Programmation Parallèle et Distribuée: OpenMP

PERACHE Marc
marc.perache@cea.fr

Open Multi Processing

- Description:

OpenMP (Open Multi-Processing) est une interface de programmation pour le calcul parallèle sur architecture à mémoire partagée. Cette API est supportée sur de nombreuses plateformes, incluant Unix et Windows, pour les langages de programmation C/C++ et Fortran. Il se présente sous la forme d'un ensemble de directives, d'une bibliothèque logicielle et de variables d'environnement.

OpenMP est portable et dimensionnable. Il permet de développer rapidement des applications parallèles à petite granularité en restant proche du code séquentiel.

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

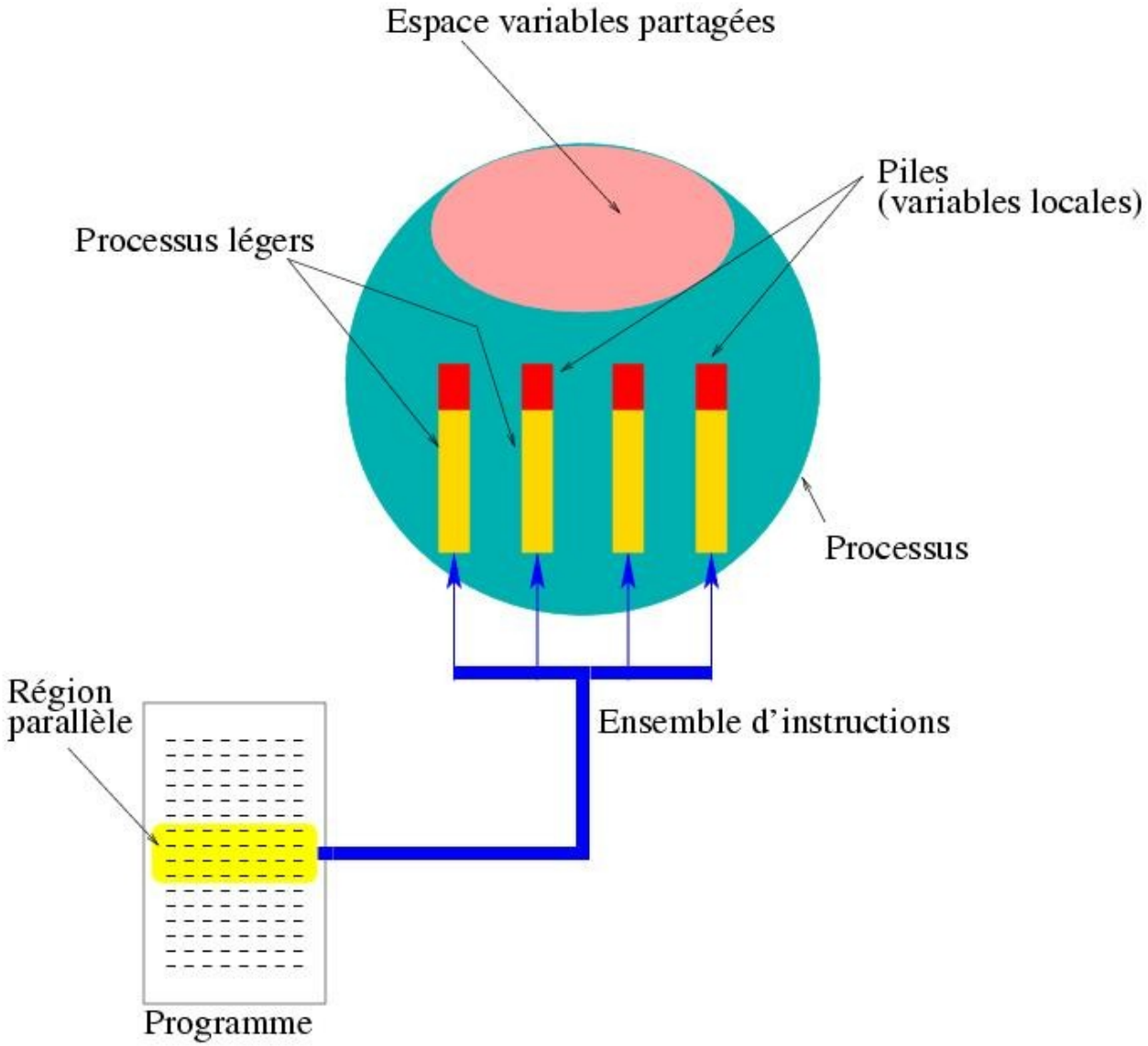
Historique

- La parallélisation multitâches existait depuis longtemps chez certains constructeurs (ex. CRAY, NEC, IBM, ...), mais chacun avait son propre jeu de directives.
- Le retour en force des machines multiprocesseurs à mémoire partagée a poussé à définir un standard.
- La tentative de standardisation de PCF (Parallel Computing Forum) n'a 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 ».
- Les spécifications d'OpenMP appartiennent aujourd'hui à l'ARB (Architecture Review Board), seul organisme chargé de son évolution.
- Une version OpenMP-2 a été finalisée en novembre 2000. Elle apporte surtout des extensions relatives à la parallélisation de certaines constructions Fortran 95.

Concepts généraux

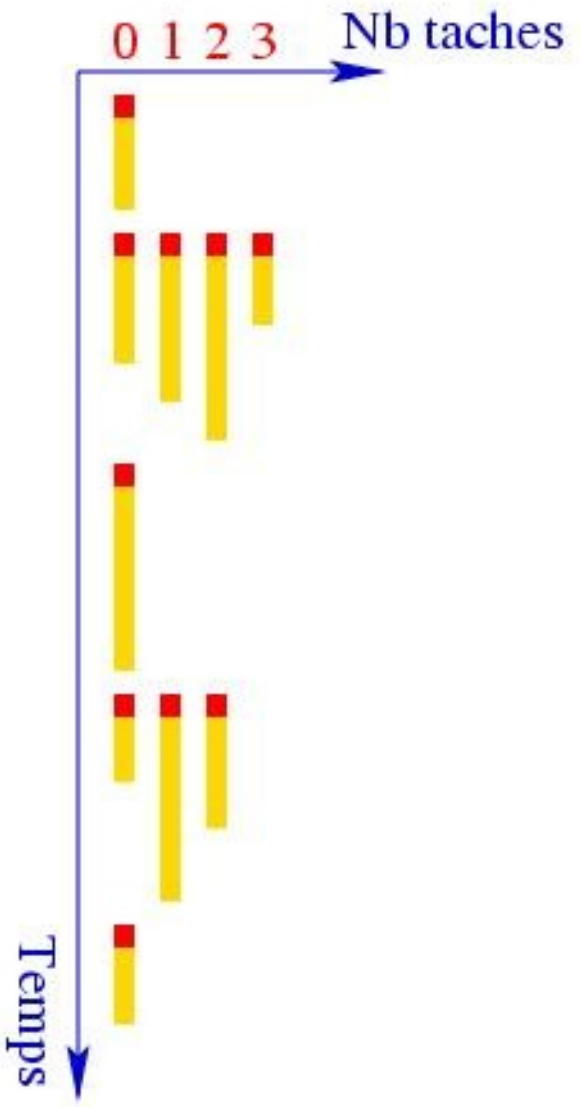
- Un programme OpenMP est exécuté par un processus unique.
- Ce processus active des processus légers (threads) à l'entrée d'une région parallèle.
- Chaque processus léger exécute une tâche composée d'un ensemble d'instructions.
- Pendant l'exécution d'une tâche, une variable peut être lue et/ou modifiée 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 par

