EISTI
Ecole Internationale des Sciences du Traitement de l'Information

Algorithme et programmation parallèle Open MP

clauses en bleu

directives en rouge

Extraits tirés du cours de *Blaise Barney*

Bibliographie

- Le site officiel du standard OpenMP (en Anglais)

http://www.openmp.org

Le site microsoft France (en Français)

http://msdn.microsoft.com/fr-fr/library/tt15eb9t.aspx

- Les cours de l'IDRIS (en Français)

http://www.idris.fr/data/cours/parallel/openmp/

Avant Propos

- OpenMP est une interface standard de haut niveau pour une programmation parallèle de type SPMD(Single Program Multiple Data) sur machine à mémoire partagée ou au moins virtuellement partagée(SGI O200).
- Basée sur les techniques du *multithreading, on* peut considérer OpenMP comme l'un des grands standards au service du calcul scientifique.

Mémoire partagée avec OpenMP

- Introduction
  - Introduction
  - Architecture
  - Directive et clause
  - Parallélisme et multithread
- Bibliothèque Runtime
- Variables d’environnement
- Variables partagées et privées
- Directives
- Clauses

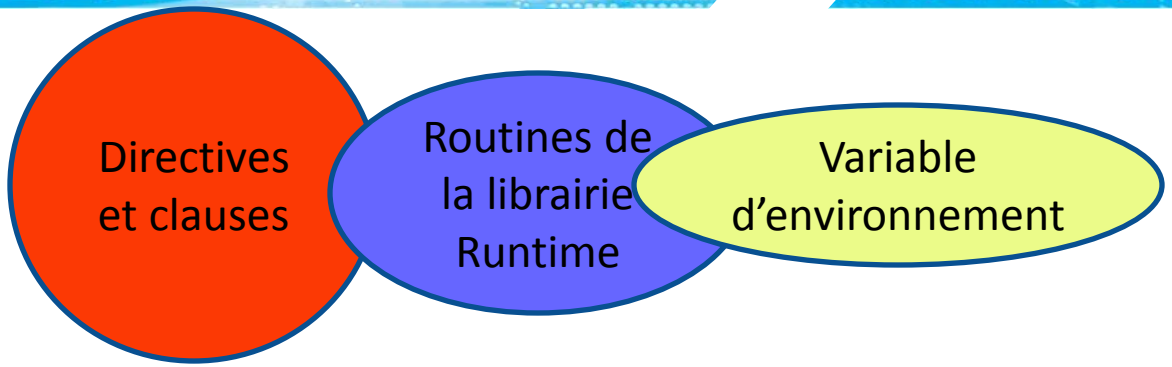
Introduction

- **Il s'agit d'un ensemble identifié de procédures et de directives de compilation par un mot clef initial #pragma visant à réduire le temps de restitution lors de l'exécution d'un programme sans en changer la sémantique.**
  - Standard "**industriel" basé sur une API pour**
    - la **portabilité d'applications *multithreadées,***
    - sur machines à mémoire **partagée ou à mémoire virtuellement partagée.**
  - Standardisation de la parallélisation
    - à grain fin (sur des boucles),
    - à gros grain (exemple : décomposition de domaines).

Terminologie

- Il existe un vocabulaire de base propre à OpenMP
  - *compliant : conforme,*
  - *team : équipe (sous-entendu de tâches),*
  - *construct : région, zone ou construction,*
  - *extent : littéralement extension ou terme de* portée,
  - *statements : instructions,*
  - *binding : relation de parenté,*
  - *nesting : nidification (imbrication),*
  - *orphaning : orphelinat,*
  - *scheduling : ordonnancement,*
  - *chunk : paquet ou taille du paquet*

Architectures d'OpenMP



- Les directives et leurs clauses sont de 3 types :
  - ✓partage du travail (***work-sharing),***
  - ✓partage des données,
  - ✓synchronisation.

- La Runtime Library et les variables d'environnement permettent de contrôler l'environnement d'exécution comme par exemple le nombre de *threads*

Les objets d'OpenMP

- clauses : firstprivate, lastprivate, copyprivate, copyin,...
- directives : threadprivate, ...
- fonctions (Runtime): omp_set_num_threads, omp_get_wtime, omp_get_num_threads, omp_get_thread_num, ...
- variables d'environnement : OMP_NUM_THREADS, OMP_SCHEDULE, OMP_DYNAMIC, ...
- Un fichier d'entête pour le C/C++ omp.h, permet de définir les prototypes des fonctions OpenMP

Parallélisme et multithread

- La parallélisation se fait en insérant des directives dans le code séquentiel pour construire des régions parallèles

Parallélisme et multithread

- Un programme OpenMP est exécuté par un processus lourd unique.
- Ce processus lourd active des processus légers (threads) à l'entrée d'une région parallèle.
- Chaque thread accède à toutes les ressources (mémoires, descripteurs de fichiers, ...) du processus.
- Chaque thread exécute une tâche composée d'un ensemble d'instructions différentes grâce à sa propre pile(stack).
- Un programme OpenMP est une alternance de régions séquentielles et de régions parallèles.
- Les threads sont identifiés par leur rang, et celui du thread maitre est 0
- Une région séquentielle est toujours exécutée par le thread maître

Parallélisme et multithread

- OpenMP correspond plutôt à un parallélisme de contrôle qu'à un parallélisme de données. Pendant l'exécution d'une tâche par un thread, 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 tous les processus légers ; on parle alors de variable partagée

Parallélisme et multithread

OpenMP est un ensemble de constructions parallèles, basé sur des directives de compilation pour architecture à mémoire partagée.

- Il est basé sur le principe du fork and join
- Le programme est séquentiel
- Un seul thread(tâche) maître s'exécute jusqu'à la 1ère région parallèle
- Ce thread crée une équipe de threads dont il est maître
- Son effectif est contrôlé par variable d'environnement ou appel librairie
- Après la région parallèle toutes les tâches se synchronisent
- Seul le thread maître poursuit son exécution et l'exécution redevient séquentielle

Les directives OPENMP

#pragma omp directive [clause [clause] . . .]

