0;
}
```

```
$ gcc –o prog -fopenmp prog.c

$ export OMP_NUM_THREADS=4
$ ./prog
a vaut : 92290. ; p vaut : 1
a vaut : 92290. ; p vaut : 1
a vaut : 92290. ; p vaut : 1
a vaut : 92290. ; p vaut : 1
```

Fonction OpenMP

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

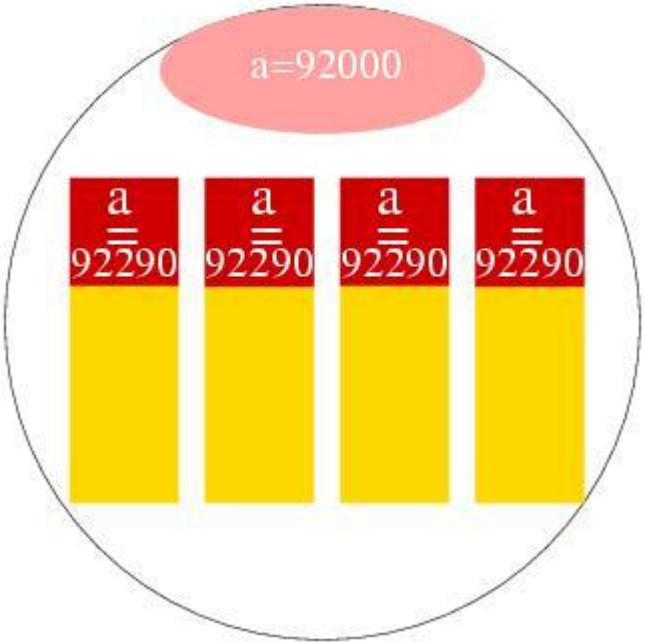
```
#include <stdio.h>
#include <omp.h>

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

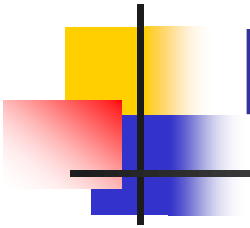
- Cependant, grâce à la clause FIRSTPRIVATE, il est possible de forcer l'initialisation de cette variable privée à la dernière valeur qu'elle avait avant l'entrée dans la région parallèle.



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.
```



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 ».

Étendue d'une région parallèle

```
#include <omp.h>

void sub(void);

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

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

void sub(void) {
  int p=0;
#ifdef _OPENMP
  p = omp_in_parallel();
#endif
  printf("Parallele ? :
   %d\n",  p);
}
```

```
$ gcc –o prog -fopenmp prog.c sub.c
$ export OMP_NUM_THREADS=4
$ ./prog
Parallele ? : 1
Parallele ? : 1
Parallele ? : 1
Parallele ? : 1
```

Étendue d'une région parallèle

- Dans un sous-programme appelé dans une région parallèle, les variables locales et automatiques sont implicitement privées à chacune des tâches

```
int main() {
#pragma omp parallel
   default(shared)
{
  sub();
}
return 0;
}
```

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

void sub(void) {
int a;

a=92290;
a = a +
   omp_get_thread_num();
printf("a vaut : %d\n",
   a);
}
```

```
$ ./prog
a vaut : 92293
a vaut : 92291
a vaut : 92292
a vaut : 92290
```