Concepts généraux



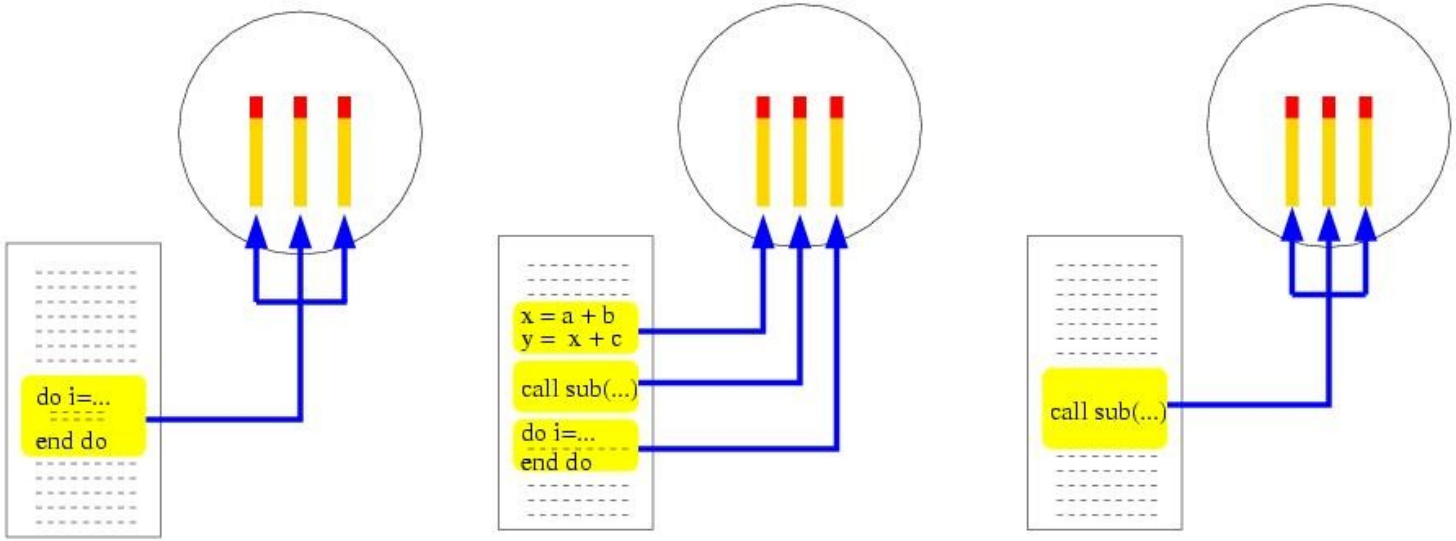
Concepts généraux

- 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.



Concepts généraux

- 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 (orphaning).



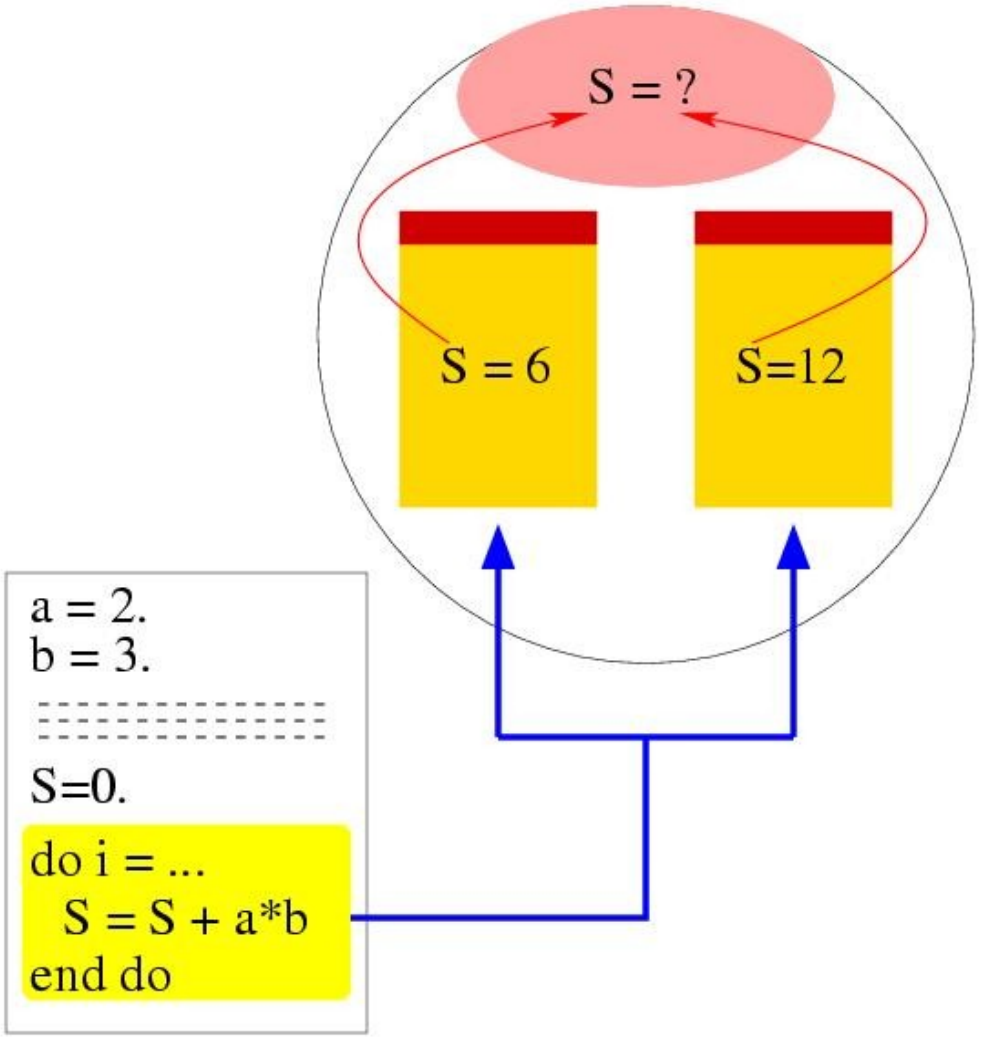
Boucle parallèle (Looplevel parallelism)

Sections parallèles

Procédure parallèle (orphaning)

Concepts généraux

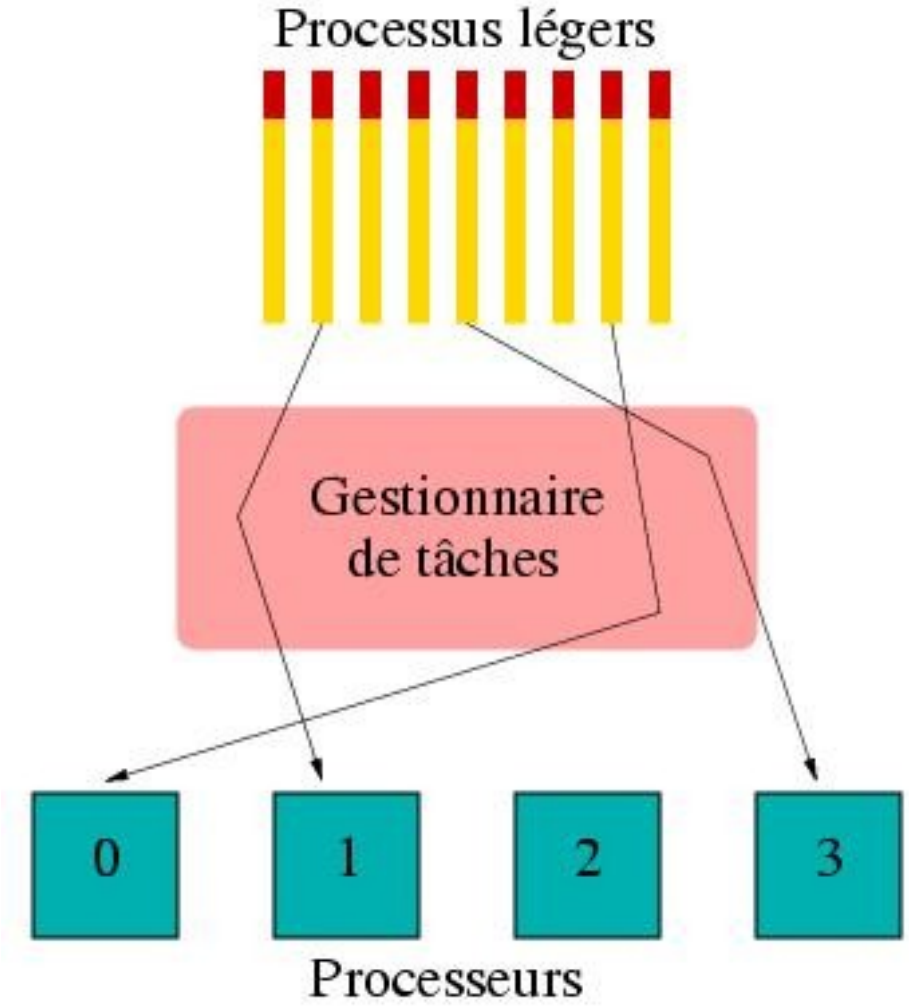
- Il est parfois nécessaire d'introduire une synchronisation entre les tâches concurrentes pour éviter, par exemple, que celles-ci modifient dans un ordre quelconque la valeur d'une même variable partagée (cas des opérations de réduction).