- Formées de quatres parties :

  1 La sentinelle : #pragma omp

  2 Un nom de directive valide

  3 Une liste de clauses optionnelles (infos supplémentaires)

  4 Un retour à la ligne

- Règles générales :
  - Attention à respecter la casse
  - Une directive s'applique sur le bloc de code suivant
  - Les directives longues peuvent se poursuivre à la ligne en
  - utilisant le caractère antislash « \ » en fin de ligne

Directives

- Les directives ne sont que des commentaires activables par le compilateur grâce une option adéquate.
- Un compilateur ne supportant pas OpenMP les ignorera.
- Permet de garantir
  - **la portabilité des codes OpenMP,**
  - la maintenance d'une version **unique, séquentielle et parallèle,** du code
- exemple

Compilation conditionnelle par sentinelle ou par macro

Une ligne de code peut être compilée conditionnellement de 3 façons :

- grâce à la parallélisation conditionnelle avec la clause if**,**
- si elle est précédée d'une sentinelle de compilation conditionnelle
  - les sentinelles possibles sont : **!$, C$, *$;**

```
C$ Boolean=omp_in_parallel();
```

- grâce à la macro de *préprocessing* ***_OPENMP prédéfinie,***

```
#ifdef _OPENMP
        #pragma omp  parallel  default(shared) private(beta, pi)
#endif
```

Directive : région parallèle conditionnelle

- **i f** (/ Expression scalaire /)
- Si elle est présente, elle doit évaluer la valeur non-zéro (C / C + +) pour créer une équipe de threads, sinon, la région est exécutée séquentiellement par le thread maître.

```
#include < stdio.h>
#define PARALLEL 1 / / 0 pour sé quent iel , != 0 pour parallèle
int main ( ) {
  #pragma omp parallel if (PARALLEL)
  p r i n t f ( " Hello , openMP\ n " ) ;
  return 0;
}
```

Directives

- Format des directives

sentinelle nom-directive [clause,...] newline

#pragma omp parallel default(shared) private(beta, pi)

int nbth = omp_get_num_threads();

Où :

#pragma omp	obligatoire comme préfixe des directives.
parallel	est une directive.
[clause,...]	sont les clauses, optionnelles, qui permettent de décrire la visibilité des variables

Directives

- Règles générales :
  - sensible à la casse
  - Directives suivent les conventions de C / C + + standard pour les directives de compilation
  - Seul un nom de directive peut être spécifié par la directive
  - Chaque directive s'applique au plus à l'instruction suivante, qui doit être un bloc structuré.
  - Les directives longues peuvent être "poursuivies" sur les lignes suivantes en échappant par le caractère de nouvelle ligne avec une barre oblique inverse ("\") à la fin d'une ligne directive

Les routines de la bibliothèque Runtime

- gestion de verrous
- gestion de l'environnement d'exécution
  - modification/vérification du nombre de processus légers
  - test de région parallèle
  - nombre de processeurs dans le système
  - dynamicité du nombre de processus légers
  - création de nouveaux processus légers lors de l'imbrication de régions parallèles
- priorité par rapport aux variables d'environnement mais pas par rapport aux directives

Les routines de la bibliothèque Runtime

- L'API OpenMP inclut un nombre sans cesse croissant de routines de la bibliothèque Runtime.
- Ces routines sont utilisées pour des fins très variées:
  - getter et setter du nombre de threads
  - getter de l'identifiant du thread, de celui de son ancêtre, de la taille du pool de threads
  - getter et setter des caractéristiques du thread dynamique
  - demander si on est une région parallèle, et à quel niveau d'imbrication
  - setter et setter du parallélisme imbriqué
  - initialisation et de terminaison de verrous et verrous imbriqués
  - getter du temps et de la durée de résolution

Les routines de la bibliothèque Runtime

- Pour le C / C + +, toutes les routines de la bibliothèque Runtime sont des fonctions ou des procédures.

```
#include <omp.h>

//demander le nombre de threads de l'équipe
int omp_get_num_threads(void)

//demander l 'id du thread courant, le thread maitre porte le numéro 0
int omp_get_thread_num (void)

//retourne le nombre maximal de threads qui vont exécuter la prochaine région parallèle
int omp_get_max_threads()
```

Variables d'environnement

- Contrôle de l'ordonnancement « runtime » : OMP_SCHEDULE
- Nombre de processus légers à utiliser : OMP_NUM_THREADS
- Dynamicité du nombre de processus légers : OMP_DYNAMIC
- Contrôler la taille de la pile d'exécution des threads OMP_STACKSIZE
- Création de nouveaux threads lors de l'imbrication de régions parallèles : OMP_NESTED
- Spécifiant comment les itérations des boucles ont divisés
- Liaison des threads au processeur
- Activation / désactivation de parallélisme imbriqué; fixation des niveaux maximum d'imbrication
- Réglage de la taille de pile de thread

Variable d’environnement

- Définition des variables d'environnement OpenMP se fait de la même manière lorsque vous définissez n’import quelle autre variable d'environnement, et dépend du shell que vous utilisez.

- **OMP_NUM_THREADS**
  - **setenv OMP_NUM_THREADS 8** **!---** csh/tcsh
  - **export OMP_NUM_THREADS=8** **!---** sh/bash

- Si le mode dynamique est désactivé, elle fixe le nombre de tâches de toutes les régions parallèles.
- Si le mode dynamique est activé, elle fixe le nombre maximum de tâches que peut comporter l’équipe de chaque région parallèle.

Variable d'environnement

- **OMP_SCHEDULE**
  - export **OMP_SCHEDULE="GUIDED,256"** **!--- En ksh**
  - export **OMP_SCHEDULE="dynamic"** **!--- En ksh**
  - export **OMP_SCHEDULE="static "** **!--- En ksh**
- **OMP_DYNAMIC**
  - export **OMP_DYNAMIC=FALSE** **!--- En ksh**
  - setenv **OMP_DYNAMIC FALSE** **!--- En csh**
- **OMP_NESTED**
  - export **OMP_NESTED=TRUE** **!--- En ksh**
  - setenv **OMP_NESTED TRUE** **!--- En csh**

La Run-time Library d'OpenMP

- Les procédures ou fonctions de la *Run-time Library* ayant un équivalent parmi les variables d'environnement sont prioritaires localement.
  - Les **omp_set_*** sont des procédures à appeler en général dans les régions séquentielles.
  - Les **omp_get_*** sont des fonctions à appeler en général dans les régions parallèles et renvoient des int ou booléen .

Exemple de la structure d'un code source OpenMP

```
#include <omp.h>
main () {
int var1, var2, var3;
//code serie.
//Début d'une section parallèle. Création du pool de threads.
//Spécifiez la portée des variables
#pragma omp parallel private(var1, var2) shared(var3)
{
      Section parallèle exécutée par tous les threads.
     Autre directive  OpenMP.
     Appel la librairie Run-time.
     Tous les threads rejoignent le thread principal pour qu'ils soient démantelés
}
//Reprise du code série.
}
```

Exemple de la région parallèle

```
double V[10000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int TID = omp_thread_num();
    printf(" mon TID = %d ", TID);
}
Printf("fin\n");

// ici les 4 thread exécutent le même code, une seule copie du
// vecteur V est partagée entre tous les threads
```

exemple

```
int main(){
#pragma omp parallel
// bloc 1
#pragma omp parallel
    omp_num_threads(2)
// bloc 2
#pragma omp parallel if(0)
// bloc 3
}
```

- Bloc 1 sera exécuté par tous les threads
  - On verra comment ce nombre par défaut est défini
- Bloc 2 sera exécuté par une équipe de 2 threads
- Uniquement le thread principal va exécuter le Bloc 3
  - 0 = faux!
  - Si on aurait mis if (1) ça serait équivalent au premier pragma

Directive: parallel

- Une région parallèle est délimitée par les directives suivantes :

  #pragma omp parallel { région parallèle }

- Les directives de début et de fin de région parallèle doivent figurer dans le même sous-programme.

