Modèles de programmation parallèle

Frédéric Desprez
INRIA
LIP ENS Lyon
Equipe Avalon

Florence Zara
Université Lyon 1
LIRIS
Equipe SAARA

Quelques références

- **Cours Pthreads**, John Mellor-Crummey, Rice University
- **Cours OpenMP,** F. Roch (Grenoble)
- **The X-Kaapi's Application Programming Interface. Part I: Data Flow Programming,** F. Le Mentec, T. Gautier, V. Danjean
- **X-Kaapi C programming interface**, F. Le Mentec, T. Gautier, V. Danjean
- http://www.openmp.org
- http://www.openmp.org/mp-documents/spec30.pdf
- http://www.idris.fr
- http://ci-tutor.ncsa.illinois.edu/login.php
- **Using OpenMP , Portable Shared Memory Model**, Barbara Chapman
- **Cours composants,** C. Perez (Lyon)

Agenda

- KaaPI
- Pthreads
- OpenMP
- Composants parallèles
- MapReduce

Introduction

Modèle de programmation: comment (d)écrire un programme parallèle

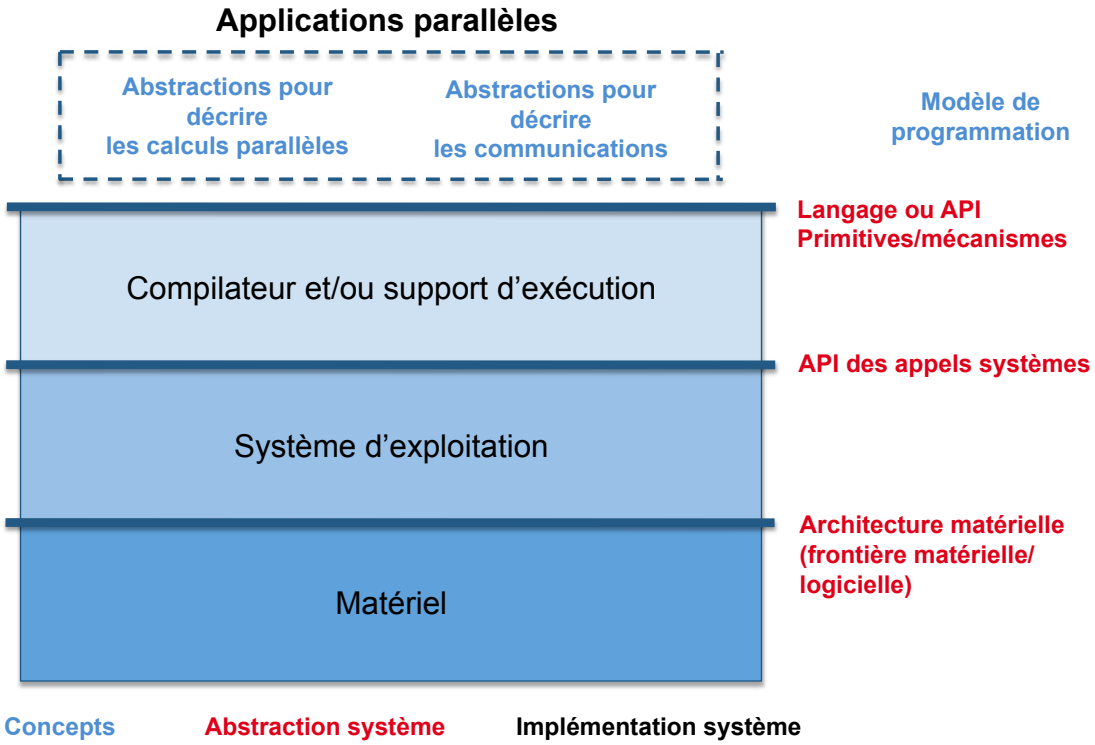
On a vu MPI (Message Passing Interface)

- L'utilisateur gère tout (distribution des données, distribution des calculs, synchronisation des processeurs, échanges des données)
- **Avantages**
  - Plus grand contrôle pour l'utilisateur
  - Performances (si code bien écrit !)
- **Inconvénients**
  - Assembleur du parallélisme
  - Portabilité (en termes de performances)
  - Moins de transparence

Autre solution

- Donner plus de travail aux environnements d'exécution !

Introduction



KAAPI

KAAPI

Projet INRIA-MOAIS

http://kaapi.gforge.inria.fr

Librairie C / C++ (simplement ajout d'instructions dans le code)

Portabilité

- Pas besoin de changer le code selon l'architecture
- Exécution possible en CPU, multi-CPU, grappe, GPU
- Nombre de nœuds variable

Langage de programmation parallèle de haut niveau

- Abstraction de l'architecture (mémoire virtuelle partagée)
- Ordonnancement dynamique géré par l'environnement

Concept

Problème initial est décomposé en **tâches de calcul**
Tâches s'exécutent en parallèle

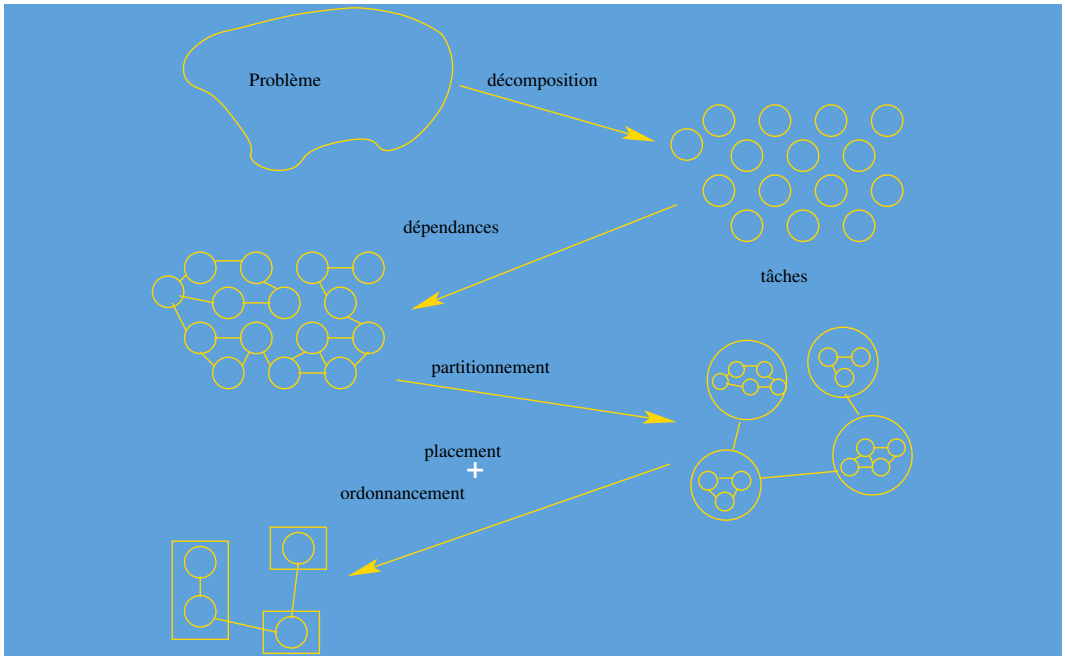
Tâches travaillent sur des données (virtuellement) partagées

- Différents accès possibles pour ces données
- Taille des données partagées définit la granularité

Exécution des tâches de calculs en parallèle

- Ordre d'exécution des tâches selon le **graphe de flots de données**
- Partitionnement du DFG (*Data Flow Graph*)
- Placement et ordonnancement sur les nœuds

Illustration



API C++ de Kaapi - Kaapi++

Concrètement dans le code

`#include «kaapi++»`

Utilisation du namespace **`ka::`**

```
// Initialisation de la librairie
int main(int argc, char** argv) {
 /* Join the initial group of computation */
  ka::Community com =
ka::System::join_communit(argc,argv);

 /* Start computation by spawning the main task */
  ka::SpawnMain<doit>()(argc, argv);

  /* Leave the community */
  com.leave();
  /* */
  ka::System::terminate();
}
```

Définition des tâches – Hello World

Prototype de la tâche

- Définit le nombre de paramètres / type / et les modes d'accès pour chacun des paramètres

```
/* Kaapi Hello task: print an integer n */
struct TaskHello: public ka::Task<1>::Signature<int>{};
```

Déclaration de la tâche

- Spécification de l'architecture (CPU / GPU)

```
/* CPU implementation */
template<>
struct TaskBodyCPU<TaskHello> {
 void operator() (int n) {
   std::cout << "Hello World !, n=" << n << std::endl;
 }
};
```


Exécution des tâches – Hello World

Appel de la fonction = création de la tâche (emploi du mot-clé **ka::spawn)**

```
/* The "doit" main task */
template<class T>
struct doit {
       void operator()(int argc, char** argv)
       {
          ka::Spawn<TaskHello>()(atoi(argv[1]));
       }
};
```

```
/* Ecriture de tous les arguments */
template<class T>
struct doit {
       void operator()(int argc, char** argv )
       {
              for (int i=1; i<argc; ++i)
                  ka::Spawn<TaskHello>()(atoi(argv[i]));
       }
};
```

Exécution des tâches

Création de tâches : **opération non bloquante**

```
/* The "doit" main task */
template<class T>
struct doit {
void operator()(int argc, char** argv )
{
       int a; int* b = ...;
       ka::Spawn<TaskThatRead_or_WriteData>()(&a, &b);
       /* here:
       1- Kaapi does not guarantee execution of the task
       2- a and b can accessed and should have a correct scope
       */
}
};
```