Concepts généraux

- Généralement, les tâches sont affectées 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.

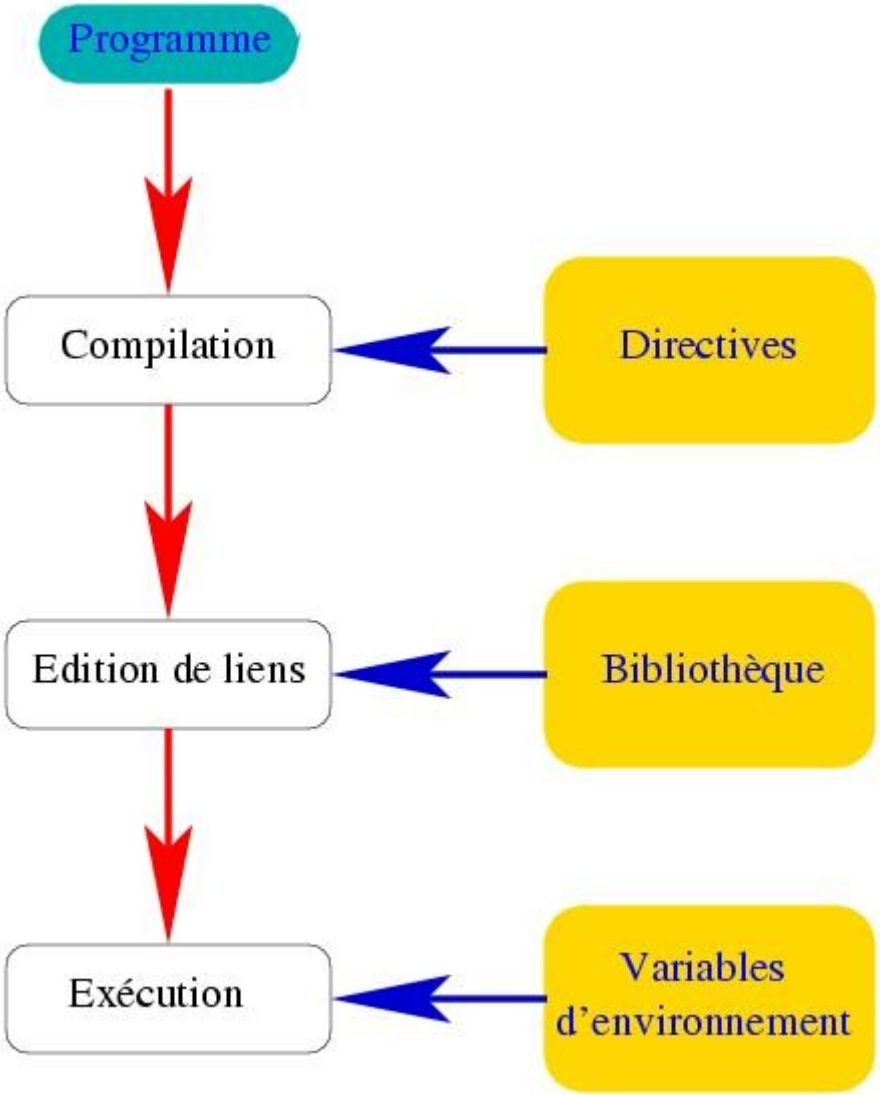
Concepts généraux



Structure d´OpenMP

- Directives et clauses de compilation :
  - elles servent à définir le partage du travail, la synchronisation et le statut privé ou partagé des données ;
  - elles sont considérées par le compilateur comme des lignes de commentaires 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.

Structure d´OpenMP



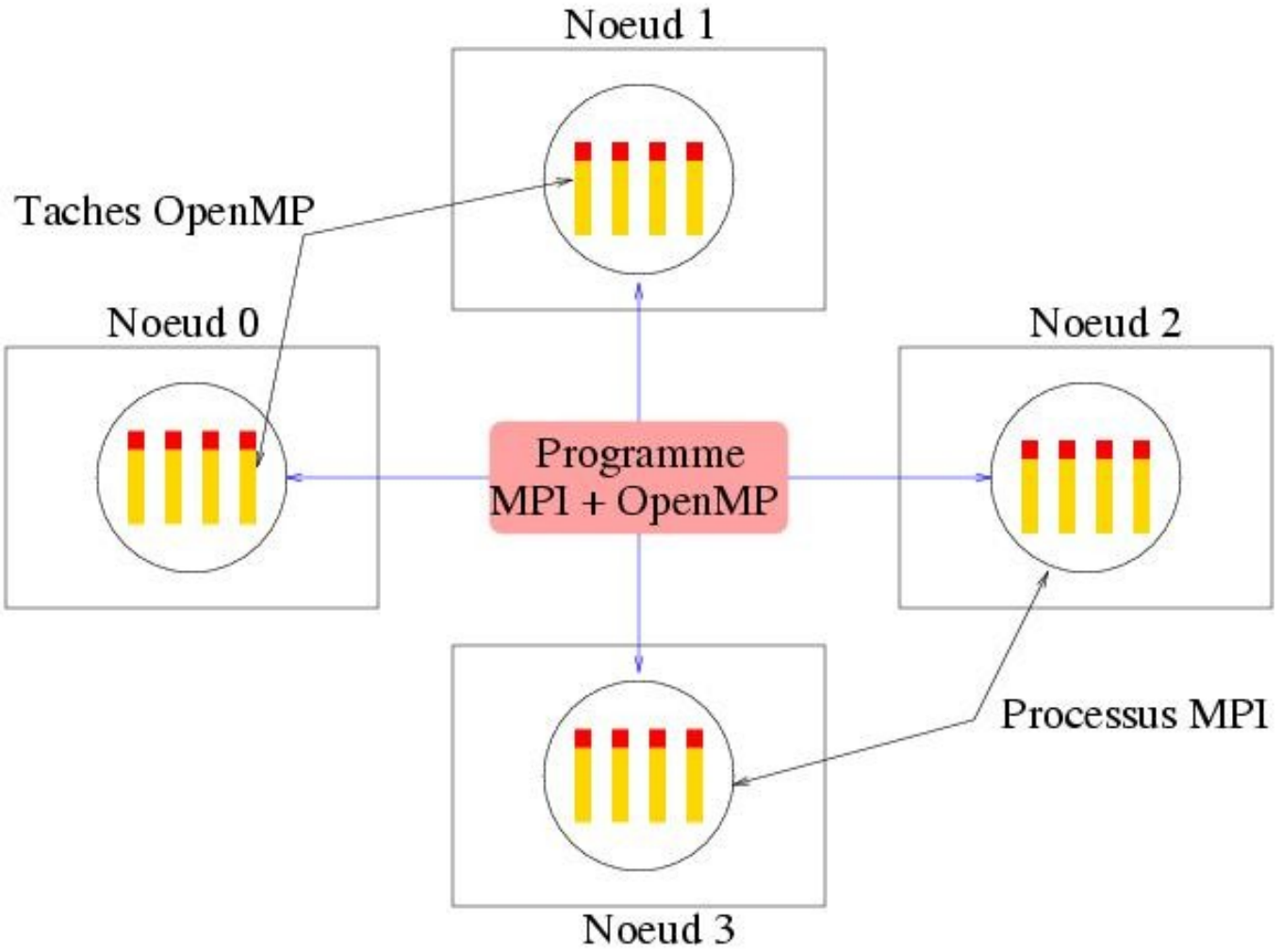
OpenMP versus MPI

- Ce sont deux 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).