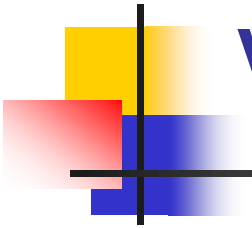
Transmission par argument

- Dans une procédure, toutes les variables transmises par argument par référence, héritent du statut défini dans l'étendue lexicale (statique) de la région.

```
#include <stdio.h>
int main() {
  int a, b;
  a = 92000;
#pragma omp parallel shared(a) \
   private(b)
  {
    sub(a, &b);
    printf("b vaut : %d\n",b);
  }
  return 0;
}
```

```
#include <omp.h>

void sub(int x, int *y)
{
*y = x +
omp_get_thread_num();
}
```

```
$ ./prog
b vaut : 92003
b vaut : 92001
b vaut : 92002
b 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

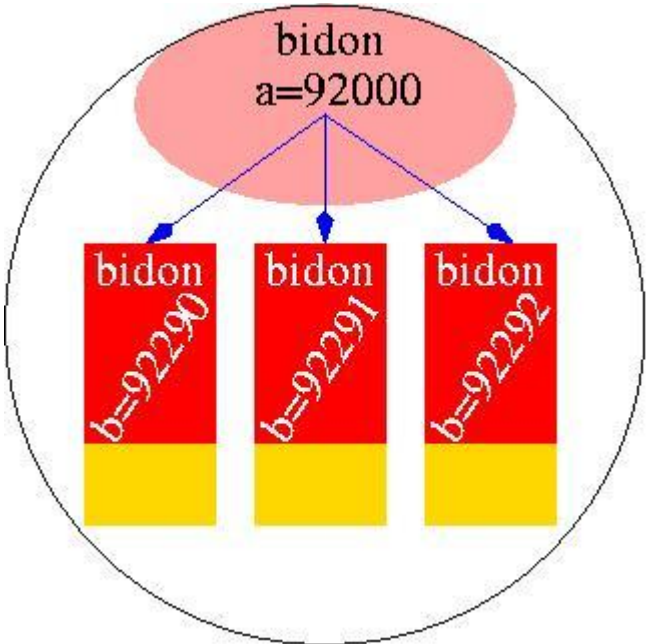
```
#include <omp.h>
float a;

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.

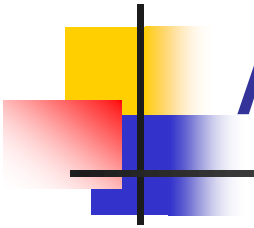


Variables statiques

```
#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
```



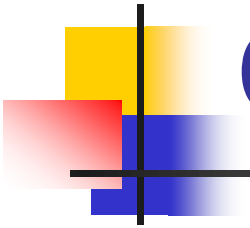
Allocation dynamique

- L'opération d'allocation/désallocation dynamique de mémoire peut être effectuée à l'intérieur d'une région parallèle.
- Si l'opération porte sur une variable privée, celle-ci sera locale à chaque tâche.
- Si l'opération porte sur une variable partagée, il est alors plus prudent que seule une tâche (ex. la tâche maître) se charge de cette opération

Allocation dynamique

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(){
  int n, debut, fin, rang, nb_taches, i;
  float *a;
  n=1024, nb_taches=4;
  a=(float *) malloc(n*nb_taches*sizeof(float));
  #pragma omp parallel default(none) \
     private(debut,fin,rang,i) shared(a,n) if(n > 512)
  {
    rang=omp_get_thread_num();
    debut=rang*n;
    fin=(rang+1)*n;
    for (i=debut; i<fin; i++)
       a[i] = 92291. + (float) i;
    printf("Rang : %d ; A[%.4d],...,A[%.4d] : %#.0f,...,%#.0f \n",
           rang,debut,fin-1,a[debut],a[fin-1]);
  }
  free(a);
  return 0;
}
```

```
$ OMP_NUM_THREADS=4 ./prog
Rang : 3 ; A[3072],...,A[4095] : 95363.,...,96386.
Rang : 0 ; A[0000],...,A[1023] : 92291.,...,93314.
Rang : 1 ; A[1024],...,A[2047] : 93315.,...,94338.
Rang : 2 ; A[2048],...,A[3071] : 94339.,...,95362.
```



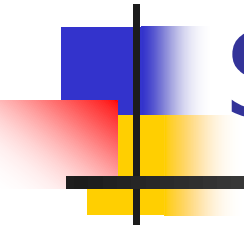
Compléments

- 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.

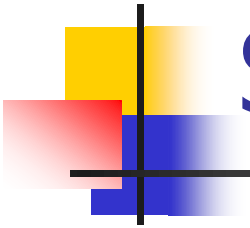
Compléments