Exécution des tâches

Ce qui est garanti par Kaapi

- Une tâche débute quand tous ses paramètres d'entrée sont effectifs (contraintes issues du DFG)
- L'exécution parallèle produit toujours le même résultat que l'exécution séquentielle
- A la fin du programme, toutes les tâches créées ont été exécutées

Double Hello World !

```
/* The "doit" main task */
template<class T>
struct doit {
       void operator()(int argc, char** argv )
       {
              ka::Spawn<TaskHello>()(atoi(argv[1]));
              ka::Spawn<TaskHello>()(atoi(argv[2]));
       }
};
```

Traces possibles de l'exécution :

```
>./helloworld 1 2
Hello World !, n=1
Hello World !, n=2
```

Si un seul cœur, tjrs cela

```
>./helloworld 1 2
Hello World !, n=2
Hello World !, n=1
```

Possible si il y a au moins 2 cœurs

Synchronisation

Utilisation du mot-clé **`ka::Sync()`**

- Force l'exécution des tâches

```
/* The "doit" main task */
template<class T>
struct doit {
   void operator()(int argc, char** argv ) {
        ka::Spawn<TaskHello>()(atoi(argv[1]));
        ka::Sync();
        ka::Spawn<TaskHello>()(atoi(argv[2]));
   }
};
```

Toujours cette trace :

```
>./helloworld 1 2
Hello World !, n=1
Hello World !, n=2
```

Peut mettre une contrainte dans le graphe de flots : **`ka::Sync(<pointer>)`**

- Attente jusqu'à ce que la valeur pointée soit produite

Définition des tâches – Passage des paramètres

Par valeur

- Copie faite pour la tâche (exemple : `TaskHello`)

Par référence

- Pas de copie
- Tâche doit déclarer **le type d'accès à la donnée partagée**
  - Accès concurrents

    Accès en **lecture** : R (Read)

    Accès en **écriture cumulée** : CW (Cumulative Write)

  - Accès exclusifs

    Accès en **écriture** : W (Write)

    Accès en **lecture/écriture** : RW (Read Write)

Définition des tâches – Passage des paramètres

Prototype de la tâche définit les modes d'accès pour chacun des paramètres

- Ecriture : **`ka::W<T>`**
- Lecture : **`ka::R<T>`**
- Lecture / écriture : **`ka::RW<T>`**
- Ecriture cumulée : **`ka::CW<T>`**

Doivent correspondre à la déclaration de la tâche :

- **`ka::pointer_w<T>`**
- **`ka::pointer_r<T>`**
- **`ka::pointer_rw<T>`**
- **`ka::pointer_cw<T,F>`**

Définition des tâches – Passage des paramètres

Prototype de la tâche

```
struct TaskFibo: public ka::Task<2>::Signature<ka::W<int>,int>{};
```

Déclaration de la tâche

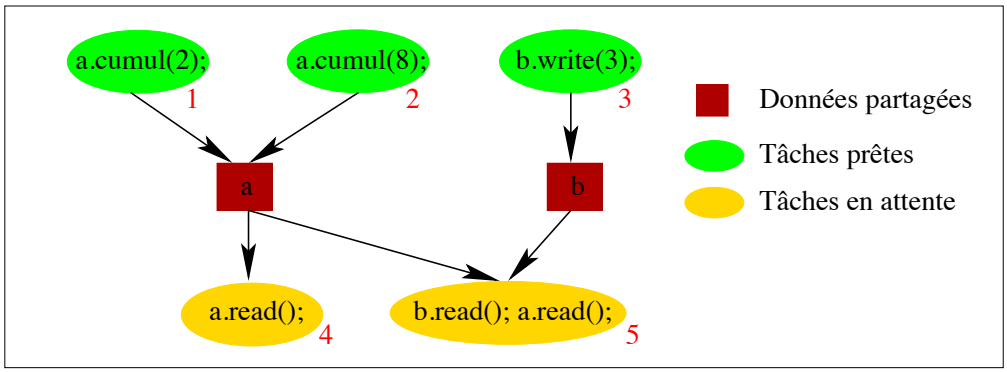
```
/* CPU implementation */
template<>
struct TaskBodyCPU<TaskFibo> {
        void operator() (ka::pointer_w<long> res, long n ) { ... }
};
```


Dépendance entre les tâches

Exécution des tâches selon l'ordre de référence

Ordre basé sur le **graphe de flots de données**

- Construit dynamiquement durant l'exécution
- Décrit les dépendances entre les tâches et les données



Dépendance entre les tâches

Deux tâches partagent une donnée commune ssi ils accèdent à la même donnée en mémoire

- Même donnée == même pointeur

```
ka::pointer<T> a;
ka::pointer<T> b = a+100;
ka::Spawn<TaskRW1>()(a); /* rw on a */
ka::Spawn<TaskRW2>()(b); /* rw on b */
```

TaskRW1 et TaskRW2 sont **indépendantes**

Ordonnancement

Ordonnancement semi-statique du GFD à effectuer

- Placement des données sur les processeurs
- Ordonnancement des tâches : date + n° processeur

→ Effectués dans un souci de performance

Ordonnancement local des tâches

- Respecte les dépendances

Heuristiques d'ordonnancement fournies par Kaapi

- Cyclic
- LPTF (coûts associés aux tâches)
- ORB (coûts + localisation spatiale des données)
- Basées sur un partitionnement du graphe (Scotch)
- Possibilité de spécialiser l'ordonnancement
- Ordonnancement par vol de tâches

Exemples / TP

- Hello
- Fibonacci
- N Reines
- Factorisation LU de matrices

- Affichage du GFD

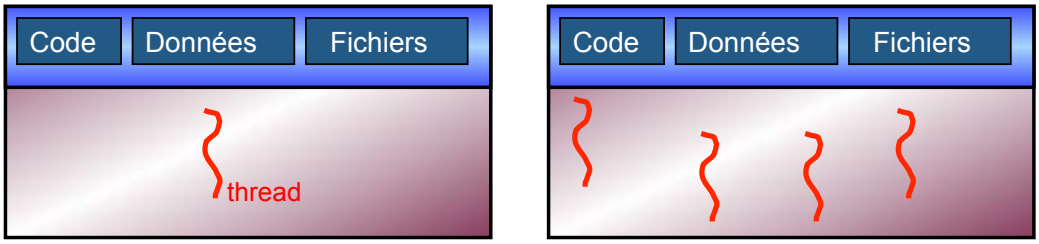
PTHREADS

Thread

- Un processus est défini par la ressource qu'il utilise et par son état d'exécution (PC, registres)

- Un thread est une subdivision du processus
  - Un flot de contrôle dans un processus

- Un thread (processus léger) est un processus qui partage son bloc de code, son bloc de données et ses ressources du SE (ex: fichiers) avec d'autres processus. Cependant il a ses propres valeurs de PC, registre, un espace mémoire propre.



Pourquoi les threads

Réactivité: un processus peut être subdivisé en plusieurs threads.
Exemple
- 1 thread dédié à l'interaction avec les usagers
- 1 thread dédié au traitement des données

Partage de ressources: mémoire, fichiers

Économie: les threads sont moins lourds dans leur entretien que les processus
- Création, commutation beaucoup plus simples
- pas besoin d 'allocation de grosse mémoire ...

Utilisation de multiprocesseurs: les threads peuvent s'exécuter en parallèle sur des UCT différentes

Pourquoi les threads, suite

- **Portable, modèle de programmation largement adopté**
  - Utilisé à la fois dans les systèmes séquentiels et parallèles
- **Utile pour masquer la latence**
  - Latence due aux entrées/sorties, aux communications
- **Utile pour l'ordonnancement et l'équilibrage des charges**
  - Surtout pour la concurrence dynamique
- **Relativement facile à programmer**
  - Significativement plus simple que le passage de message !

Partage de ressources

Il existe de nombreux cas où un processus requiert une ressource partagée

Les différents threads d'un processus partagent l'espace d'adressage et les ressources d'un processus

- lorsqu'un thread modifie une variable (non locale), tous les autres threads voient la modification
- un fichier ouvert par un thread est accessible aux autres threads (du même processus)

API de Threads POSIX (Pthreads)

- API standard supportée par la plupart des vendeurs des machines

- Les concepts derrière l'interface des pthreads est largement applicable
  - Largement indépendants de l'API
  - Utile pour programmer des applications avec d'autres API de threads
    - NT threads
    - Threads Solaris
    - Threads Java
    - …

Création d'un pthread

Appel asynchrone de la fonction thread_function dans un nouveau thread

```
#include <pthread.h>
int pthread_create(
  pthread_t *thread_handle, /* returns handle here */
  const pthread_attr_t *attribute,
  void * (*thread_function)(void *),
  void *arg); /* single argument; perhaps a structure */
```

Attributs (`attribute`) créés par `pthread_attr_init`

Contient des détails à propos
- si la police d'ordonnancement est hérité ou explicite
- politique d'ordonnancement, priorité
- taille de la pile, …

Attendre la terminaison d'un thread

Suspendre l'exécution du thread appelant jusqu'à ce que thread termine

```
#include <pthread.h>
int pthread_join ( pthread_t thread, /* thread id */
void **ptr); /* ptr to location for return code a
                terminating thread passes to
                pthread_exit                        */
```