OpenMP versus MPI

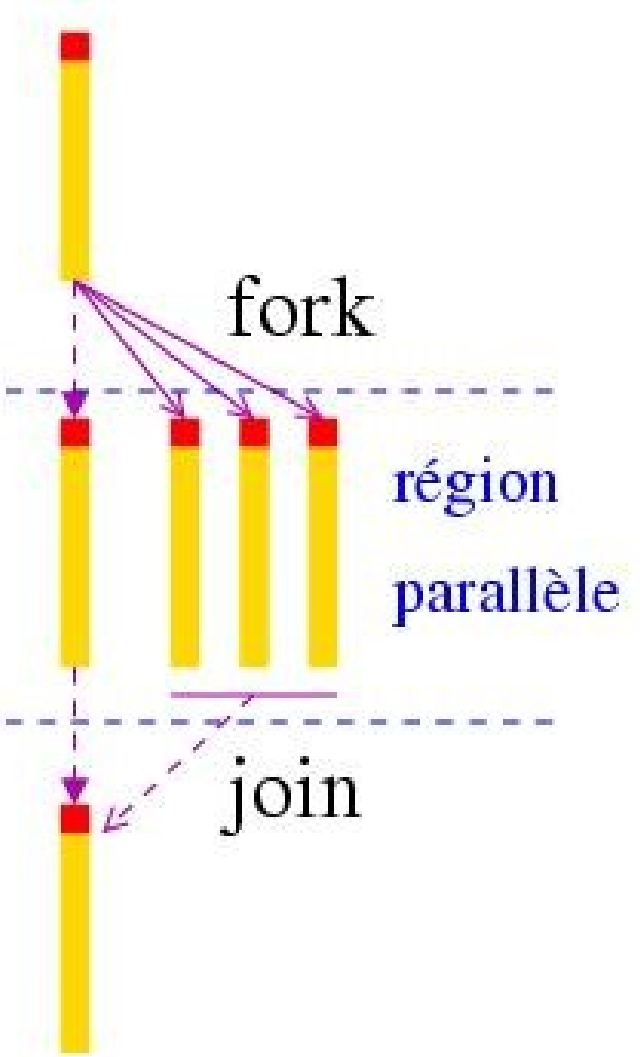
- MPI est utilisé en général sur des machines multiprocesseurs à mémoire distribuée.
- OpenMP est utilisé sur des machines multiprocesseurs à mémoire partagée.
- Sur une grappe de machines indépendantes (noeuds) multiprocesseurs à mémoire partagée, la mise en oeuvre d'une parallélisation à deux niveaux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

OpenMP versus MPI



Principes

- Il est à la charge du développeur d'introduire des directives OpenMP dans son code (du moins en l'absence d'outils de parallélisation automatique).
- À l'exécution du programme, le système d'exploitation construit une région parallèle sur le modèle « fork and 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 d´une directive

- 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.
- 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.

Syntaxe générale d´une directive

```
#include <omp.h>

...

pragma omp parallel private(a,b) \
                    firstprivate(c,d,e)
{
        ...
}
```

Construction d´une région parallèle

- Dans une région parallèle, par défaut, le statut des variables est partagé.
- 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.

Construction d´une région parallèle

```
#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;
}
```

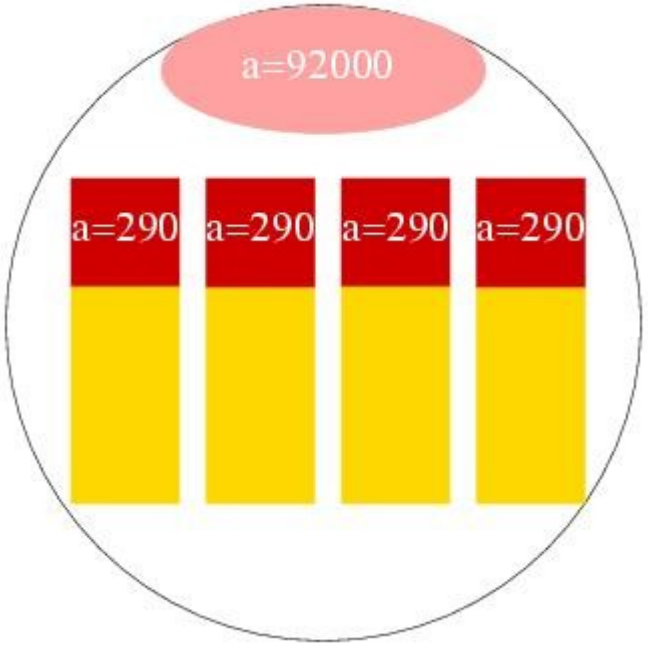
Construction d´une région parallèle

```
> gcc ... -fopenmp prog.c
> export OMP_NUM_THREADS=4
> a.out
```

```
a vaut : 92290. ; p vaut : 1
a vaut : 92290. ; p vaut : 1
a vaut : 92290. ; p vaut : 1
a vaut : 92290. ; p vaut : 1
```

Construction d´une région parallèle

- 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).



Construction d´une région parallèle

```
#include <stdio.h>
#include <omp.h>
int main()
{
      float a;

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

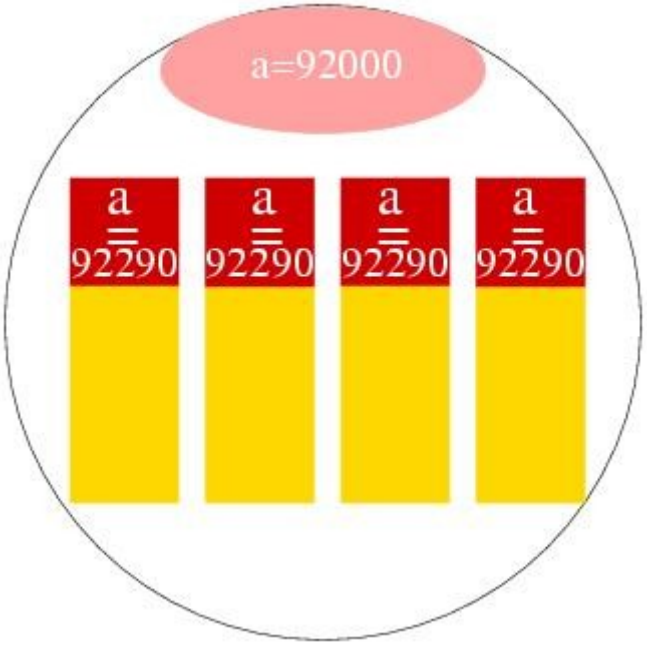
Construction d´une région parallèle

```
> gcc ... -fopenmp prog.c
> export OMP_NUM_THREADS=4
> a.out
```

```
a vaut : 290.
a vaut : 290.
a vaut : 290.
a vaut : 290.
```

Construction d´une région parallèle

- 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.



Construction d´une région parallèle

```
#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;
}
```

Construction d´une région parallèle

```
> gcc ... -fopenmp prog.c
> export OMP_NUM_THREADS=4
> a.out
```

```
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

```
int main()
{
void sub(void);

    #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);
}
```

Étendue d´une région parallèle

```
> gcc ... -fopenmp prog.c sub.c
> export OMP_NUM_THREADS=4
> a.out
```

```
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 (elles sont définies dans la pile de chaque tâche).