- Cela signifie qu'il faut attendre que tous les threads doivent avoir exécuté leurs instructions pour pouvoir sortir de la région parallèle. Dès lors que tous les threads ont exécuté leurs tâches, les processus légers fils meurent et le processus maître exécute l'instruction qui suit directement la fin de région parallèle.

Directive: parallel (suite)

- Une région parallèle est un bloc de code qui sera exécuté par plusieurs threads. C'est la construction fondamentale parallèle d'OpenMP.
- Au début de cette région parallèle, le code est dupliqué et tous les thread exécutent le même code
- Il existe une barrière implicite à la fin d'une section parallèle. Seul le thread principal continue l'exécution après ce point.
- La construction et la destruction d'une région parallèle ayant un coût, il est intéressant de construire une région parallèle sous certaines conditions, exemple si le nombre d'itérations est suffisamment important

Directive : parallel (suite)

- Le nombre de threads dans une région parallèle est déterminé par les facteurs suivants, par ordre de préséance:
  - Réglage de la clause num_threads(n)
  - Fonction de bibliothèque Utiliser des omp_set_num_threads (n)
  - Réglage de la variable d'environnement OMP_NUM_THREADS
  - Défaut de mise en œuvre - le plus souvent le nombre de CPU sur un nœud, mais il pourrait être dynamique (voir point suivant).
- Les threads sont numérotés de 0 (thread maître) à N-1
- L'ordonnancement des threads peut être statique ou dynamique

Directive : parallel (suite)

- Thread dynamique
  - Utilisez la routine omp_get_dynamic () de la bibliothèque afin de déterminer si les threads dynamiques sont activés.
  - Si elle est supportée, les deux méthodes disponibles pour permettre les discussions dynamiques sont les suivants:
    - La routine de bibliothèque omp_set_dynamic ()
      - void omp_set_dynamic(int val);
    - Réglage de la variable d'environnement OMP_DYNAMIC à TRUE

Directive master

- La directive MASTER spécifie une région qui doit être exécutée uniquement par le thread maître de l'équipe. Tous les autres threads de l'équipe ignorent cette section de code

=> Il n'y a pas de barrière implicite associée à cette directive

```
#pragma omp master
{
    bloc structuré
}
```

Directive single

- La directive *single* précise que le code doit être exécuté par un seul thread dans l'équipe, pas nécessairement le thread principal.
- Threads dans l'équipe qui n'exécutent pas la directive *single*, attendent à la fin du bloc de code, à moins qu'une clause nowait est spécifiée.

=> Il y a une barrière implicite associée à cette directive

Exemple : omp_set_dynamic.c

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

int main()
{
   omp_set_dynamic(9);
   omp_set_num_threads(4);
   printf("%d\n", omp_get_dynamic( ));
   #pragma omp parallel
      #pragma omp master
      {
         printf("%d\n", omp_get_dynamic( ));
      }
}
```

Le nombre de threads ne dépasse jamais la valeur définie par omp_set_num_threads ou par OMP_NUM_THREADS.

```
gcc -fopenmp omp_set_dynamic.c -o omp_set_dynamic
./omp_set_dynamic
```

```
1
1
```

Directive : parallel (suite)

- Régions parallèles imbriquées
  - Utilisez la fonction de bibliothèque omp_get_nested() pour déterminer si le parallélisme imbriqué est activé.
  - Les deux méthodes disponibles pour permettre aux régions parallèles imbriquées (si supporté) sont les suivants:
    - La routine de bibliothèque omp_set_nested ()
    - Réglage de la variable d'environnement OMP_NESTED à TRUE
  - Si ce n'est pas pris en charge, une région parallèle imbriquée dans une autre région résulte en parallèle à la création d'une nouvelle équipe, composée d'un thread, par défaut (traitement séquentiel).

Exemple : omp_set_nested.c

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

int main( )
{
   omp_set_nested(1);
   omp_set_num_threads(4);
   printf("%d\n", omp_get_nested( ));
   #pragma omp parallel
      #pragma omp master
      {
         printf("%d\n", omp_get_nested( ));
      }
}
```

1

1

Le nombre de threads augmente de façon exponentielle en imbriquant les régions parallèles

gcc -fopenmp omp_set_nested.c -o omp_set_nested

./omp_set_nested

Exemple : omp_set_nested2.c

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

int main( )
{
int i, j;
 #pragma omp parallel for collapse(2)
   for ( i=0; i<N; i++) {
          for ( j=0; j<M; j++) {
                .....
               }
   }
}
```

Nombre de boucle qui doit être parallélisées, en comptant à partir de la boucle extérieur

```
#pragma omp for
for ( i = 0; i < 100; i ++)
f ( i /10 , i %10);
```

gcc -fopenmp omp_set_nested2.c -o omp_set_nested2
./omp_set_nested2

Directive : parallel (suite)

- Restrictions
  - Il est illégal de brancher (goto) dans ou hors d'une région parallèle
  - Seule une seule clause IF est autorisée
  - Seule une seule clause NUM_THREADS est autorisée

La portée d'une région parallèle

  - Les directives de début et de fin d'une région parallèle doivent être dans le même sous-programme.
  - Les branchements hors d'une région parallèle sont illégaux (i.e. le constructeur doit les repérer et les interdire).
- La portée lexicale d'une région parallèle
  - Elle inclut les instructions comprises entre les directives de début et de fin de cette région
- La portée dynamique
  - Elle est constituée par la portée lexicale et les sous-programmes (et les sous-sous-programmes) appelés au sein de cette portée lexicale.

Portée d'une région parallèle

```
int main(){
 #pragma omp parallel
   {
      !--- bloc1
      appel sub1()
      !--- bloc2
      appel sub2()
      !--- bloc3
  }
}// fin de main

procedure sub1(){ }
procedure sub3(){ }

procedure sub2(){
  call sub3()
}
```

- La portée lexicale couvre les blocs 1, 2, 3.

- La portée dynamique est constituée de ses 3 blocs mais aussi de sub1, sub2, sub3.

Directive de partage du travail

OpenMP dispose de deux directives permettant de répartir le travail sur les threads :

- la directive for pour répartir les itérations d'une boucle sur les threads SPMD/SIMD.
- les directives *sections* / *section* / …/ permettant d'exécuter des portions de code en concurrence sur plusieurs threads. Programmation MPMD/MIMD

Ces directives devront figurer à l'intérieur d'une région parallèle

Directive : sections

- La directive *sections* est une construction de non-itérative travail partagé.
- Il précise que les sections ci-joints de code doivent être réparties entre les threads de l'équipe.
- Les directives *section* indépendentes sont imbriquées dans une directive *sections*.
- Chaque section définie par *section* est exécutée une fois par un thread dans l'équipe.
- Les différentes sections peuvent être exécutées par des threads différents.
- Il est possible pour un thread d’exécuter plus d'une section si il est assez rapide et si l’implémentation le permet.

Directive : sections

- Il y a une barrière implicite à la fin d'une directive *sections*, sauf si la clause nowait est utilisée.

Directive sections /section