Exemple: création et terminaison

```
#include <pthread.h>
#include <stdlib.h>
#define NUM_THREADS 32
void *compute_pi (void *);
...
int main(...) {
   ...
   pthread_t p_threads[NUM_THREADS];
   pthread_attr_t attr;
   pthread_attr_init(&attr);
   for (i=0; i< NUM_THREADS; i++) {
      hits[i] = i;
      pthread_create(&p_threads[i], &attr, compute_pi,
         (void*) &hits[i]);
   }
   for (i=0; i< NUM_THREADS; i++) {
      pthread_join(p_threads[i], NULL);
      total_hits += hits[i];
   }
   ...
```

default attributes

thread function

thread argument

Sections critiques et exclusion mutuelle

- Section critique = on doit exécuter le code par un seul thread à la fois

```
/* threads compete to update global variable best_cost */
if (my_cost < best_cost) best_cost = my_cost;
```

- Le lock de mutex bloque les sections critiques dans les Pthreads
  - Etat des locks: bloqué ou pas
  - Seul un thread peut bloquer un lock de mutex à un instant donné
- Utiliser les locks de mutex
  - Demander un lock avant d'entrer dans une section critique
  - Entrer dans la section critique lorsque le lock est obtenu
  - Relacher le lock à la sortie de la section critique
- Opérations

```
int pthread_mutex_init (pthread_mutex_t *mutex_lock,
                        const pthread_mutexattr_t *lock_attr)
int pthread_mutex_lock(pthread_mutex_t *mutex_lock)
int pthread_mutex_unlock(pthread_mutex_t *mutex_lock)
```


Exemple: réduction avec des locks mutex

```
pthread_mutex_t cost_lock;
...
int main() {
   ...
   pthread_mutex_init(&cost_lock, NULL);
   ...
}
void *find_best(void *list_ptr) {
   ...
   pthread_mutex_lock(&cost_lock);    /* lock the mutex */
   if (my_cost < best_cost)
      best_cost = my_cost;
   pthread_mutex_unlock(&cost_lock); /* unlock the mutex */
}
```

use default (normal) lock type

critical section

Producteur/consommateur

Contraintes

- **Thread producteur**
  - On ne doit pas écraser un tampon partagé tant que la tâche précédente n'a pas été récupérée par un consommateur

- **Thread consommateur**
  - Ne doit pas récupérer une tâche tant qu'une tâche n'est pas disponible dans la file
  - Doit récupérer une tâche après l'autre

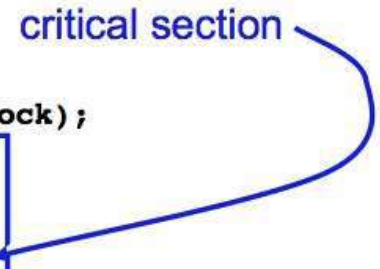
Producteur/consommateur

```
pthread_mutex_t task_queue_lock;
int task_available;
...
main() {
    ...
    task_available = 0;
    pthread_mutex_init(&task_queue_lock, NULL);
    ...
}
void *producer(void *producer_thread_data) {
    ...
    while (!done()) {
       inserted = 0;
       create_task(&my_task);
       while (inserted == 0) {
           pthread_mutex_lock(&task_queue_lock);
           if (task_available == 0) {
              insert_into_queue(my_task); task_available = 1;
              inserted = 1;
           }
           pthread_mutex_unlock(&task_queue_lock);
       }
    }
}
```

critical section

Producteur/consommateur

```
void *consumer(void *consumer_thread_data) {
    int extracted;
    struct task my_task;
    /* local data structure declarations */
    while (!done()) {
       extracted = 0;
       while (extracted == 0) {
          pthread_mutex_lock(&task_queue_lock);
          if (task_available == 1) {
             extract_from_queue(&my_task);
             task_available = 0;
             extracted = 1;
          }
          pthread_mutex_unlock(&task_queue_lock);
       }
       process_task(my_task);
    }
```



Surcoût des locks

- Ils renforcent la séquentialisation
  - Les threads doivent exécuter les sections critiques une après l'autre
- Les grandes sections critiques peuvent dégrader les performances de manière importante
- Réduire le surcoût en recouvrant les calculs avec l'attente

```
int pthread_mutex_trylock(pthread_mutex_t *mutex_lock)
```

  - Acquérir un lock s'il est disponible
  - Retourner EBUSY s'il n'est pas disponible
  - Permet à un thread de faire autre chose tant qu'il n'est pas disponible

Variables de condition pour la synchronisation

Variable condition: associée avec un prédicat et un mutex

Utiliser une variable de condition

- Un thread peut se bloquer tant qu'une condition n'est pas vraie
  - Un thread bloque un mutex
  - Il teste un prédicat définit par une variable partagée
    Si le prédicat est faux, alors attendre sur la variable condition
    L'attente sur une variable condition débloque le mutex associé
- Lorsqu'un thread rend un prédicat vrai
  - Ce thread peut signaler la variable condition pour
    se réveiller sur un thread en attente
    se réveiller sur tous les threads en attente
  - Lorsque le thread relache le mutex, il est passé au premier thread en attente

API pour les variables conditions

```
/* initialize or destroy a condition variable */
int pthread_cond_init(pthread_cond_t *cond,
      const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);

/* block until a condition is true */
int pthread_cond_wait(pthread_cond_t *cond,
      pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
      pthread_mutex_t *mutex, const struct timespec *wtime);

/* signal one or all waiting threads that condition is true */
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Terminer si temps dépassé

Réveiller un thread

Réveiller tous les threads

Producteur consommateur

```
pthread_cond_t cond_queue_empty, cond_queue_full;
pthread_mutex_t task_queue_cond_lock;
int task_available;
/* other data structures here */

main() {
  /* declarations and initializations */
  task_available = 0;
  pthread_init();
  pthread_cond_init(&cond_queue_empty, NULL);
  pthread_cond_init(&cond_queue_full, NULL);
  pthread_mutex_init(&task_queue_cond_lock, NULL);
  /* create and join producer and consumer threads */
}
```

Valeurs par défaut

Producteur consommateur

```
void *producer(void *producer_thread_data) {
  int inserted;
  while (!done()) {
    create_task();
    pthread_mutex_lock(&task_queue_cond_lock);
    while (task_available == 1)
      pthread_cond_wait(&cond_queue_empty,
          &task_queue_cond_lock);
    insert_into_queue();
    task_available = 1;
    pthread_cond_signal(&cond_queue_full);
    pthread_mutex_unlock(&task_queue_cond_lock);
}
```

Releases mutex on wait

reacquires mutex when woken

Producteur consommateur