```
int main()
{
     void sub(void);

     #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);
}
```

Étendue d´une région parallèle

```
> gcc ... -fopenmp prog.c sub.c
> export OMP_NUM_THREADS=4
> a.out
```

```
a vaut : 92293
a vaut : 92291
a vaut : 92292
a vaut : 92290
```

Cas de la transmission par argument

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

```
#include <stdio.h>
int main()
{
    void sub(int x, int *y);
    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();
}
```

Cas de la transmission par argument

```
> gcc ... -fopenmp prog.c sub.c
> export OMP_NUM_THREADS=4
> a.out
```

```
b vaut : 92003
b vaut : 92001
b vaut : 92002
b vaut : 92000
```

Cas des 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.

Cas des variables statiques

```
void sub(void);
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);
}
```

Cas des variables statiques

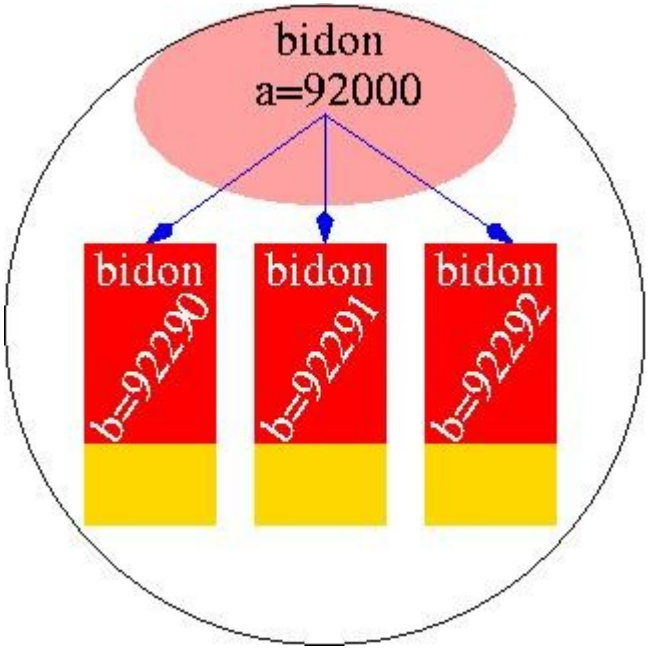
```
> gcc ... -fopenmp prog.c sub.c
> export OMP_NUM_THREADS=2
> a.out
```

B vaut : 92290

B vaut : 92290

Cas des 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.



Cas des variables statiques

```
#include <stdio.h>
#include <omp.h>
void sub(void);
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);
}
```

Cas des variables statiques

```
> gcc ... -fopenmp prog.c sub.c
> export OMP_NUM_THREADS=4
> a.out
```

```
B vaut : 92290
B vaut : 92291
B vaut : 92292
B vaut : 92293
Hors region, A vaut : 92000
```

Cas de l´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 (c'est le cas dans l'exemple ci-contre).

Cas de l´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;
}
```

Cas de l´allocation dynamique

```
> gcc ... -fopenmp prog.c
> export OMP_NUM_THREADS=4
> a.out
```

```
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.
- 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;
}
```

Compléments

```
> gcc ... -fopenmp prog.c
> export OMP_DYNAMIC=true
> export MP_NESTED=true
> a.out

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
```

Partage du travail

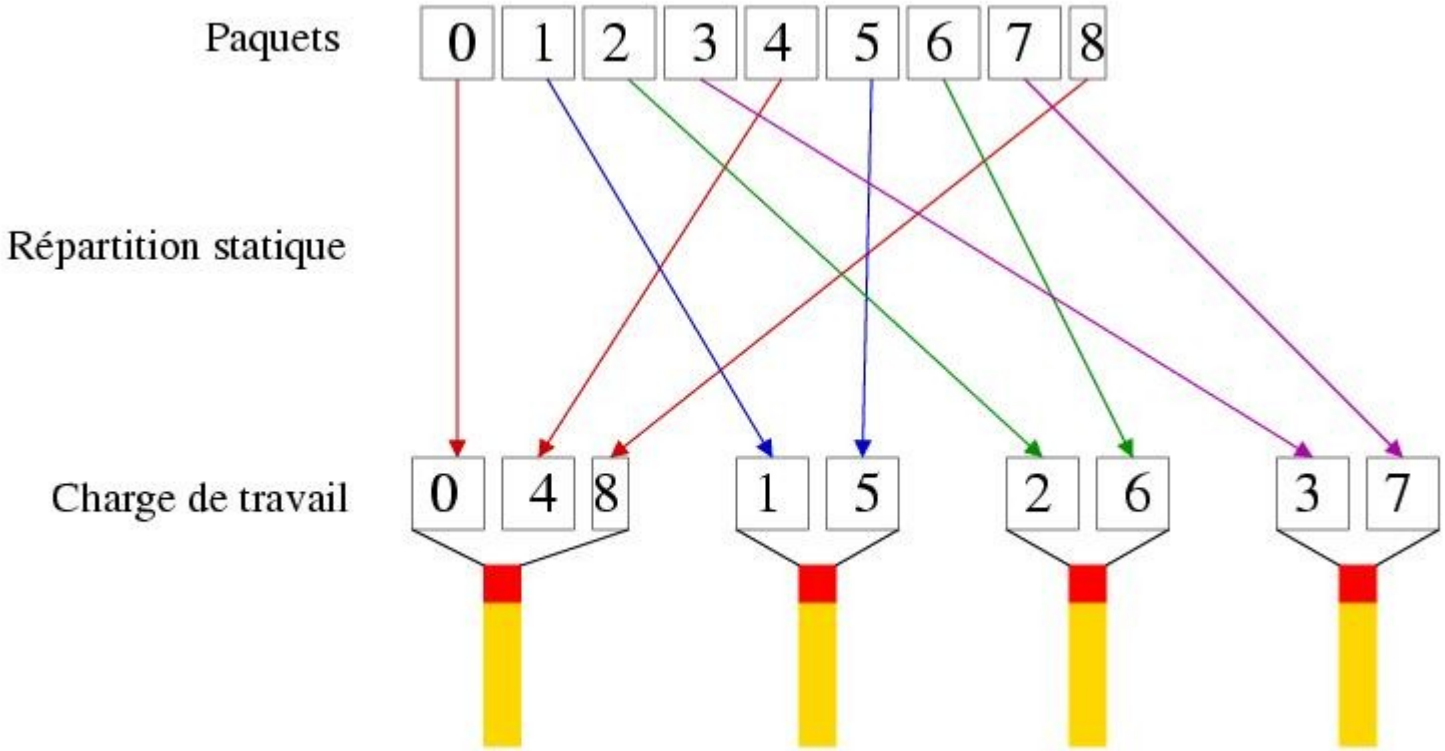
- En principe, la construction d'une région parallèle et l'utilisation de quelques fonctions OpenMP suffisent à eux seuls pour paralléliser une portion de code.
- Mais il est, dans ce cas, à la charge du programmeur de répartir aussi bien le travail que les données et d'assurer la synchronisation des tâches.
- Heureusement, OpenMP propose trois directives (DO, SECTIONS et WORKSHARE) 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.

Boucle parallèle

- C'est un parallélisme par répartition des itérations d'une boucle.
- La boucle parallélisée est celle qui suit immédiatement la directive DO.
- Les boucles « infinies » et do while ne sont pas parallélisables avec OpenMP.
- Le mode de répartition des itérations peut être spécifié dans la clause SCHEDULE.
- Le choix du mode de répartition permet de mieux contrôler l'équilibrage de la charge de travail entre les tâches.
- Les indices de boucles sont des variables entières privées.
- Par défaut, une synchronisation globale est effectuée en fin de construction END DO à moins d'avoir spécifié la clause NOWAIT.

Clause SCHEDULE

- La répartition STATIC consiste à diviser les itérations en paquets d'une taille donnée (sauf peut-être pour le dernier). 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.