```
#include <stdio.h>
#include <omp.h>
int main(){
 #pragma omp parallel num_threads(4)
 {
    #pragma omp sections {
       #pragma omp section
       {
             printf ("id = %d, \n", omp_get_thread_num());
        }
       #pragma omp section
      {
           printf ("id = %d, \n", omp_get_thread_num());
      }
   }//sections
  }// parallel
}// main
```

Par défaut, il y a une «barrier» à la fin du bloc *section*. La clause “nowait” élimine la barrière

Section parallèle

```
#pragma omp parallel shared(n,a,b,c,d) private(i)
{
#pragma omp sections nowait {
  #pragma omp section
    for (i=0; i<n; i++)
        d[i] = 1.0/c[i];
  #pragma omp section
    for (i=0; i<n-1; i++)
        b[i] = (a[i] + a[i+1])/2;
 } /*-- End of sections --*/
} /*-- End of parallel region
```

Question

Qu'advient-il si on a :

- Plus de threads que de sections
  - Certains threads définis par le système ne vont pas exécuter de section, et les threads qui exécutent des sections varient d'une exécution à l'autre
- Moins de threads que de sections
  - L'implémentation va définir comment le surplus de sections sera exécuté

Directive for

- La directive for précise que les itérations de la boucle immédiatement après doivent être exécutées en parallèle par l'équipe de processeurs.
- Cela suppose une région parallèle a déjà été entamée, sinon la boucle est exécutée en série sur un seul processeur.

Directive for (Boucle parallèle)

- Diviser les indices d'une boucle for entre les threads disponibles
- Peut être utilisée avec la directive *parallel pour* exécuter en parallèle les parties de la boucle

➔Il faut que le calcul dans la boucle soit indépendant des résultats précédents

```
x = 0;
for ( i = 0 ; i < n ; i ++ ) {
   v[ i ] = x; // chaque v[i] dépend du x précédent!
   x += dx ;
}
```

Example: for et reduce

- On va calculer la somme des composantes d'un vecteur

```
int sumVecteur(int n, int* v){
    int i;
    int sum = 0;
    #ifdef _OPENMP
    #pragma omp parallel reduction(+:sum)
    #endif
    {
        #pragma omp for
        for ( i = 0 ; i < n ; i ++ ) // i automatiquement prive
            sum += v [ i ];
    }
    return sum;
}
```

Directive orpheline

```
#include <stdio.h>
#define SIZE 1024
void init ( int * vec ) {
   int i ;
   #pragma omp for
   for ( i = 0; i < SIZE ; i ++)
     vec [ i ] = 0;
}
int main ( ) {
int vec [SIZE] ;
#pragma omp parallel
init ( vec ) ;
return 0;
}
```

L'influence d'une région parallèle porte sur le bloc de code qui la suit directement (étendue statique) et sur les fonctions appelées dedans (étendue dynamique)

- Directives en dehors de l'étendue statique dites « orphelines »
- Liées à la région parallèle qui les exécute immédiatement
- Ignorées à l'exécution si non liées à une région parallèle

Clauses

clause reduction

- La clause REDUCTION effectue une réduction sur les variables qui apparaissent dans la liste.
- Une copie privée pour chaque variable de la liste est créée pour chaque thread. À la fin du bloc de réduction, la réduction de la variable est appliquée à toutes les copies privées de la variable partagée, et le résultat final est écrit dans la variable partagée globale.

reduction *(operator: list)*

- La variable reduction doit être declarée shared dans le contexte

clause reduction

- La clause de réduction est conçue pour être utilisée sur une construction de partage de tâches, dans laquelle la variable de réduction **x** est utilisée uniquement dans des situations qui ont une des formes suivantes :

x = x op expr
x = expr op x (sauf une soustraction pour op)
x binop = expr
x++
++x
x--
--x

x est une variable scalaire dans la liste
expr est une expression scalaire qui ne fait pas reference à ***x***
op n'est pas surchargé et il peut être : +, *, -, /, &, ^, |, &&, ||
binop n'est pas surchargé et il peut être : +, *, -, /, &, ^, |

clause schedule : Comment diviser les indices?

- Si on parallélise le *for, on donne des indices* à chaque thread
  - Quelle est la meilleure stratégie?
- On peut utiliser la clause ***schedule(option)***

où option est

- STATIC[, taille]
- DYNAMIC[, taille]
- GUIDED
- AUTO
- RUNTIME

clause schedule : option static

- Itérations de la boucle sont divisées en morceaux de taille chunk, puis attribuées statiquement au threads.
- Si la taille chunk n’est pas spécifiée, les itérations sont régulièrement (si possible) divisées par bloc contigus et attribuées au thread.

clause schedule : option dynamic

- Itérations de la boucle sont divisées en morceaux de taille chunk, puis attribuées statiquement au threads;
- lorsqu'un thread termine un morceau, il lui est attribué dynamiquement un autre morceau.

clause schedule : option guided

- Itérations sont affectées dynamiquement à des threads jusqu'à ce jusqu'à ce qu'il ne reste aucun bloc à attribuer.

- Semblable à DYNAMIC sauf que la taille du bloc diminue chaque fois qu'une parcelle de travail est confiée à un thread afin d'occuper tous les threads.

- La taille du bloc initial est proportionnelle à:

  number_of_iterations / number_of_threads

- Les sous blocs suivants sont proportionnels à :
  number_of_iterations_remaining / number_of_threads

- Si la division n'est pas parfaire, on prend la valeur entière supérieure
- Le paramètre morceau définit la taille minimale du bloc. La taille de bloc par défaut est 1.

clause schedule : option runtime ou auto

RUNTIME

La décision d'ordonnancement est différée jusqu'à l'exécution de la variable d’environnement OMP_SCHEDULE. Il est illégal de spécifier une taille de bloc de cette clause.

AUTO

La décision d'ordonnancement est déléguée au compilateur et / ou système d'exécution.

clauses : nowait

- nowait: Si elle est spécifiée, les threads ne se synchronisent pas à la fin de la boucle parallèle.
- Elle s'applique aux directives suivantes :
  - for
  - sections
  - single

clauses : ordered

- ordered : Indique que les itérations de la boucle doivent être exécutées comme ils le seraient dans un programme séquentiel.

La primitive omp_set_dynamic

void omp_set_dynamic(int dynamic_threads);

- dynamic_threads ≠ 0 alors le nombre de threads utilisés pour exécuter les régions parallèles suivantes peut être ajusté au mieux par le système
- dynamic_threads = 0 la modification dynamique est désactivé.

Clauses de statut des variables

OpenMP cible les architectures à mémoire partagée ; la plupart des variables sont donc partagées par défaut

On peut contrôler le statut de partage des données

- Quelles données des sections séquentielles sont transférées dans les régions parallèles et comment
- Quelles données seront visibles par tous les threads ou privées à chaque thread

Principales clauses :

  - private : définit une liste de variables privées
  - firstprivate : private avec initialisation automatique
  - lastprivate : private avec mise à jour automatique
  - shared : définit une liste de variables partagées
  - default : change le statut par défaut
  - reduction : définit une liste de variables à réduire

Déclaration du statut d'une variable

Le statut d'une variable utilisée à l'intérieur d'une région parallèle est soit partagé, soit privé.

- Si une **variable** est **partagée**, elle figure dans la mémoire globale. Chaque thread peut y accéder et modifier son contenu.
- Si une **variable** est **privée**, cette variable figure dans la pile de chaque thread et **sa valeur est indéfinie** en entrée de région parallèle. Un thread ne peut alors accéder qu'au contenu de la variable figurant dans sa pile, et lui seul peut la modifier.

Clause de déclaration du statut d'une variable

- Par défaut dans les régions parallèles, toutes les variables sont partagées (ou publiques).
- La déclaration se fait avec la clause PRIVATE ou SHARED.

- **PRIVATE** (liste_de_variables) pour des **variables privées**
- **SHARED** (liste_de_variables) pour des **variables partagées**

Private vs. Shared

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

using namespace std;

int main(){
    int x = 9;
    #ifdef _OPENMP
    #pragma omp parallel shared(x) num_threads (2)
    {
      x++;
      cout << "je suis le thread:" << omp_get_thread_num() << " x = " << x << endl;
    }
    #endif
    cout << "x vaut maintenant " << x << endl;
}
```