```
void *consumer(void *consumer_thread_data) {
  while (!done()) {
    pthread_mutex_lock(&task_queue_cond_lock);
    while (task_available == 0)
      pthread_cond_wait(&cond_queue_full,
      &task_queue_cond_lock);
    my_task = extract_from_queue();
    task_available = 0;
    pthread_cond_signal(&cond_queue_empty);
    pthread_mutex_unlock(&task_queue_cond_lock);
    process_task(my_task);
  }
}
```

Releases mutex on wait

reacquires mutex when woken

OPENMP

Modèle de programmation multi-tâches sur architecture à mémoire partagée

Plusieurs tâches s'exécutent en parallèle

La mémoire est partagée (physiquement ou virtuellement)

Les communications entre tâches se font par lectures et écritures dans la mémoire partagée.

Par ex. les processeurs multicoeurs généralistes partagent une mémoire commune

Les tâches peuvent être attribuées à des « cores » distincts

Modèle de programmation multi-tâches sur architecture à mémoire partagée

La librairie Pthreads : librairie de threads POSIX, adoptée par la plupart des OS

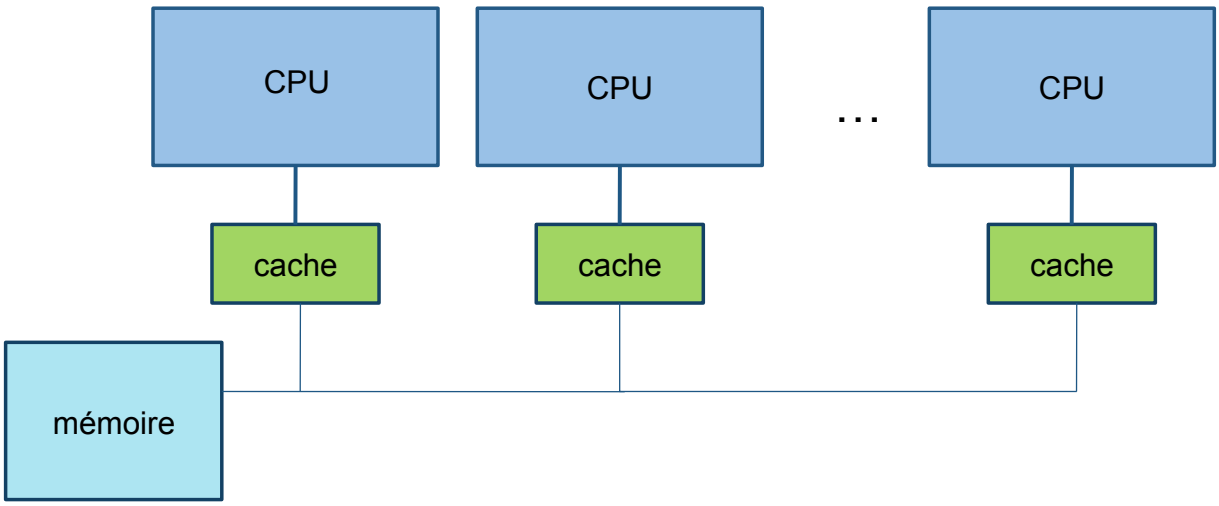
L'écriture d'un code nécessite un nombre considérable de lignes spécifiquement dédiées aux threads

Ex : paralléliser une boucle implique :

déclarer les structures de thread, créer les threads, calculer les bornes de boucles, les affecter aux threads, …

OpenMP : une alternative plus simple pour le programmeur

Programmation multi-tâches sur les architectures UMA

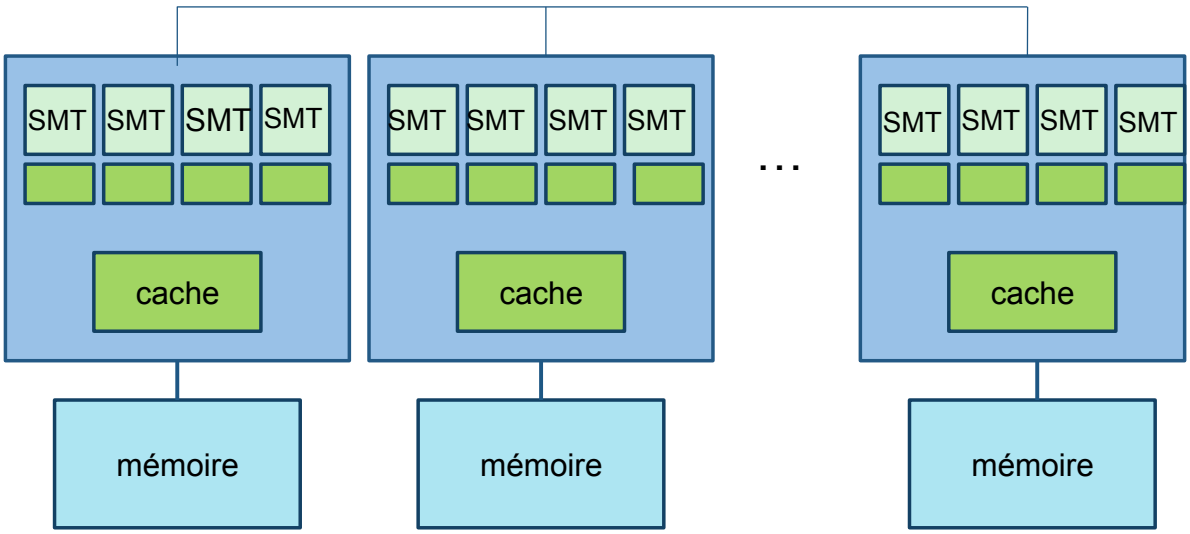


La mémoire est commune,

Des architectures à accès mémoire uniforme (UMA)

Un problème inhérent : les contentions mémoire

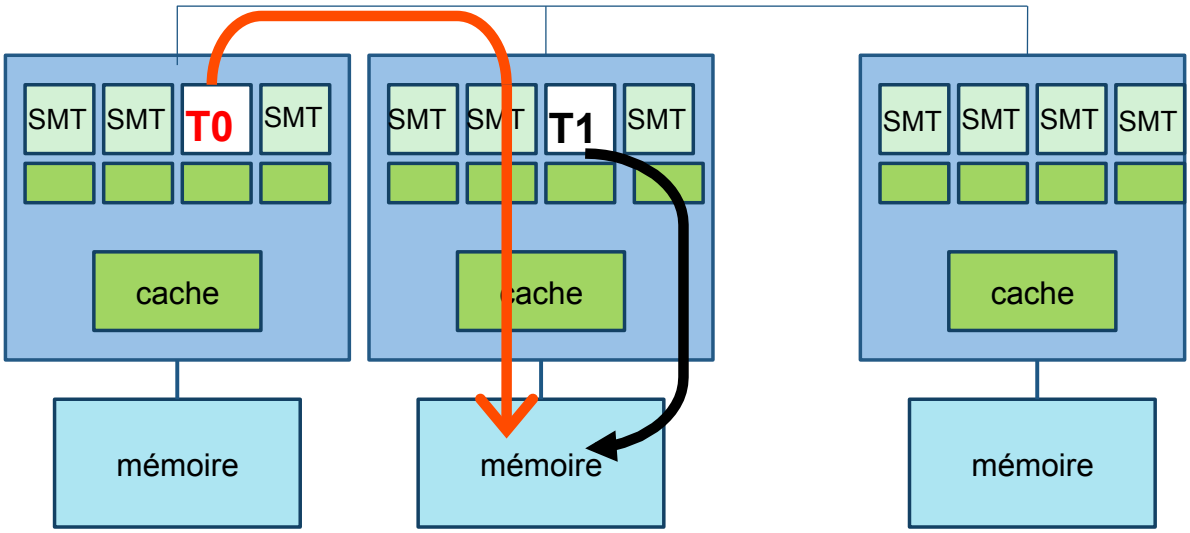
Programmation multi-tâches sur les architectures multicoeurs NUMA



La mémoire est directement attachée aux puces multicoeurs

Des architectures à accès mémoire non uniforme NUMA

Programmation multi-tâches sur les architectures multicoeurs NUMA



ACCES DISTANT

ACCES LOCAL

Caractéristiques du modèle OpenMP

Avantages

- Gestion de « threads » transparente et portable
- Facilité de programmation

Inconvénients

- Problème de localité des données
- Mémoire partagée mais non hiérarchique
- Efficacité non garantie (impact de l'organisation matérielle de la machine)
- Passage à l'échelle limité, parallélisme modéré

OpenMP
(Open specifications for MultiProcessing)

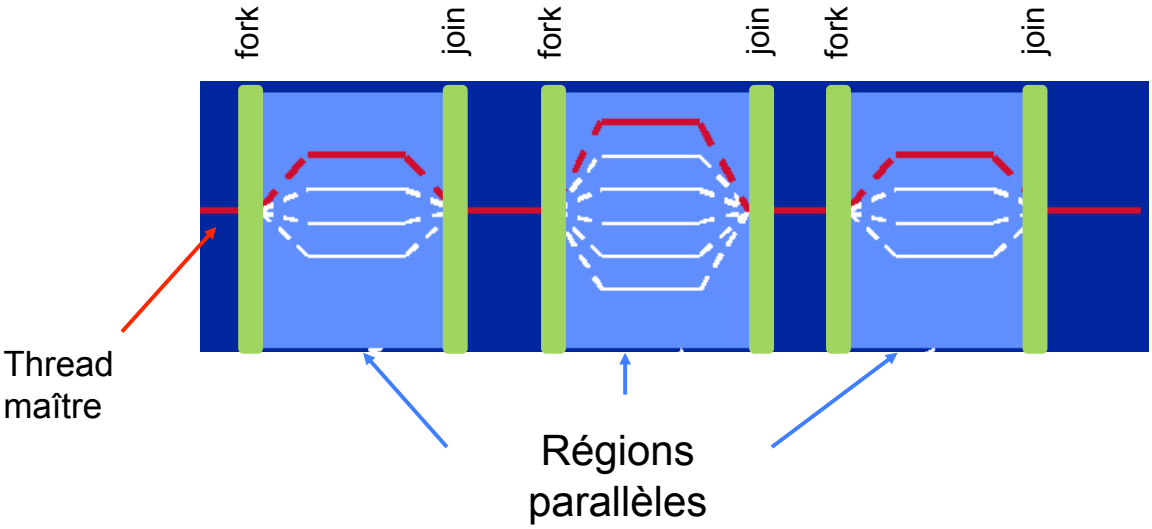
Introduction
Structure d'OpenMP
portée des variables
Constructions de partage du travail
Construction task
Synchronisation
Performances
Conclusion

Introduction : supports d'OpenMP

La parallélisation multi-tâches existait avant pour certains compilateurs (Ex:Cray,NEC,IBM)
OpenMP est une API pour un modèle à mémoire partagé
Spécifications pour les langages C/C++, Fortran
Supporté par beaucoup de systèmes et de compilateurs
- OpenMP-2 2000 , OpenMP-3 2008, OpenMP-4 2013
- Specs : http://openmp.org

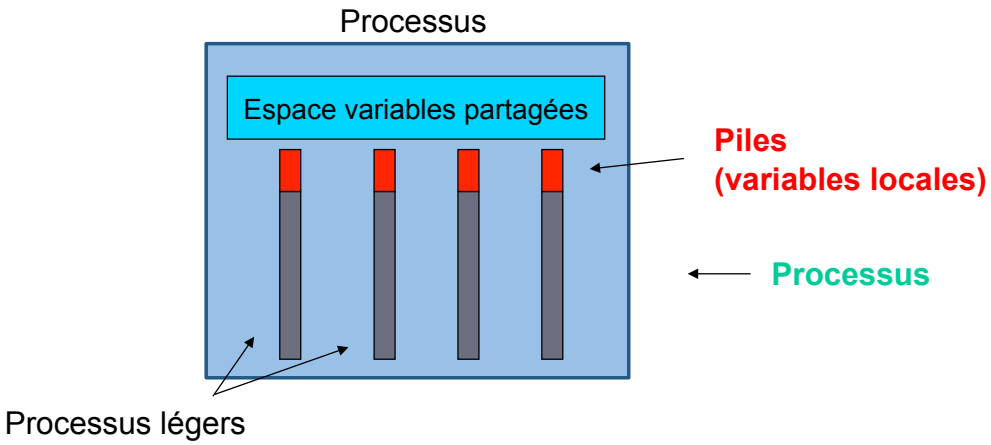
Introduction : Modèle d'exécution

Un programme OpenMP est exécuté par un processus unique (sur un ou plusieurs cores)



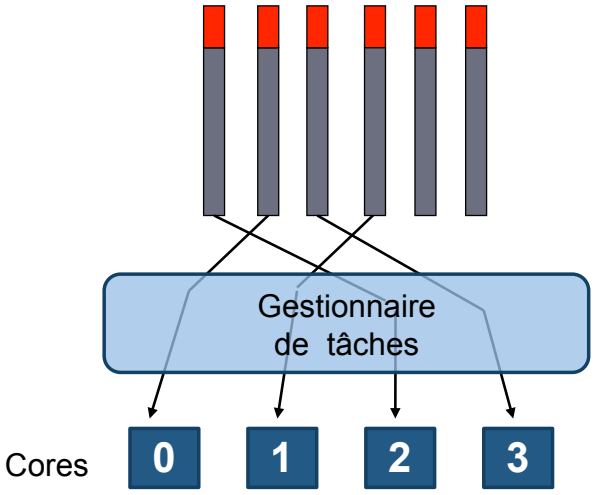
Introduction : les threads

Les threads accèdent aux mêmes ressources que le processus.
Elles ont une pile (stack, pointeur de pile et pointeur d'instructions propres)



Introduction : exécution d'un programme OpenMP sur un multicoeur

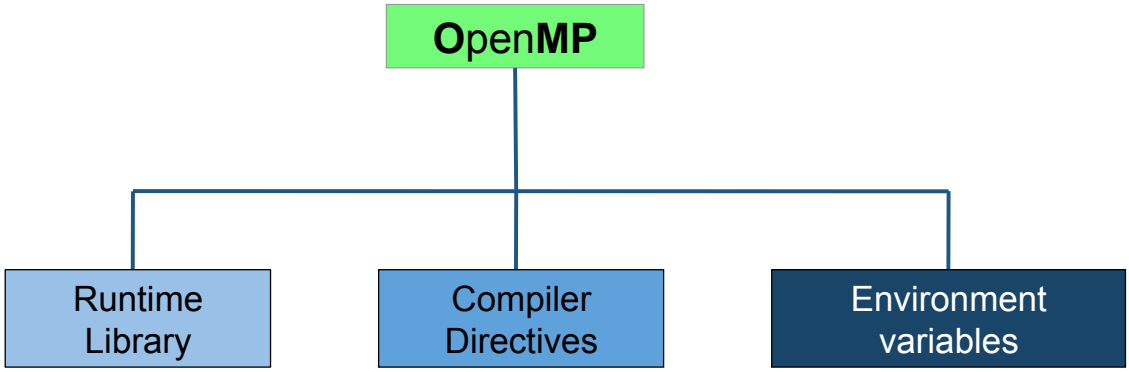
Le gestionnaire de tâches du système d'exploitation affecte les tâches aux cores.



OpenMP

- Introduction
  - Structure d'OpenMP
- Portée des données
- Constructions de partage du travail
- Construction task
- Synchronisation
- Performances
- Conclusion

Structure d'OpenMP : architecture logicielle



Structure d'OpenMP : format des directives/pragmas

- Sentinelle directive [clause[clause]..]

Fortran

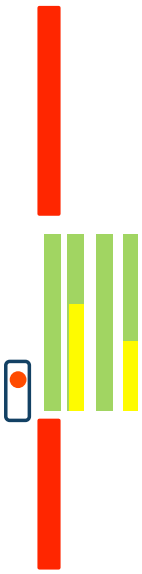
```
!$OMP PARALLEL PRIVATE(a,b)  &
!$OMP FIRSTPRIVATE(c,d,e)
...
!$OMP END PARALLEL
```

C/C++

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

- La ligne est interprétée si option openmp à l'appel du compilateur sinon commentaire
- ➔ portabilité

Structure d'OpenMP : Construction d'une région parallèle



fortran