Clause SCHEDULE

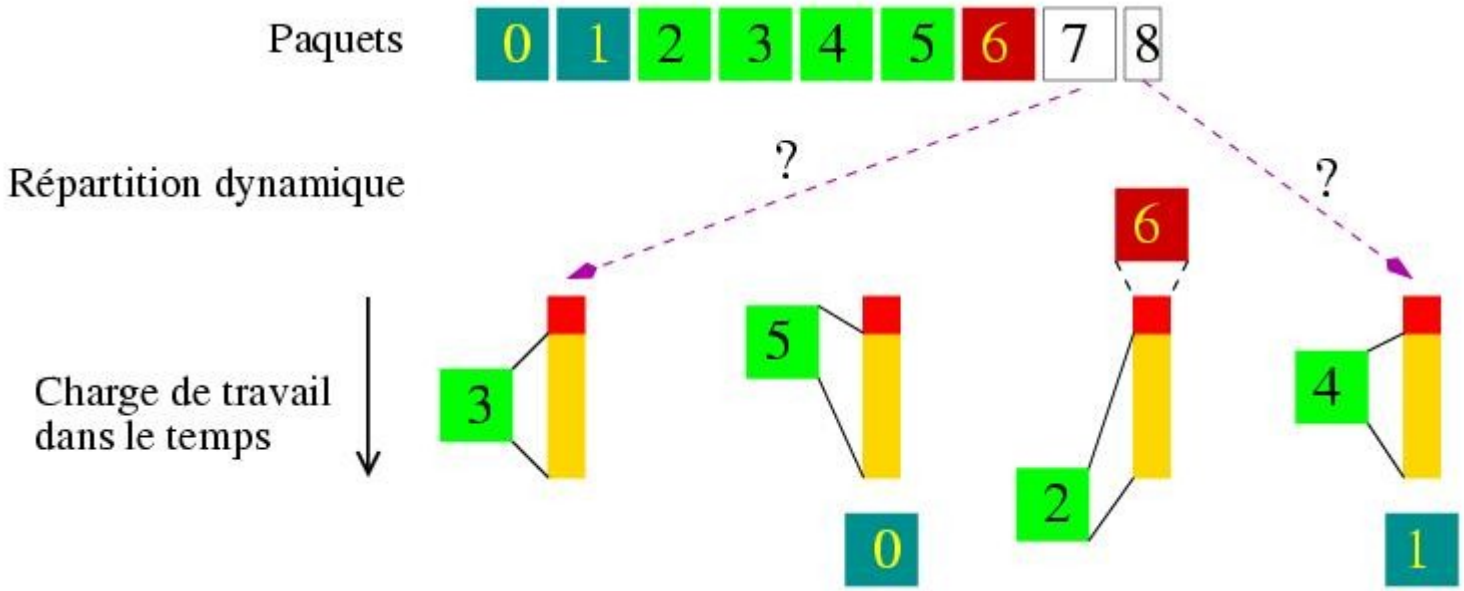
- Nous aurions pu différer à l'exécution le choix du mode de répartition des itérations à l'aide de la variable d'environnement OMP_SCHEDULE.
- Le choix du mode de répartition des itérations d'une boucle peut être un atout majeur pour l'équilibrage de la charge de travail sur une machine dont les processeurs ne sont pas dédiés.
- Attention, pour des raisons de performances vectorielles ou scalaires, éviter de paralléliser les boucles faisant référence à la première dimension d'un tableau multi-dimensionnel.

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 ; a.out

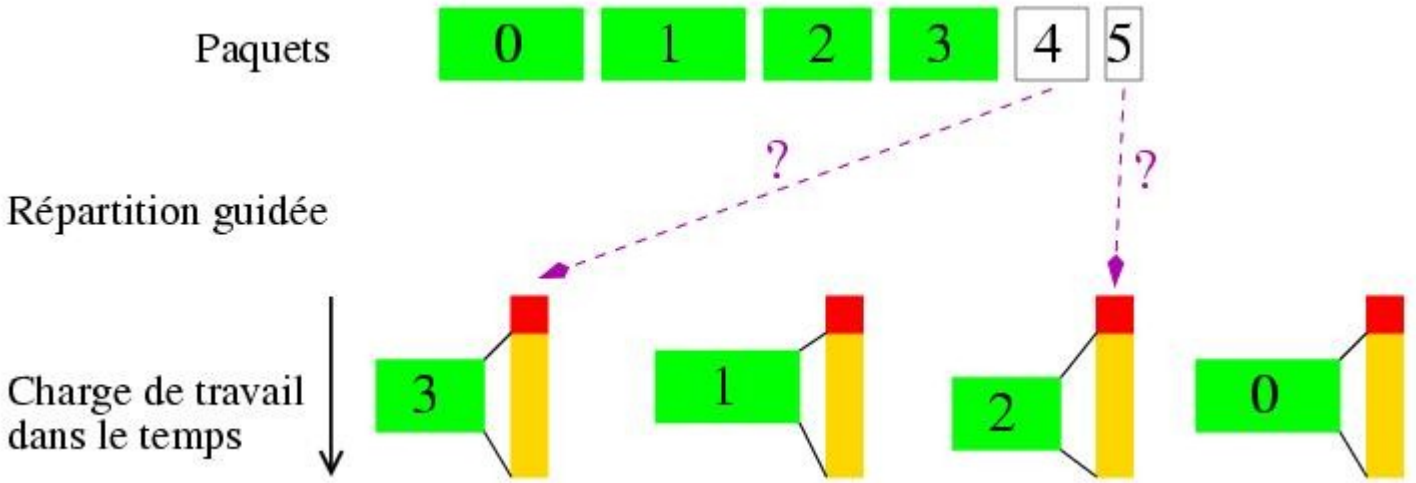


Clause SCHEDULE

- 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 ; a.out



Cas d´une exécution ordonnée

- Il est parfois utile (cas de débogage) d'exécuter une boucle d'une façon ordonnée.
- L'ordre des itérations sera alors identique à celui correspondant à une exécution séquentielle.

Cas d´une 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;
}
```

Cas d´une exécution ordonnée

- 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 indépendemment des autres. Elles se synchronisent ensuite pour mettre à jour le résultat final.

Cas d´une exécution ordonnée

```
#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;
}
```

Compléments

- Les autres clauses admises dans la directive DO sont :
  - 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.

Compléments

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

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 END SECTIONS pour lever la barrière de synchronisation implicite.

Construction SECTIONS

```
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;
}
```

Compléments

- 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

- 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 SINGLE

- La construction SINGLE permet de faire exécuter une portion de code par une et une seule tâche sans pouvoir indiquer laquelle.
- 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 END SINGLE, la terminaison de celle qui en a la charge, à moins d'avoir spécifié la clause NOWAIT.

Construction SINGLE

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

int main()
{
    int rang;
    float a;

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

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

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

Construction SINGLE

- Une clause supplémentaire admise uniquement par la directive de terminaison END SINGLE 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.

Construction SINGLE

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

int main()
{
    int rang;
    float a;

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

        #pragma omp single copyprivate(a)
        {
            a = -92290.;
        }

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

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;
}
```

Procédures orphelines

- Une procédure (fonction ou sous-programme) appelée dans une région parallèle est exécutée par toutes les tâches.
- En général, il n'y a aucun intérêt à cela si le travail de la procédure n'est pas distribué.
- Cela nécessite l'introduction de directives OpenMP (DO, SECTIONS, etc.) dans le corps de la procédure si celle-ci est appelée dans une région parallèle.
- Ces directives sont dites « orphelines » et, par abus de langage, on parle alors de procédures orphelines (orphaning).
- Une bibliothèque scientifique multi-tâches avec OpenMP sera constituée d'un ensemble de procédures orphelines.

Procédures orphelines

- Attention, car il existe trois contextes d'exécution selon le mode de compilation des unités de programme appelantes et appelées :

1. la directive PARALLEL de l'unité appelante est interprétée (l'exécution peut être Parallèle) à la compilation ainsi que les directives de l'unité appelée (le travail peut être Distribué) ;

2. la directive PARALLEL de l'unité appelante est interprétée à la compilation (l'exécution peut être Parallèle) mais pas les directives contenues dans l'unité appelée (le travail peut être Répliqué) ;

3. la directive PARALLEL de l'unité appelante n'est pas interprétée à la compilation. L'exécution est partout Séquentielle même si les directives contenues dans l'unité appelée ont été interprétées à la compilation.

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).