Ceci affiche 10 ou 11

Ceci affiche 11

x a la même valeur que la variable globale

Tout le monde partage une même variable globale. Ceci est très dangereux car il faut synchroniser les accès

Private vs. Shared

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

using namespace std;

int main(){
    int x = 9;
    #ifdef _OPENMP
    #pragma omp parallel private (x) num_threads (2)
    {
      x++;
      cout << "je suis le thread:" << omp_get_thread_num() << " x = " << x << endl;
    }
    #endif
    cout << "x vaut maintenant " << x << endl;
}
```

Ici x a une valeur indéfinie

Ici x vaut 9 comme au début

private crée une variable **non initialisée** propre à chaque thread. Il ne faut pas de synchronisation dans ce cas.

Autres modificateurs d'accès

- **default (private | shared | none)** pour spécifier un **statut par défaut**
- Si la clause default (none) est spécifiée, toute variable de la région parallèle devra avoir explicitement le statut private ou shared.
- Une variable privée peut recevoir la valeur qu'elle avait juste avant la région parallèle. Cette variable se verra attribuer le statut privé grâce à la clause **firstprivate (private avec initialisation automatique)**
- **lastprivate private avec mise à jour automatique)**

clause firstprivate

- Variables initialisées à partir de variables partagées
- Les objets C++ sont construits par copie

```
incr = 0;
#pragma omp parallel for firstprivate(incr)
for (i = 0; i <= MAX; i++) {
if ((i%2)==0) incr++;
A[i] = incr;
}
```

Chaque thread prend sa propre copie de incr avec une valeur initiale à 0

Exemple d'utilisation de la clause private

```
#include  <stdio.h>
#include  <unistd.h>
#include <stdlib.h>
int  main( ) {
 int  val ;
 #pragma omp parallel
 // private (val)
 {
    val = rand( ) ;
    sleep ( 1 ) ;
    printf ( "My val : %d \ n " , val ) ;
 }
 return 0 ;
}
```

- Compilez et exécutez ce code avec et sans private(val)
- Qu'observez-vous et pourquoi ?
- Est-ce risqué même avec la clause private ?

clause lastprivate

- La variable met à jour des variables partagées en utilisant la valeur de la dernière itération
- Les objets C++ sont mis à jour comme si par affectation

```
void sq2(int n, double *lastterm)
{
  double x; int i;
 #pragma omp parallel for lastprivate(x)
 for (i = 0; i < n; i++){
    x = a[i]*a[i] + b[i]*b[i];
   b[i] = sqrt(x);
 }
 *lastterm = x;
}
```

“x” a la valeur qu'elle détenait
pour la "dernière itération séquentielle"
itération (i.e., for i=(n-1))

Directive single

- #pragma omp single [clauses] { code_block }
- La directive single prend en charge les clauses suivantes OpenMP :
  - copyprivate
  - firstprivate
  - nowait
  - private

Directive threadprivate

- La directive threadprivate est utilisée pour faire des variables globales au fichier et les rendre persistantes au thread à travers l'exécution de plusieurs régions parallèles.

  #pragma omp threadprivate *(var)*

- var : une liste avec la virgule comme séparateur des variables que vous souhaitez rendre exclusives à un thread.

Directive threadprivate (suite)

- La directive doit apparaître après la déclaration de la liste des variables globales ou statiques concernées.
- Chaque thread reçoit sa propre copie de variable, afin que la donnée écrite par un thread ne soit pas visibles aux autres threads.
- La valeur des variables n'est pas spécifiée dans la première région parallèle sauf si la clause copyin est utilisée
- Ensuite, les variables sont préservées
- Le nombre de threads doit être fixe (omp_set_dynamic(0))
- Clauses possibles : aucune

Exercice

```
#include <stdio.h>
#include <omp.h>
int tid , tprivate , rprivate ;
#pragma omp threadprivate ( tprivate )
int main ( ) {
 // On interdit explicitement les threads dynamiques
 omp_set_dynamic (0) ;
 printf ( "Région parallèle 1\ n " ) ;
 #pragma omp parallel private ( tid, rprivate )
 {
    tid = omp_get_thread_num( ) ;
    tprivate = tid ;
    rprivate = tid ;
    printf ( "Thread %d : tprivate=%d rprivate=%d \ n " , tid , tprivate , rprivate ) ;
 }
 printf ( "Région parallèle  2\ n " ) ;
 #pragma omp parallel private ( tid , rprivat e)
 {
  tid = omp_get_thread_num( ) ;
  printf ( " Thread %d : tprivate=%d rprivate=%d \ n " , tid , tprivate , rprivate ) ;
 }
 return 0;
}
```

Clause copyin

- La clause copyin fournit un moyen pour affecter la même valeur aux variables threadprivate pour tous les threads de l'équipe.
- copyin(var)
- copyin s'applique aux directives suivantes :
  - parallel
  - for (OpenMP)
  - sections (OpenMP)

Clause copyprivate

- Demande la copie des valeurs des variables privées d'un thread dans les variables privées correspondantes des autres threads d'une même équipe
- Utilisable seulement avec la directive single
- Ne peut être utilisée en conjonction avec la clause nowait

Clause copyprivate

Utilisé avec une région single pour diffuser les valeurs des privates d'un thread membre de l'équipe au reste du l"equipe