```
PROGRAM example

!$  USE OMP_LIB

Integer :: a, b, c

!  Code sequentiel  exécuté par le maître

!$OMP PARALLELPRIVATE(a,b)  &
!$OMP SHARED(c)
.
!  Zone parallèle exécutée par toutes les
! threads

!$OMP END PARALLEL

! Code séquentiel

END PROGRAM example
```

C/C++

```
#include <omp.h>

Main  () {

Int a,b,c:

/*  Code séquentiel  exécuté par  le maître */

#pragma omp parallel private(a,b)  \
                              shared(c)
   {
      /* Zone parallèle exécutée par toutes les
         threads */
    }

/*  Code Sequentiel */

}
```


Clause IF de la directive PARALLEL

Création conditionnelle d'une région parallèle clause IF(expression_logique)

fortran

```
!Code séquentiel

!$OMP PARALLEL IF(expr)
! Code parallèle ou séquentiel suivant la valeur de expr
!!$OMP END PARALLEL

! Code séquentiel
```

L'expression logique sera évaluée avant le début de la région parallèle.

Structure d'OpenMP : prototypage

Il existe

- Un module fortran 95 OMP_LIB
- un fichier d'inclusion C/C++ omp.h

qui définissent les prototypes de toutes les fonctions de la librairie OpenMP

Fortran

```
Program example
!$  USE OMP_LIB
!$OMP PARALLEL PRIVATE(a,b) &
              ...
tmp= OMP_GET_THREAD_NUM()
!$OMP END PARALLEL
```

C/C++

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


Threads OpenMP

Définition du nombre de threads

Via une variable d'environnement OMP_NUM_THREADS
Via la routine : OMP_SET_NUM_THREADS()
Via la clause NUM_THREADS() de la directive PARALLEL

Les threads sont numérotés

- le nombre de threads n'est pas nécessairement égal au nombre de cores physiques
- Le thread de numéro 0 est la tâche maître
- OMP_GET_NUM_THREADS() : nombre de threads
- OMP_GET_THREAD_NUM() : numéro de la thread
- OMP_GET_MAX_THREADS() : nb max de threads

Structure d'OpenMP : Compilation et exécution

ifort (ou icc) –openmp prog.f (INTEL)
f90 (ou cc ou CC) –openmp prog.f (SUN Studio)
gcc/gfortran –fopenmp –std=f95 prog.f (GNU)
export OMP_NUM_THREADS=2

./a.out

ps –eLF
USER PID PPID LWP C NLWP SZ RSS PSR …

OpenMP

- Introduction
- Structure d'OpenMP
  - Portée des variables
- Constructions de partage du travail
- Construction task
- Synchronisation
- Performances
- Conclusion

Rappel : allocation mémoire- portée des variables

Variables statiques et automatiques

- statique : emplacement en mémoire défini dès sa déclaration par le compilateur
- automatique : emplacement mémoire attribué au lancement de l'unité de programme où elle est déclarée (existence garantie que pendant l'exécution de l'unité)

Variables globales

globale : déclarée au début du programme principal, elle est statique

2 cas

- initialisées à la déclaration (exemple : parameter, data)
- non-initialisées à la déclaration (exemple : en fortran les common, en C les variables d'unités de fichier, les variables externes ou static)

Variables locales

variable à portée restreinte à l'unité de programme où elle est déclarée, 2 catégories

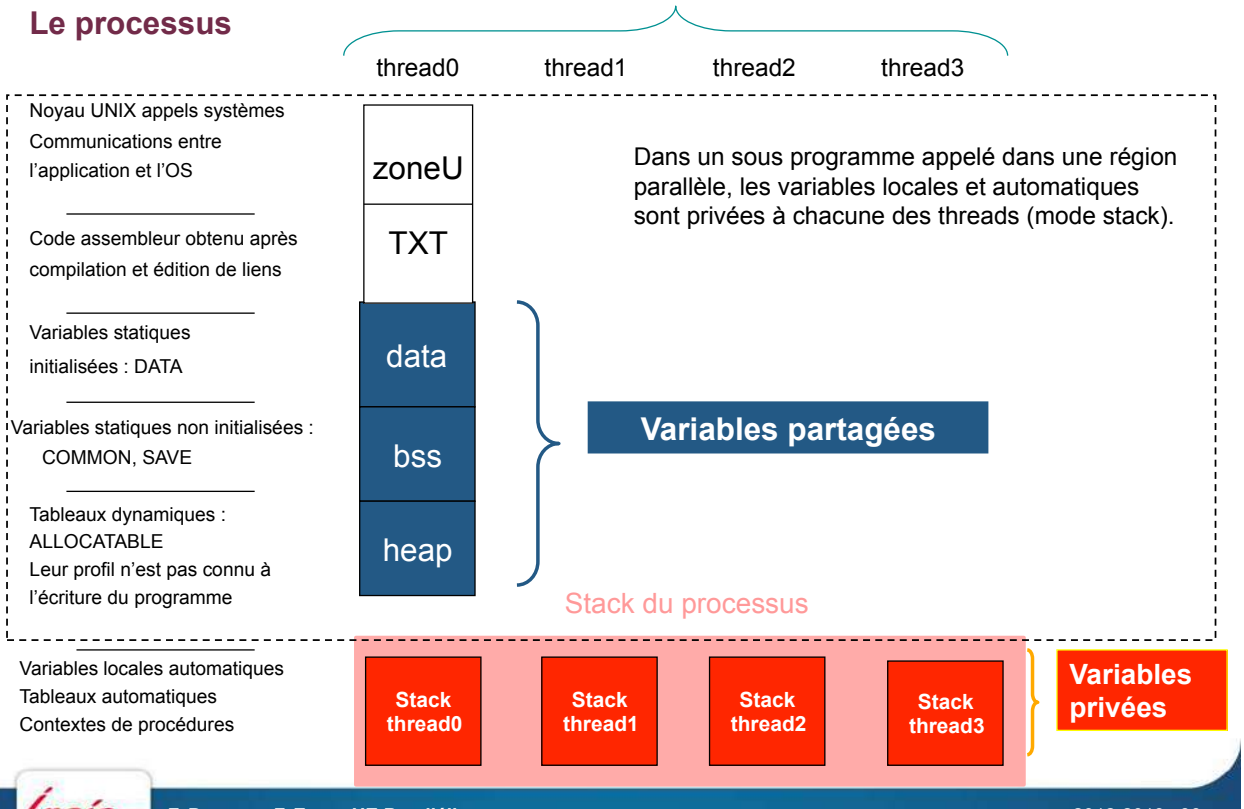
- Variables locales automatiques
- Variables locales rémanentes (statiques) si elles sont

a) initialisées explicitement à la déclaration,

b) déclarées par une instruction de type DATA,

c) déclarées avec l'attribut SAVE => valeur conservée entre 2 appels

Rappel : stockage des variables



Statut d'une variable

Le statut d'une variable dans une zone parallèle est

- soit SHARED, elle se trouve dans la mémoire globale
- soit PRIVATE, elle est dans la pile de chaque thread, sa valeur est indéfinie à l'entrée de la zone

Déclarer le statut d'une variable
!$OMP PARALLEL PRIVATE(list)
!$OMP PARALLEL FIRSTPRIVATE(list)
!$OMP PARALLEL SHARED(list)
Déclarer un statut par défaut
Clause DEFAULT(PRIVATE|SHARED|NONE)