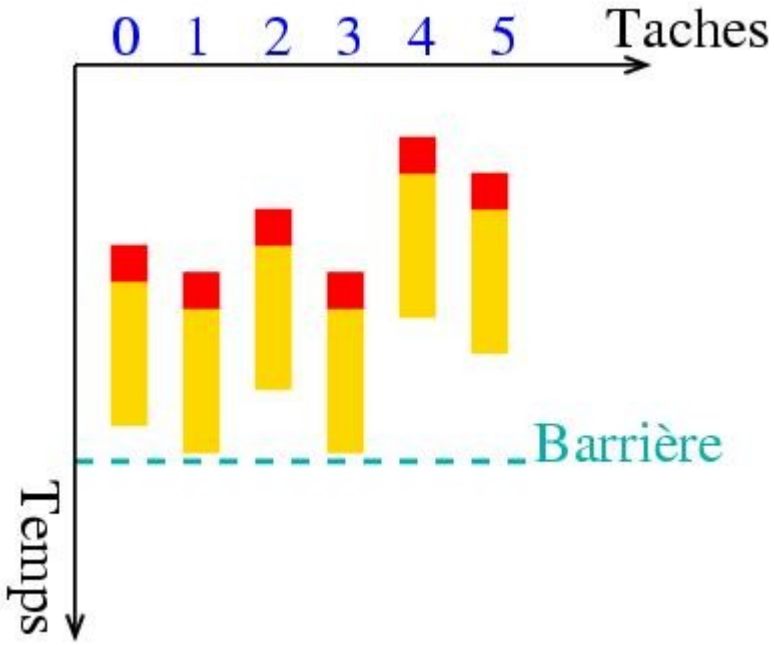
3. pour synchroniser au moins deux tâches concurrentes parmi l'ensemble (mécanisme de verrous).

Synchronisations

- Comme nous l'avons déja indiqué, l'absence de clause NOWAIT signifie qu'une barrière globale de synchronisation est implicitement appliquée en fin de construction \openmp. Mais 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

- La directive BARRIER synchronise l'ensemble des tâches concurrentes dans une région parallèle.
- 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.
- Son effet est local à l'instruction qui suit immédiatement la directive.

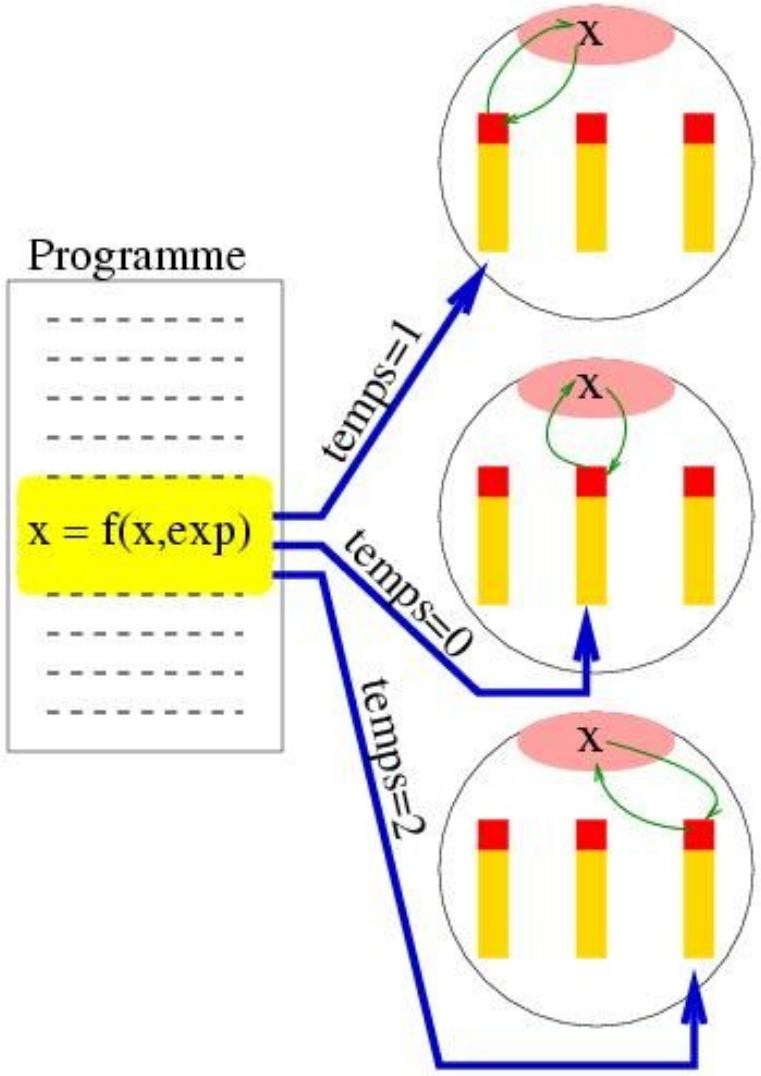
Mise à jour atomique

```
#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;
}
```

Synchronisations

- 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..
- if représente une des fonctions intrinsèques suivantes : MAX, MIN, IAND, IOR, IEOR.
- exp est une expression arithmétique quelconque indépendante de x.

Synchronisations



Régions critiques

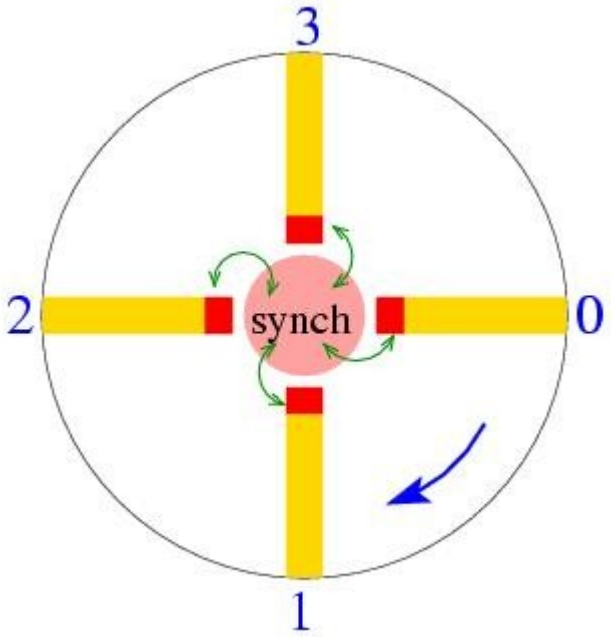
- Une région critique peut être vue comme une généralisation de la directive ATOMIC bien que les mécanismes sous-jacents soient distincts.
- Les tâches exécutent cette région dans un ordre non-déterministe mais une à la fois.
- Une région critique est définie grâce à la directive CRITICAL et s'applique à une portion de code terminée par END CRITICAL.
- Son étendue est dynamique.
- 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.



Quelques pièges

- Dans le première exemple ci-contre, le statut de la variable « s » est erroné ce qui produit un résultat indéterminé. En effet, le statut de « s » doit être SHARED dans l'étendue lexicale de la région parallèle si la clause LASTPRIVATE est spécifiée dans la directive DO (ce n'est pas la seule clause dans ce cas là). Ici, les deux implémentations, IBM et NEC, fournissent deux résultats différents. Pourtant, ni l'une ni l'autre n'est en contradiction avec la norme alors qu'un seul des résultats est correct.

Quelques pièges

```
#include <stdio.h>
#define N 9

int main()
{
 int i;
 float s, a[N];

 for(i=0; i<N; i++)
     a[i] = 92290.;

 #pragma omp parallel private(s,i) shared(a)
 {
        #pragma omp for lastprivate(s)
        for(i=0; i<N; i++)
            s = a[i];

        printf("s = %f ; a[8] = %f\n",s,a[N-1]);
    }
    return 0;
}
```

Quelques pièges

- Dans le second exemple ci-contre, il se produit un effet de course entre les tâches qui fait que l'instruction « print » n'imprime pas le résultat escompté de la variable « s » dont le statut est SHARED. Il se trouve ici que NEC et IBM fournissent des résultats identiques mais il est possible et légitime d'obtenir un résultat différent sur d'autres plateformes. La solution est de glisser, par exemple, une directive BARRIER juste après l'instruction « print ».