```
#include <omp.h>
void input_parameters (int, int);  // chercher les valeurs des paramètres d'entrée
void do_work(int, int);
void main()
{
   int Nsize, choice;
  #pragma omp parallel private (Nsize, choice)
  {
    #pragma omp single copyprivate (Nsize, choice)
    input_parameters (Nsize, choice);
    do_work(Nsize, choice);
  }
}
```

Autres modificateurs d'accès

- Les variables private et firstprivate ont une adresse unique dans chaque thread parallèle : elles sont dupliquées.
- Elles ne sont pas accessibles en dehors de leur région parallèle.
- Seules, celles déclarées comme firstprivate sont initialisées au début de la région.
- Les variables privées qui conservent leur valeur d'une région parallèle à une autre sont appelées threadprivate.
- Les variables threadprivate peuvent être initialisées par le thread master au travers de la clause copyin.
- Exemples : indices de boucle (par défaut), variables locales, tableaux automatiques, ...

Statut d'une variable d'un sous-programme

- Dans un sous-programme appelé dans une région parallèle, toute variable locale est implicitement privée pour chaque thread

Statut d'une variable transmise par argument

- A l'intérieur d'une procédure ou d'une fonction appelée dans une région parallèle, toute variable passée par argument hérite du statut qu'elle avait dans la portée lexicale de la région parallèle

Statut d'une variable de type tableau dynamique

- Un tableau dynamique est implicitement partagé s'il est alloué en dehors de toute région parallèle.

- Un tableau dynamique aura le statut privé soit s'il reçoit explicitement ce statut dans la portée lexicale via la clause PRIVATE, soit s'il reçoit implicitement ce statut dans un sous-programme appelé dans une région parallèle.

Variables partagées

- Toutes les threads accèdent à la même instance de la variable (attention aux conflits !).
- Certaines variables doivent être partagées, comme les variables des opérations de réduction, i.e. opérations associatives usuelles (produit scalaire, maximum, somme, ET logique).

Synchronisation

- L'une des faiblesses d‘OpenMP
  - Besoin d'ordonner les actions

Directive de Synchronisation

- Considérons deux threads sur deux processeurs différents qui vont tous les deux essayer d'incrémenter une variable x en même temps (en supposant que x est initialement 0)

int x = 0;

THREAD 1: **increment(x)** **{** **x = x + 1;** **}** **THREAD 1:** **10 LOAD R1, @x** **20 ADD R1, 1** **30 STORE R1, @x**	**THREAD 2:** **increment(x)** **{** **x = x + 1;** **}** **THREAD 2:** **10 LOAD R1, @x** **20 ADD R1, 1** **30 STORE R1, @x**

Directive de Synchronisation

- Une séquence d'exécution possible:
  Thread 1 charge la valeur de l'emplacement x dans le registre R1.
  Thread 2 charges la valeur de l'emplacement x dans le registre R1.
  Thread 1 ajoute 1 au registre R1
  Thread 2 ajoute 1 au registre R1
  Thread 1 enregistre le contenu du registre R1 à l'emplacement x
  Thread 2 enregistre le contenu du registre R1 à l'emplacement x
- La valeur résultante de x sera de 1, pas 2 comme il se doit
- Pour éviter une telle situation, l'incrémentation de x doit être synchronisée entre les deux threads afin de s'assurer du bon résultat.
- OpenMP offre une variété de moyens de synchronisation qui contrôlent l'exécution de chaque thread par rapport à d'autres threads du pool

Erreur type : data race

```
#include <stdio.h>
#define MAX 1000
int main( ) {
int i , n = 0;
#pragma omp parallel for
for ( i = 0; i < MAX; i ++)
    n++;
p r i n t f ( "n = %d \ n " , n ) ;
return 0;
}
```

À l'exécution du programme ci-contre, la valeur affichée peut être inférieure à MAX :

- Accès concurrents à n
- Incrémentation non atomique
- Incrémentations « perdues »

Thread A	Thread B
load R1, @n	load R1, @n
add R1, 1	add R1, 1
store @n, R1	store @n, R1

Le thread B interrompt le thread A

Le thread A écrase la valeur écrite par B

Erreur type : défaut de cohérence

```
#include <stdio.h>
int main ( ) {
int fin = 0;
 #pragma omp parallel sections
 {
    #pragma omp section
    {
      while ( ! fin )
          printf ( "Pas fini \ n " ) ;
    }
   #pragma omp section
   {
     fin = 1;
     printf ( "Fini \ n " ) ;
   }
 }
 return 0;
}
```

À l'exécution du programme ci-contre, «Pas fini» peut être affiché après «Fini» :

- Interruption de la première section entre l'évaluation de fin et l'affichage (data race)
- Utilisation d'une vue temporaire obsolète de la mémoire partagée par le thread exécutant la première section (défaut de cohérence)

Erreur type : non synchronisation

```
#include <stdio.h>
#include <omp.h>
int main( ) {
double total , part1 , part2 ;
#pragma omp parallel num_threads(2)
{
   int tid ;
  tid = omp_get_thread_num( ) ;
 if ( tid == 0)
    part1 = 25;
 if ( tid == 1)
   part2 = 17;
 if ( tid == 0) {
   total = part1 + part2 ;
   printf ( "%g \ n " , total ) ;
 }
}
return 0;
}
```

À l'exécution du programme ci-contre, la valeur affiché peut être différente de 42

- Le thread 0 n'attend pas le thread 1 pour faire le calcul et l'affichage

Mécanismes de synchronisation

- Barrière pour attendre que tous les threads aient atteint un point donné de l'exécution avant de continuer
  - Implicite en fin de construction OpenMP (hors nowait)
  - Directive barrier
- Ordonnancement pour garantir un ordre global d'exécution
  - Clause ordered
- Exclusion mutuelle pour assurer qu'une seule tâche à la fois exécute une certaine partie de code
  - Directive critical
  - Directive atomic
- Attribution pour affecter un traitement à un thread donné
  - Directive master
- Verrou pour ordonner l'exécution d'au moins deux threads
  - Fonctions de la bibliothèque OpenMP

Directive barrier

- Synchronisation entre tous les threads d'une équipe
- Quand un thread arrive à la directive barrier il attend que tous les autres threads y soient arrivés ; quand cela arrive, les threads poursuivent leur exécution en parallèle
- Doit être rencontrée par tous les threads ou aucun : attention aux deadlocks
- Clauses possibles : aucune

Directive barrier

Arrête les threads jusqu'à ce que tout le monde ait fini son traitement courant