```
program private_var.f
!$USE OMP_LIB
integer:: tmp =999
Call OMP_SET_NUM_THREADS(4)
!$OMP PARALLEL PRIVATE(tmp)
   print *, tmp
   tmp= OMP_GET_THREAD_NUM()
   print *, OMP_GET_THREAD_NUM(), tmp
!$OMP END PARALLEL
print *, tmp
end
```


Clauses de la directive PARALLEL

- NONE
  - Equivalent de l' IMPLICIT NONE en Fortran. Toute variable devra avoir un statut défini explicitement
- SHARED (liste_variables)
  - Variables partagées entre les threads
- PRIVATE (liste_variables)
  - Variables privées à chacune des threads, indéfinies en dehors du bloc PARALLEL
- FIRSTPRIVATE (liste_variables)
  - Variable initialisée avec la valeur que la variable d'origine avait juste avant la section parallèle
- DEFAULT (PRIVATE | SHARED|NONE)

Ex !$OMP PARALLEL DEFAULT(PRIVATE) SHARED (X)

Statut d'une variable transmise par arguments

Dans une procédure, les variables transmises par argument héritent du statut défini dans l'étendue lexicale de la région (code contenu entre les directives PARALLEL et END PARALLEL)

```
program statut
!$  USE OMP_LIB
integer :: a=100, b

!$OMP PARALLEL PRIVATE(b)
            call sub(a,b)
            print *,b
!$OMP END PARALLEL
end program  statut

Subroutine sub(x,y)
integer x,y
  y = x + OMP_GET_THREAD_NUM()
End subroutine sub
```


La directive THREADPRIVATE

La directive THREADPRIVATE

Permet de rendre privé aux threads

- un bloc COMMON (Fortran)
  les modifications apportées au bloc par une thread ne sont plus visibles des autres threads.
- Une variable globale, un descripteur de fichier ou des variables statiques (en C)
  L'instance de la variable persiste d'une région parallèle à l'autre (sauf si le mode dynamic est actif)
  L'instance de la variables dans la zone séquentielle est aussi celle de la thread 0

clause COPYIN

- permet de transmettre la valeur de la variable partagée à toutes les tâches

```
 program threadpriv
integer:: tid, x, OMP_GET_THREAD_NUM
common  /C1/ x
!$OMP THREADPRIVATE(/C1/)
!$OMP PARALLEL PRIVATE(tid)
tid = OMP_GET_THREAD_NUM()
x= tid*10+1
print *,"T:",tid,"dans la premiere region // x=",x
!$OMP END PARALLEL
x = 2
!$OMP PARALLEL PRIVATE(tid)
tid = OMP_GET_THREAD_NUM()
print *,"T:",tid,"dans la seconde region // x=",x
!$OMP END PARALLEL
end
```

Allocation mémoire

- L'option par défaut des compilateurs est généralement PRIVATE : variables locales allouées dans la stack => privé,
  mais certaines options permettent de changer ce défaut et il est recommandé de ne pas utiliser ces options pour OpenMP
- Une opération d'allocation ou désallocation de mémoire sur un variable privée sera locale à chaque tâche
- Si une opération d'allocation/désallocation de mémoire porte sur une variable partagée, l'opération doit être effectuée par une seule tâche .

Allocation mémoire: Taille du "stack"

- La taille de la stack est limitée , différentes variables d'environnement ou fonctions permettent d'agir sur cette taille
- La pile (stack) a une taille limite pour le shell (variable selon les machines). (ulimit –s)(ulimit –s unlimited), valeurs exprimées en ko.
  Ex: ulimit –s 65532
- OpenMP :

Variable d'environnement OMP_STACKSIZE : définit le nombre d'octets que chaque thread OpenMP peut utiliser pour sa stack privée

Quelques précisions

Les variables privatisées dans une région parallèle ne peuvent être re-privatisées dans une construction parallèle interne à cette région.

En fortran: les pointeurs et les tableaux ALLOCATABLE peuvent être PRIVATE ou SHARED

mais pas LASTPRIVATE ni FIRSTPRIVATE.

Quelques précisions

Quand un bloc common est listé dans un bloc PRIVATE, FIRSTPRIVATE ou LASTPRIVATE, les éléments le constituant ne peuvent pas apparaître dans d'autres clauses de portée de variables. Par contre, si un élément d'un bloc common SHARED est privatisé, il n'est plus stocké avec le bloc common

Un pointeur privé dans une région parallèle sera ou deviendra forcément indéfini à la sortie de la région parallèle

OpenMP

- Introduction
- Structure d'OpenMP
- Portée des données
    - Constructions de partage du travail
- Construction task
- Synchronisation
- Performances
- Conclusion

Partage du travail

- Répartition d'une boucle entre les threads (boucle //)
- Répartition de plusieurs sections de code entre les threads, une section de code par thread (sections //)
- Exécution d'une portion de code par un seul thread
- Exécution de plusieurs occurences d'une même procédure par différents threads
- Exécution par différents threads de différentes unités de travail provenant de constructions f95

Portée d'une région parallèle

La portée d'une région parallèle s'étend

- au code contenu lexicalement dans cette région (étendue statique)
- au code des sous programmes appelés

L'union des deux représente l'étendue dynamique

```
Program portee
implicit none
!$OMP PARALLEL
        call sub()
!$OMP END PARALLEL
End program portee
```

```
Subroutine sub()
Logical :: p, OMP_IN_PARALLEL
!$ p = OMP_IN_PARALLEL()
print *, "Parallel prog ? ", p
End subroutine sub
```


Partage du travail

• Directives permettant de contrôler la répartition du travail, des données et la synchronisation des tâches au sein d'une région parallèle :

- DO
- SECTIONS
- SINGLE
- MASTER
- WORKSHARE

Partage du travail : boucle parallèle

Directive DO (for en C)

parallélisme par répartition des itérations d'une boucle.

- Le mode de répartition des itérations peut être spécifié dans la clause SCHEDULE (codé dans le programme ou grâce à une variable d'environnement)
- Une synchronisation globale est effectuée en fin de construction END DO (sauf si NOWAIT)
- Possibilité d'avoir plusieurs constructions DO dans une région parallèle.
- Les indices de boucles sont entiers et privés
- Les boucles infinies et *do while* ne sont pas parallélisables

Directives DO et PARALLEL DO

```
Program loop
 implicit none
 integer, parameter :: n=1024
 integer          :: i, j
 real, dimension(n, n) :: tab
 !$OMP PARALLEL
      ...              ! Code répliqué
  !$OMP DO
    do j=1, n     ! Boucle partagée
     do i=1, n    ! Boucle répliquée
                       tab(i, j) = i*j
     end do
    end do
  !$OMP END DO
 !$OMP END PARALLEL
end program loop
```