```
#include <omp.h>
int main() {
  int rang;
  #pragma omp parallel private(rang) \
     num_threads(3)
  {
    rang=omp_get_thread_num();
    printf(
      "Mon rang dans region 1 : %d \n",rang);
    #pragma omp parallel private(rang) \
      num_threads(2)
    {
      rang=omp_get_thread_num();
      printf(
       "    Mon rang dans region 2 : %d \n",rang);
    }
  }
  return 0;
}
```

```
$ gcc –o prog -fopenmp prog.c
$ export OMP_DYNAMIC=true
$ export OMP_NESTED=true
$ ./prog

Mon rang dans region 1 : 0
        Mon rang dans region 2 : 1
        Mon rang dans region 2 : 0
Mon rang dans region 1 : 2
        Mon rang dans region 2 : 1
        Mon rang dans region 2 : 0
Mon rang dans region 1 : 1
        Mon rang dans region 2 : 0
        Mon rang dans region 2 : 1
```

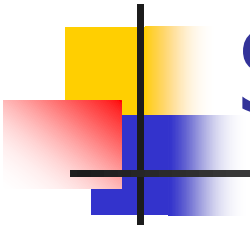


Synchronisations



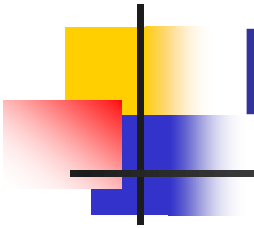
Synchronisations

- 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.

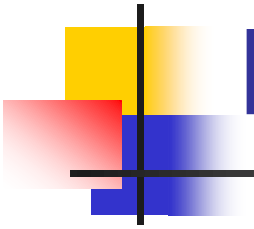


Barrière

- 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.

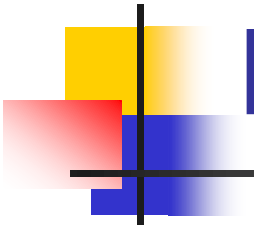
```
#pragma omp atomic
```

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

Mise à jour atomique

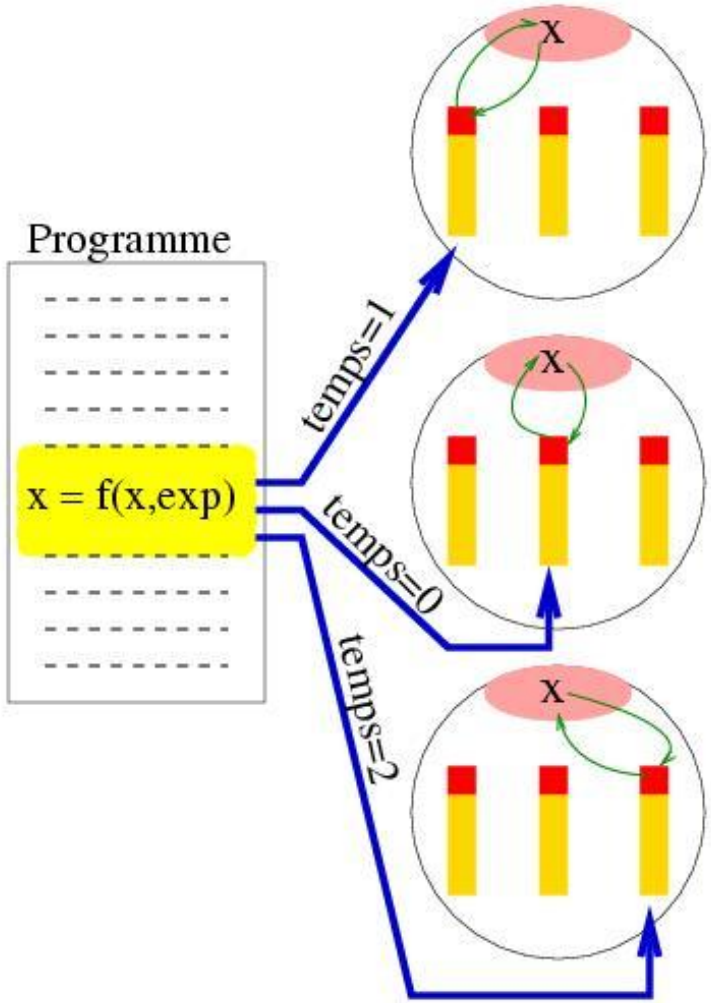
```
#include <stdio.h>
#include <omp.h>