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;
}
```

Performances

- En général, les performances dépendent de l'architecture (processeurs, liens d'interconnexion et mémoire) de la machine et de l'implémentation OpenMP utilisée.
- Il existe, néanmoins, quelques règles de « bonnes performances » indépendantes de l'architecture.
- En phase d'optimisation avec OpenMP, l'objectif sera de réduire le temps de restitution du code et d'estimer son accélération par rapport à une exécution séquentielle.

Règles de bonnes performances

- Minimiser le nombre de régions parallèles dans le code.
- Adapter le nombre de tâches demandé à la taille du problème à traiter afin de minimiser les surcoûts de gestion des tâches par le système.
- Dans la mesure du possible, paralléliser la boucle la plus externe.
- Utiliser la clause SCHEDULE(RUNTIME) pour pouvoir changer dynamiquement l'ordonnancement et la taille des paquets d'itérations dans une boucle.
- La directive SINGLE et la clause NOWAIT peuvent permettre de baisser le temps de restitution au prix, le plus souvent, d'une synchronisation explicite.
- La directive ATOMIC et la clause REDUCTION sont plus restrictives mais plus performantes que la directive CRITICAL.

Règles de bonnes performances

- Utiliser la clause IF pour mettre en place une parallélisation conditionnelle (ex. sur une architecture vectorielle, ne paralléliser une boucle que si sa longueur est suffisament grande).
- Éviter de paralléliser la boucle faisant référence à la première dimension des tableaux (en Fortran) car c'est celle qui fait référence à des éléments contigus en mémoire.

Règles de bonnes performances

```
#include <stdio.h>
#include <stdlib.h>
#define N 1025

int main()
{
 int i, j;
 float a[N][N], b[N][N];

 for(j=0; j<N; j++)
     for(i=0; i<N; i++)
         a[i][j] = (float) drand48();

 #pragma omp parallel do schedule(runtime) if(N > 514)
 {
        for(j=1; j<N-1; j++)
            for(i=0; i<N; i++)
                b[i][j] = a[i][j+1] - a[i][j-1];
 }
 return 0;
}
```

Règles de bonnes performances

- Les conflits inter-tâches (de banc mémoire sur une machine vectorielle ou de défauts de cache sur une machine scalaire), peuvent dégrader sensiblement les performances.
- Sur une machine multi-noeuds à mémoire virtuellement partagée (ex. SGI-O2000), les accès mémoire (type NUMA : Non Uniform Memory Access) sont moins rapides que des accès intra-noeuds.
- Indépendamment de l'architecture des machines, la qualité de l'implémentation OpenMP peut affecter assez sensiblement l'extensibilité des boucles parallèles.

Mesures du temps

- OpenMP offre deux fonctions :

  1. OMP_GET_WTIME pour mesurer le temps de restitution en secondes ;

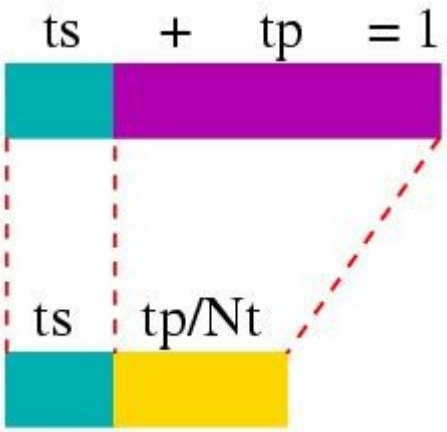
  2. OMP_GET_WTICK pour connaître la précision des mesures en secondes.

- Ce que l'on mesure est le temps écoulé depuis un point de référence arbitraire du code.
- Cette mesure peut varier d'une exécution à l'autre selon la charge de la machine et la répartition des tâches sur les processeurs.

Accélération

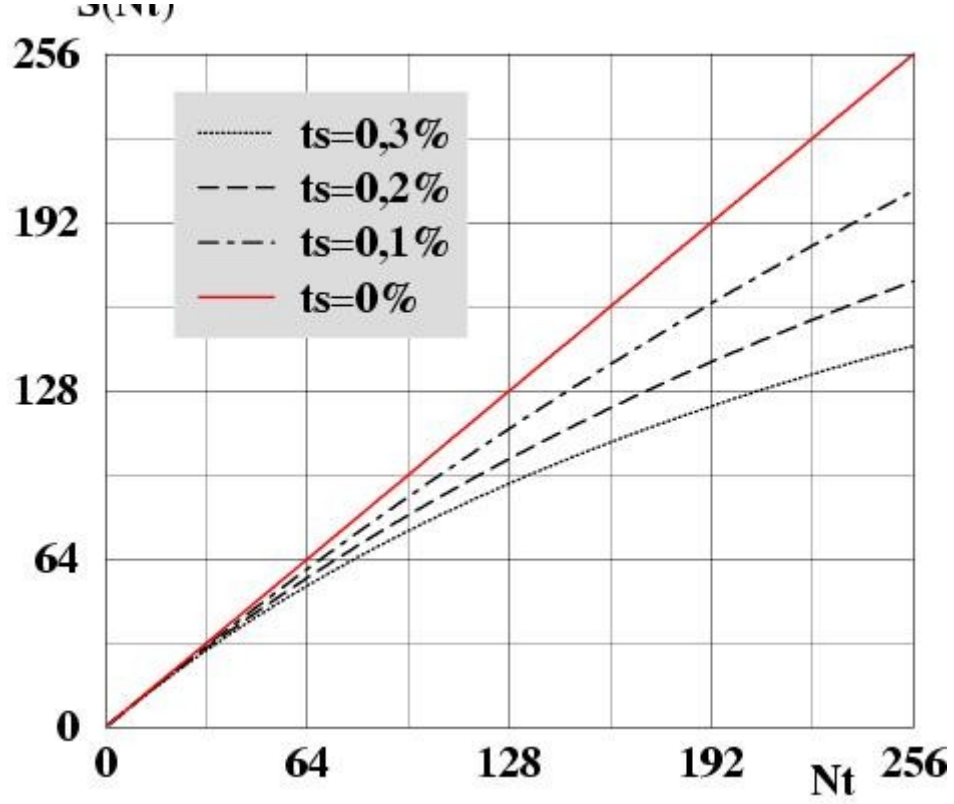
- Le gain en performance d'un code parallèle est estimé par rapport à une exécution séquentielle.
- Le rapport entre le temps séquentiel Ts et le temps parallèle Tp sur une machine dédiée est déjà un bon indicateur sur le gain en performance. Celui-ci définit l'accélération S(Nt) du code qui dépend du nombre de tâches Nt.
- Si l'on considère Ts=ts+tp=1 (ts représente le temps relatif à la partie séquentielle et tp celui relatif à la partie parallélisable du code), la loi dite de « Amdhal » S(Nt)=1/(ts+(tp/Nt)) indique que l'accélération S(Nt) est majorée par la fraction séquentielle 1/ts du programme.

Accélération



$$S(Nt) = \frac{1}{ts + \frac{tp}{Nt}}$$

Accélération



Conclusion

- Nécessite une machine multi-processeurs à mémoire partagée.
- Mise en oeuvre relativement facile, même dans un programme à l'origine séquentiel.
- Permet la parallélisation progressive d'un programme séquentiel.
- Tout le potentiel des performances parallèles se trouve dans les régions parallèles.
- Au sein de ces régions parallèles, le travail peut être partagé grâce aux boucles et aux sections parallèles. Mais on peut aussi singulariser une tâche pour un travail particulier.
- Les directives orphelines permettent de développer des procédures parallèles.

Conclusion

- Des synchronisations explicites globales ou point à point sont parfois nécessaires dans les régions parallèles.
- Un soin tout particulier doit être apporté à la définition du statut des variables utilisées dans une construction.
- L'accélération mesure l'extensibilité d'un code. Elle est majorée par la fraction séquentielle du programme et est ralentie par les surcoûts liés à la gestion des tâches.