```
#pragma omp parallel
{
   f( ) ;
   g( ) ;
}
```

la directive parallèle appelle implicitement une barrière à la fin du bloc

```
#pragma omp parallel
{
   f( ) ;
   #pragma omp barrier
   g( ) ;
}
```

Avec la directive barrier, tous les threads doivent finir avec la fonction f() avant de pouvoir exécuter la fonction g()

Clause ordered

#pragma omp ordered

{

/ / Bloc

}

- Spécifie que les exécutions du bloc suivant la directive devront respecter l'ordre séquentiel
- Les threads s'attendent si besoin pour respecter cet ordre
- Les parties de la boucle parallèle non gardées par cette directive peuvent s'exécuter en parallèle
- Il faut aussi indiquer la clause ordered à la construction for

Directive critical

- La directive critical spécifie que seulement un thread peut entrer et travailler dans une zone dite *critique*

#pragma omp critical *[name]*

structured_block

Directive critical (suite)

- Si un thread est en cours d'exécution à l'intérieur d'une région critique et un autre thread atteint cette région critique et tente de l'exécuter, il sera bloqué jusqu'à ce que le premiers thread sort de cette région critique.
- Le nom, optionnel, permet à plusieurs différentes régions critiques d'exister:
  - Différentes régions CRITIQUES portant le même nom sont considérées comme une même région.
  - Toutes les sections critiques qui sont sans nom, sont traitées comme la même section

Directive critical (exemple)

```
#include <omp.h> main()
{
    int x;
    x = 0;
    #pragma omp parallel shared(x)
    {
          #pragma omp critical
        {
              x = x + 1;
        }
    }  /* end of parallel section */
}
```

Directive atomic

- La directive atomic précise qu'un emplacement de mémoire spécifique doit être mis à jour de façon atomique, plutôt que de laisser plusieurs threads tentent d'y écrire. Essentiellement, cette directive prévoit une section mini-critique.
- Exécuter une expression atomique (simple) sur une variable sensible

```
#pragma omp atomic
        variable := expression;
```

- Plus efficace que la directive critical dans ce cas
- Clauses possibles : aucune

Directive critical (exemple)

```
#include <omp.h> main()
{
    int x;
    x = 0;
    #pragma omp parallel shared(x)
    {
        #pragma omp critical
            x = x + 1;
    }  /* end of parallel section */
}
```

Les verrous

- Deux types de verrous et leurs fonctions associées :
  - omp_lock_t pour les verrous simples
  - omp_nest_lock_t pour les verrous à tours, pouvant être verrouillés plusieurs fois par un même thread et devant être déverrouillés autant de fois par ce thread pour être levés
- Alternatives plus flexibles à atomic et critical
- Préférer comme même atomic ou critical aux verrous quand c'est possible
- Verrous portables entre Unix et Windows
- Attention à bien les initialiser avec les fonctions adaptées
- Attention à ne pas verrouiller plusieurs fois un verrou simple

Exemple d'utilisation d'un verrou

```
#include <stdio.h> #include <omp.h> #define MAX 10
int main () {
 int i;
 omp_lock_t locka, lockb;
 omp_init_lock(&locka); omp_init_lock(&lockb);
 #pragma omp parallel sections private(i)
 {
     #pragma omp section
    for (i = 0; i < MAX; i++) {
        omp_set_lock(&locka);
        printf("Thread A: locked work\n");
       omp_unset_lock(&locka);
     }
     #pragma omp section
    for (i = 0; i < MAX; i++) {
        if (omp_test_lock(&lockb)) {
              printf("Thread B: locked work\n");
               omp_unset_lock(&lockb);
        } else {  printf("Thread B: alternative work\n");    }
    }
  }
omp_destroy_lock(&locka); omp_destroy_lock(&lockb);
return 0;
}
```

Directive flush

#pragma omp flush [(var)]

var est facultatif (toute la mémoire partagée)

- La directive FLUSH identifie un point de synchronisation où la mise en œuvre doit fournir une vue cohérente de la mémoire.
- Spécifie que tous les threads ont le même point de vue de la mémoire pour tous les objets partagés.
- Implicite après une région parallèle, un partage de travail (hors nowait), une section critique ou un verrou
- À faire après écriture dans un thread et avant lecture dans un autre pour partager une variable de manière cohérente
- Indispensable même sur un système à cohérence de cache
- Clauses possibles : aucune

Directive flush

- Utilisation de flush correctement est difficile et sujette à des bugs subtils
- Ne l'utilisez pas, sauf si vous êtes sûr à 100% que vous savez ce que vous faites!

Directive flush

```
#include <stdio.h>
#include <omp.h>
int main()
{
   int rang, nb_taches, synch=0;
   #pragma omp parallel private(rang,nb_taches)
   {
       rang=omp_get_thread_num();
       nb_taches=omp_get_num_threads();
       if (rang == 0) {
           #pragma omp flush(synch)
           while(synch != nb_taches-1);
       }
       else {
         #pragma omp flush(synch)
         while(synch != rang-1);
       }
      printf("Rang : %d ; synch : %d\n",rang,synch);
      synch=rang;
      #pragma omp flush(synch)
   }
  return 0;
}
```

Elle est utile dans une région parallèle pour rafraichir 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.

Exercice

- Corriger les codes donnés en exemple des erreurs type :
- Data race
- Défaut de cohérence
- Défaut de synchronisation

OpenMP tasks (d'après A. Duran, Tasking in OpenMP)

- Pourquoi des tâches ?

  Exemple : parcours de liste en parallèle avec OpenMP 2.5

```
void parcours_liste(liste l){
   element e = l → first;
   #pragma omp parallel firstprivate(e)
      while (e != NULL){
            #pragma omp single nowait
            traiter(e) ;
            e = e → next;
      }
}
```

- Peu naturel
- Mauvaises performances
- Peu composable

Pourquoi des tâches