```
Program parallelloop
 implicit none
 integer, parameter :: n=1024
 integer          :: i, j
 real, dimension(n, n) :: tab
 !$OMP PARALLEL DO
   do j=1 n       ! Boucle partagée
     do i=1, n    ! Boucle répliquée
       tab(i, j) = i*j
     end do
   end do
 !$OMP END PARALLEL DO
end program parallelloop
```

PARALLEL DO est une fusion des 2 directives

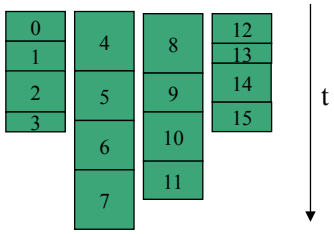
Attention : END PARALLEL DO inclut une barrière de synchronisation

Répartition du travail : clause SCHEDULE

!$OMP DO SCHEDULE(STATIC, taille_paquets)

Avec par défaut taille_paquets=nbre_itérations/nbre_thread

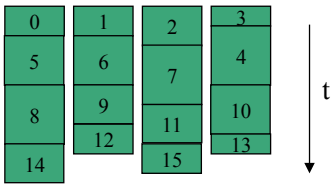
Ex : 16 itérations (0 à 15), 4 threads : la taille des paquets par défaut est de 4



!$OMP DO SCHEDULE(DYNAMIC, taille_paquets)

Les paquets sont distribués aux threads libres de façon dynamique

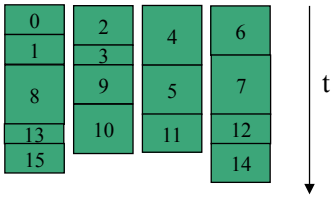
Tous les paquets ont la même taille sauf éventuellement le dernier, par défaut la taille des paquet est 1.



!$OMP DO SCHEDULE(GUIDED, taille_paquets)

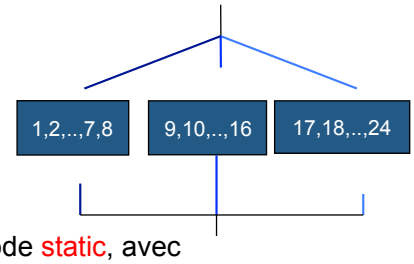
Taille_paquets : taille minimale des paquets (1 par défaut) sauf le dernier.

Taille des paquets maximale en début de boucle (ici 2) puis diminue pour équilibrer la charge.

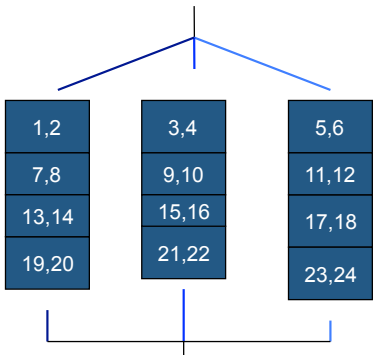


Répartition du travail : clause SCHEDULE

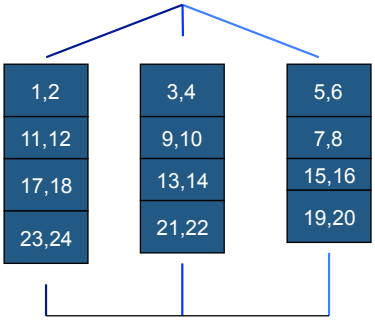
Ex: 24 itérations, 3 threads



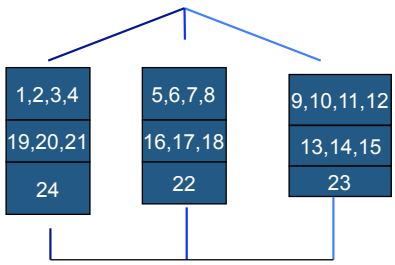
Mode static, avec
Taille paquets=nb itérations/nb threads



Cyclique : STATIC



Glouton : DYNAMIC



Glouton : GUIDED

Répartition du travail : clause SCHEDULE

Le choix du mode de répartition peut être différé à l'exécution du code avec SCHEDULE(RUNTIME)

Prise en compte de la variable d'environnement OMP_SCHEDULE

- Ex

  export OMP_SCHEDULE="DYNAMIC,400"

Reduction : pourquoi ?

```
Ex séquentiel :
      Do i=1,N
         X=X+a(i)
      enddo
```

```
En parallele :
!$OMP PARALLEL DO SHARED(X)
      do i=1,N
         X = X + a(i)
      enddo
!$OMP END PARALLEL DO
```

Reduction

opération associative appliquée à des variables scalaires partagées

- Chaque tâche calcule un résultat partiel indépendamment des autres. Les réductions intermédiaires sur chaque thread sont visibles en local.
- Puis les tâches se synchronisent pour mettre à jour le résultat final dans une variable globale, en appliquant le même opérateur aux résultats partiels.
- Attention, pas de garantie de résultats identiques d'une exécution à l'autre, les valeurs intermédiaires peuvent être combinées dans un ordre aléatoire

Reduction

Ex : !$ OMP DO REDUCTION(op:list) (op est un opérateur ou une fonction intrinsèque)

Les variables de la liste doivent être partagées dans la zone englobant la directive!;
Une copie locale de chaque variable de la liste est attribuée à chaque thread et initialisée selon l'opération (par ex 0 pour +, 1 pour *)

La clause s'appliquera aux variables de la liste si les instructions sont d'un des types suivants :

x = x opérateur expr
x = expr opérateur x
x = intrinsic (x, expr)
x = intrinsic (expr,x)

x est une variable scalaire,
expr est une expression scalaire ne référençant pas x
intrinsic = MAX, MIN, IAND, IOR, IEOR
opérateur = +,*, .AND., .OR., .EQV., .NEQV.

En V2, la REDUCTION peut être appliquée à des tableaux

```
program reduction
implicit none
integer, parameter :: n=5
integer :: i, s=0, p=1, r=1

!$OMP PARALLEL
!$OMP DO REDUCTION(+:s) &
!$OMP REDUCTION(*:p,r)
    do i=1,n
        s=s+1
        p=p*2
        r=r*3
    end do
!$OMP END PARALLEL
print *,"s=",s, " , p=",p,"  ,r =",r
end program reduction
```


Clauses portant sur le statut des variables

- SHARED ou PRIVATE
- FIRSTPRIVATE
  - privatise et assigne la dernière valeur affectée avant l'entrée dans la région //
- LASTPRIVATE
  - privatise et permet de conserver, à la sortie de la construction, la valeur calculée par la tâche exécutant la dernière itération de la boucle

```
program parallel
 !$  USE OMP_LIB
 implicit none
 integer, parameter :: n=9
 integer            :: i, mytid
 integer            :: iter

 !$OMP PARALLEL PRIVATE (mytid)
   !$OMP DO LASTPRIVATE(iter)
      do i=1, n
         iter = i
      end do
   !$OMP END DO
   mytid = OMP_GET_THREAD_NUM()
   print *, "mytid:", mytid, ";iter=",iter
 !$OMP END PARALLEL
end program parallel
```

Exécution ordonnée : ORDERED

Exécuter une zone séquentiellement
- Pour du débogage
- Pour des IOs ordonnées

Clause et Directive :ORDERED

l'ordre d'exécution des instructions de la zone encadrée par la directive sera identique à celui d'une exécution séquentielle, càd dans l'ordre des itérations

```
program parallel
  implicit none
  integer, parameter :: n=9
  integer                :: i,rang
  integer   :: OMP_GET_THREAD_NUM
!$OMP PARALLEL DEFAULT (PRIVATE)
 rang = OMP_GET_THREAD_NUM()
!$OMP DO SCHEDULE(RUNTIME) ORDERED
       do i=1, n
            !$OMP  ORDERED
            print *, "Rang:", rang, ";itération ",i
            !$OMP END ORDERED
        end do
  !$OMP END DO NOWAIT
  !$OMP END PARALLEL
end program parallel
```


Dépliage de boucles imbriquées : directive COLLAPSE

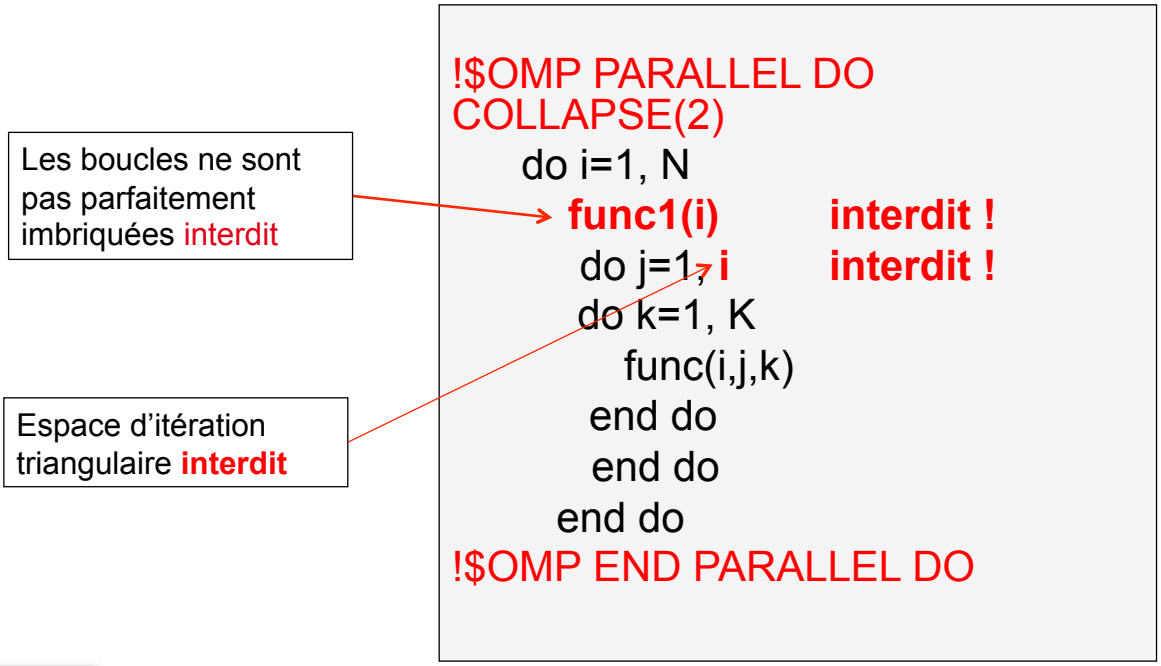
La Clause COLLAPSE(N) permet de spécifier un nombre de boucle à déplier pour créer un large espace des itérations

Les boucles doivent être parfaitement imbriquées

Ex. : Si les boucles en i et j peuvent être parallélisées, et si N et M sont petits, on peut ainsi paralléliser sur l'ensemble du travail correspondant aux 2 boucles