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;
}
```

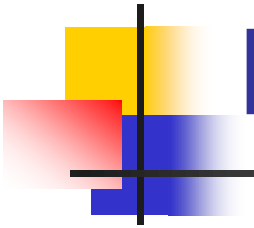
Mise à jour atomique



- L'instruction en question doit avoir l'une des formes suivantes :
  - x=x~(op)~exp ;
  - x=exp~(op)~x ;
  - x=f(x,exp) ;
  - x=f(exp,x) ;
- `(op)` représente l'une des opérations suivantes : +, --, *, /, .AND., .OR., .EQV., .NEQV..
- `f` représente une des fonctions intrinsèques suivantes : MAX, MIN, IAND, IOR, IEOR.
- `exp` est une expression arithmétique quelconque indépendante de x.

Mise à jour atomique





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
```

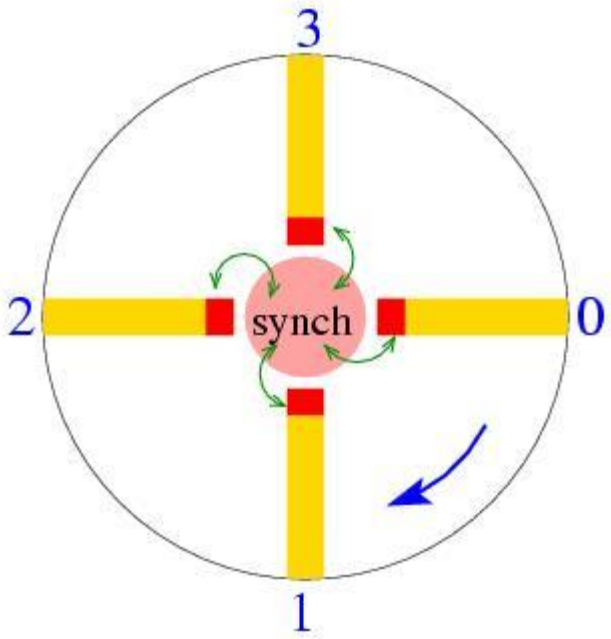
- Possibilité de créer des région critiques *nommées*
- Pour des raisons de performances, il est déconseillé d'émuler une instruction atomique par une région critique.

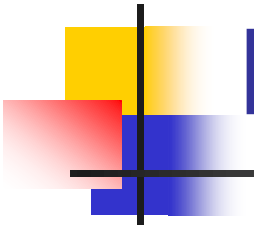
Régions critiques

```
#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;
}
```

Directive FLUSH

- Elle est utile dans une région parallèle pour rafraîchir la valeur d'une variable partagée en mémoire globale.
- Elle est d'autant plus utile que la mémoire d'une machine est hiérarchisée.
- Elle peut servir à mettre en place un mécanisme de point de synchronisation entre les tâches.





Résumé

- Introduction à OpenMP
  - Modèle de programmation parallèle à mémoire partagée
  - Gestion implicite de threads
  - Version courante : 4.0 (parallalélisme de données et de tâches)
- Flot de données à contrôler par le programmeur
- Mécanismes de synchronisations