```
void parcours_arbre (arbre *a){
  #pragma omp parallel sections
  {
     #pragma omp section
     if (a → gauche)
              parcours_arbre ( a → gauche) ;
     #pragma omp section
     if ( a → droit)
              parcours_arbre ( a → droit) ;
   }
   traiter (a) ;
}
```

trop de régions parallèles : surcoûts, synchronisations supplémentaires, pas toujours bien supporté par l'implémentation .

Gestion des tâches (openmp version 3)

#pragma omp task : déclaration d'une tâche fille

- Crée une nouvelle tâche composée du bloc d'instruction suivant la directive et de l'environnement de données au moment de la création de la tâche
- La tâche créée est soit exécutée immédiatement par le thread qui l'a créée (voir clauses), soit ajoutée au pool (différée)
- Hautement composable : imbrication possible dans des
  - régions parallèles
  - d'autres tâches
  - des directives « omp for », « omp sections », « omp single »

OpenMP tasks : portée des variables et synchronisations

- Privilégier : firstprivate (attention aux pointeurs...)
- Attention aux variables (non firstprivate) dans la pile :
  - La tâche mère ne doit pas les rendre invalides avant que les tâches filles ne s'en servent...
  - Solutions possibles :
    - Utiliser firstprivate
    - Déplacer les variables dans le tas (mais quand les «désallouer » ?)
    - Introduire des synchronisations
- Barrières OpenMP (#pragma omp barrier) → toutes les tâches créées par un thread de la team courante sont terminées à la sortie de la barrière
- Barrière de tâche : #pragma omp taskwait → La tâche courante attend alors la complétion de toutes ses tâches filles (seulement « filles directes », pas descendants)

Gestion des tâches (openmp version 3)

#pragma omp taskwait : attente de la fin des tâches filles

- Spécifie un point d'attente de la terminaison de toutes les sous-tâches créées par le thread rencontrant la directive
- Il s'agît d'une barrière spécifique aux tâches
- Clauses possibles : aucune

Parcours de liste en OpenMP 3.0

```
void parcours_liste(liste l) {
  element e = l → first;
  while (e != NULL) {
     #pragma omp task firstprivate(e)
     traiter(e) ;
     e = e →next;
  }
  #pragma omp taskwait
  // garantie ici la terminaison du parcours
}
```

Exercice : affichages possibles ?

```
#include <stdio.h>
int main () {
 #pragma omp parallel
 {
    #pragma omp single
   {
      printf("Hello ,\n");
      printf("world !\n");
   }
 }
return 0;
}
```

```
#include <stdio.h>
int main () {
   #pragma omp parallel
   {
      #pragma omp single
       {
           #pragma omp task
           printf("Hello ,\n");
           #pragma omp task
           printf("world !\n");
       }
    }
return 0;
}
```

```
#include <stdio.h>
int main () {
   #pragma omp parallel
   {
      #pragma omp single
      {
         #pragma omp task
         printf("Hello ,\n");
        #pragma omp task
        printf("world !\n");
        printf("Bye\n");
      }
   }
return 0;
}
```

```
#include <stdio.h>
int main () {
   #pragma omp parallel
   {
      #pragma omp single
      {
         #pragma omp task
         printf("Hello ,\n");
        #pragma omp task
        printf("world !\n");
        #pragma omp taskwait
         printf("Bye\n");
      }
    }
return 0;
}
```

Exercice paralleliser ce code

```
#include <stdio.h>
#include <stdlib.h>
int fib (int n) {
  i f ( n < 2)
     return n ;
  return fib(n-1) + fib(n-2);
}
int main ( int argc , char argv [ ] ) {
 int n = atoi ( argv [ 1 ] ) ;
 printf ( "fibo (%d ) = %d \ n ", n, fib(n) ) ;
 return 0;
}
```

- Parallélisez ce code de calcul des nombres de Fibonacci
- Comparez les performances avec le code séquentiel

La fonction fib () doit être appelée à l'intérieur d'une région parallèle pour les différentes tâches spécifiques à exécuter en parallèle.
En outre, un seul thread de la région parallèle devrait appeler fib () à moins que plusieurs calculs de Fibonacci simultanés sont souhaités.

Exemple suite de fibonacci (pattern diviser pour regner)

```
int fib (int n)
{
  int x,y;
  if ( n < 2 ) return n;
  #pragma omp task
   x = fib(n-1);
  #pragma omp task
  y = fib(n-2);
  #pragma omp taskwait
  return x+y
}
```

La variable n est privée dans les tâches

x est une variable privée
y est une variable privée

Qu'est ce qui est faux ici ?

Des variables privées des tâches sont indéfinies à l'extérieur de la tâche

Exemple suite de fibonacci (pattern diviser pour regner)

```
int fib ( int n )
{
  int x,y;
  if ( n < 2 ) return n;
  #pragma omp task shared(x)
  x = fib(n-1);
  #pragma omp task shared(y)
  y = fib(n-2);
  #pragma omp taskwait
  return x+y;
}
```

La variable n est privée dans les tâches

x & y sont partagées

Bonne solution

Nus avons besoin des deux valeurs pour calculer la somme

Conclusion : Principales directives OpenMP

- Construction de régions parallèles
  - parallel : crée une région parallèle sur le modèle fork-join
- Partage du travail
  - for : partage des itérations d'une boucle parallèle
  - sections : définit des blocs à exécuter en parallèle
  - single : déclare un bloc à exécuter par un seul thread
- Synchronisation
  - master : déclare un bloc à exécuter par le thread maître
  - critical : bloc à n'exécuter qu'un thread à la fois
  - atomic : instruction dont l'écriture mémoire est atomique
  - barrier : attente que tous les threads arrivent à ce point
- Gestion de tâches
  - task : déclaration d'une tâche fille
  - taskwait : attente de la fin des tâches filles

Résumé des directives de synchronisation

Il y a plusieurs constructions qui permettent de restreindre ou spécifier l'ordre d'accès à des données partagées

- directive master
  accès pour le thread de rang 0 uniquement
- directive single [nowait]
  accès pour un seul thread, non déterminé à l'avance
- directive atomic
  section critique formée d'une seule instruction
- section critical
  accès à un bloc d’instructions pour un seul thread à la fois
- directive barrier
  barrière de synchronisation globale

Résumé : Clauses / Directives

Le tableau ci-dessous résume les clauses acceptés sur les directives d'OpenMP

Clause	Directive					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF	●				●	●
PRIVATE	●	●	●	●	●	●
SHARED	●	●			●	●
DEFAULT	●				●	●
FIRSTPRIVATE	●	●	●	●	●	●
LASTPRIVATE		●	●		●	●
REDUCTION	●	●	●		●	●
COPYIN	●				●	●
COPYPRIVATE				●		
SCHEDULE		●			●	
ORDERED		●			●	
NOWAIT		●	●	●		

Résumé : Clauses / Directives

- Les directives OpenMP suivantes n'acceptent pas les clauses:
  - MASTER
  - CRITIQUE
  - BARRIÈRE
  - ATOMIQUE
  - FLUSH
  - ORDONNÉ
  - THREADPRIVATE
- Les implémentations peuvent différer du standard

Exemple : producteur/consomateur

```
flag = 0;
#pragma  omp parallel
{
   #pragma omp section
   {
      fillrand(N,A );
      #pragma omp flush
      flag = 1;
     #pragma omp flush(flag)
  }
 #pragma omp section
{
    while (!flag)
    #pragma omp flush(flag)
    #pragma omp flush
    sum = sum_array(N,A)
);
}
}
```

- First flush assure le drapeau est écrit après A
- Second flush assure drapeau est écrit à la mémoire

- Première flush assure que le drapeau est lu à partir de la mémoire
- Second flush assure le bon ordre des flush