```
!$OMP PARALLEL DO COLLAPSE(2)
    do i=1, N
      do j=1, M
        do k=1, K
          func(i,j,k)
        end do
      end do
    end do
 !$OMP END PARALLEL DO
```

Dépliage de boucles imbriquées : directive COLLAPSE



Parallélisme imbriqué

Autoriser le parallélisme imbriqué

- Via une variable d'environnement ;

  Export OMP_NESTED= TRUE

- Via la routine OMP_SET NESTED()

```
!$  use OMP_LIB
    call OMP_SET_NESTED(.TRUE.)
```

```
#include <omp.h>
        omp_set_nested(1)
```

Ex d'application : parallélisation des nids de boucle, avec un découpage par blocs.

Parallélisme imbriqué

```
!$OMP PARALLEL SHARED(n, a, b)
!$OMP DO
   do j=0,n
      a[j] = j + 1;
      !$OMP PARALLEL DO
      do i=0,n
        b[i][j] = a[j]
      end do
      !$OMP END PARALLEL DO
   end do
!$OMP END DO
!$OMP END PARALLEL
```

```
!$OMP PARALLEL SHARED(n, a, b)
!$OMP DO
   do j=0, n
      a[j] = j + 1;
      !$OMP DO
      do i=0,n
        b[i][j] = a[j]
      end do
      !$OMP END DO
   end do

!$OMP END DO
!$OMP END PARALLEL
```

L'imbrication de directives de partage du travail n'est pas valide si on ne crée pas une nouvelle région parallèle.

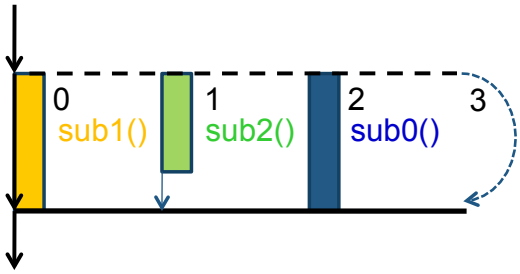
Partage du travail : SECTIONS parallèles

But : Répartir l'exécution de plusieurs portions de code indépendantes sur différentes tâches

Une section: une portion de code exécutée par une et une seule tâche

Directive SECTION au sein d'une construction SECTIONS

```
program section
call OMP_SET_NUM_THREADS(4)
  !$OMP PARALLEL SECTIONS
  !$OMP SECTION
     call sub0()
  !$OMP SECTION
    call sub1()
  !$OMP SECTION
    call sub2()
  !$OMP END PARALLEL SECTIONS
```



SECTIONS parallèles

• Les directives SECTION doivent se trouver dans l'étendue lexicale de la construction

• **Les clauses admises :**
  - PRIVATE, FIRSTPRIVATE, LASTPRIVATE, REDUCTION

LASTPRIVATE : valeur donnée par le thread qui exécute la dernière section

• END PARALLEL SECTIONS inclut une barrière de synchronisation (NOWAIT interdit)

Ex : SECTIONS + REDUCTION

```
program section
Logical  b
!$OMP PARALLEL SECTIONS REDUCTION(.AND. : b)
  !$OMP SECTION
    b = b .AND. func0()
  !$OMP SECTION
       b = b .AND. func1()
  !$OMP SECTION
       b = b .AND. func2()
  !$OMP END PARALLEL SECTIONS
IF (b) THEN
          print(*, "All of the functions succeeded")
ENDIF
```

Directive WORKSHARE

• Destinée à permettre la parallélisation d'instructions Fortran 95 intrinsèquement parallèles comme :
  - Les notations tableaux.
  - Certaines fonctions intrinsèques
  - Instruction FORALL, WHERE

• Cette directive s'applique à des variables partagées, dans l'extension lexicale de la construction WORKSHARE.
• Tout branchement vers l'extérieur est illégal.
• Clause possibles : PRIVATE, FIRSTPRIVATE, COPYPRIVATE, NOWAIT

• Fusion PARALLEL WORKSHARE possible

```
Program workshare
…
!$OMP PARALLEL
!$OMP WORKSHARE
A(:) = B(:)
somme = somme + SUM(A)
FORALL (I=1:M)  A(I)=2*B(I)
!$OMP END WORKSHARE
!$OMP END PARALLEL
…
End program
```


Partage du travail : exécution exclusive

Construction SINGLE

• Exécution d'une portion de code par une et une seule tâche (en général, la première qui arrive sur la construction).
• clause NOWAIT : permet de ne pas bloquer les autres tâches qui par défaut attendent sa terminaison.
• Clauses admises : PRIVATE, FIRSTPRIVATE,
• COPYPRIVATE(var): mise à jour des copies privées de var sur toutes les tâches (après END SINGLE)

```
!$OMP SINGLE [clause [clause ...] ]
    ...
!$OMP END SINGLE
```

Partage du travail : exécution exclusive

Construction MASTER

- Exécution d'une portion de code par la tâche maître seule
- Pas de synchronisation, ni en début ni en fin (contrairement à SINGLE)
  - Attention aux mises à jour de variables qui seraient utilisées par d'autres threads
- Pas de clause

```
!$OMP MASTER
    ...
!$OMP END MASTER
```


Partage du travail : procédures orphelines

- Procédures appelées dans une région // et contenant des directives OpenMP
  Elles sont dites orphelines
- Attention : le contexte d'exécution peut être # selon le mode de compilation (-fopenmp ou pas) des unités de programme appelantes et appelées
- Attention au statut des variables locales aux subroutines

```
Program main
  implicit none
  integer, parameter :: n=1025
  real, dimension(n,n) :: a
  real, dimension(n) :: x,y

  call random_number(a)
  call random_number(x); y(:)=0
  !$OMP PARALLEL
    call orphan(a,x,y,n)
  !$OMP END PARALLEL
End program main
```

```
Subroutine orphan(a,x,y,n)
implicit none
Integer, intent(in) :: n
real, intent(in), dimension(n,n) :: a
real, intent(in), dimension(n) :: x
real, intent(out), dimension(n) :: y

!$OMP DO
  DO i=1,n
    y(i) = SUM(a(i, : ) * x(:) )
  END DO
!$OMP END DO
End subroutine orphan
```

OpenMP

- Introduction
- Structure d'OpenMP
- Portée des données
- Constructions de partage du travail
  - Construction TASK
- Synchronisation
- Performances
- Conclusion

Partage du travail : construction TASK

Une "TASK" au sens OpenMP est une unité de travail dont l'exécution peut être différée (ou démarrer immédiatement)

Autorise la génération dynamique de tâches

Permet de paralléliser des problèmes irréguliers

- boucles non bornées
- algos récursifs
- schémas producteur/consommateur

Une TASK est composée

- d'un code à exécuter
- d'un environnement de données associé

Partage du travail : construction TASK

• La construction TASK définit explicitement une TASK OpenMP
• SI un thread rencontre une construction TASK, une nouvelle instance de la TASK est créée (paquet code + données associées)
• Le thread peut soit exécuter la tâche, soit différer son exécution. La tâche pourra être attribuée à n'importe quel thread de l'équipe.

```
!$OMP PARALLEL
 call sub()
!$OMP TASK
        …
!$OMP END TASK
        …
!$OMP END PARALLEL
```

Clauses
IF, DEFAULT(PRIVATE|SHARED)
PRIVATE, SHARED
FIRSTPRIVATE
UNTIED …

Partage du travail : construction TASK

Par défaut les variables de TASKs orphelines sont firstprivate
Sinon les variables sont firstprivate à moins qu'elles héritent d'un attribut shared du contexte qui les englobe

```
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;
}
```

Ici n est firstprivate

Partage du travail : construction TASK

Remarque : le concept existait avant la construction

Un thread qui rencontre une construction PARALLEL

- crée un ensemble de TASKs implicites (paquet code + données)
- Crée une équipe de threads
- Les TASKs implicites sont liées (*tied*) aux threads, une pour chaque thread de l'équipe

• **A quel moment la TASK est-elle exécutée ?**

- L'exécution d'une TASK générée peut être affectée, par le scheduler, à un thread de l'équipe qui a terminé son travail.
- La norme 3.0 impose des règles sur l'exécution des TASK
- Ex :Les TASK en attente dans la région //, doivent toutes être exécutées par les threads de l'équipe qui rencontrent
- Une BARRIER (implicite ou explicite)
- Une directive TASKWAIT

OpenMP

- Introduction
- Structure d'OpenMP
- Portée des données
- Constructions de partage du travail
- Construction task
  - Synchronisation
- Performances
- Conclusion

Synchronisation

- Synchronisation de toutes les tâches sur un même niveau d'instruction (barrière globale)
- Ordonnancement de tâches concurrentes pour la cohérence de variables partagées (exclusion mutuelle)
- Synchronisation de plusieurs tâches parmi un ensemble (mécanisme de verrou)

Synchronisation : barrière globale

Par défaut à la fin des constructions parallèles, en l'absence du NOWAIT

Directive BARRIER

- Impose explicitement une barrière de synchronisation: chaque tâche attend la fin de toutes les autres

```
Program barriere
  !$OMP PARALLEL &
  !$OMP  SHARED(A,B,C)  PRIVATE(tid)
     tid = OMP_GET_THREAD_NUM()
     A(tid) = big_calcul_1(tid);
  !$OMP BARRIER
  !$OMP DO
        DO i=1, n
           C(i) = big_calcul_2(i, A)
        ENDDO
  !$OMP END DO
  !$OMP DO
        DO i=1, n
           B(i) = big_calcul_3(i, C)
        END DO
  !$OMP END DO NOWAIT
     A(tid) = big_calcul_4(tid)
  !$OMP END PARALLEL
End program barriere
```

Barrière explicite

Barrière implicite

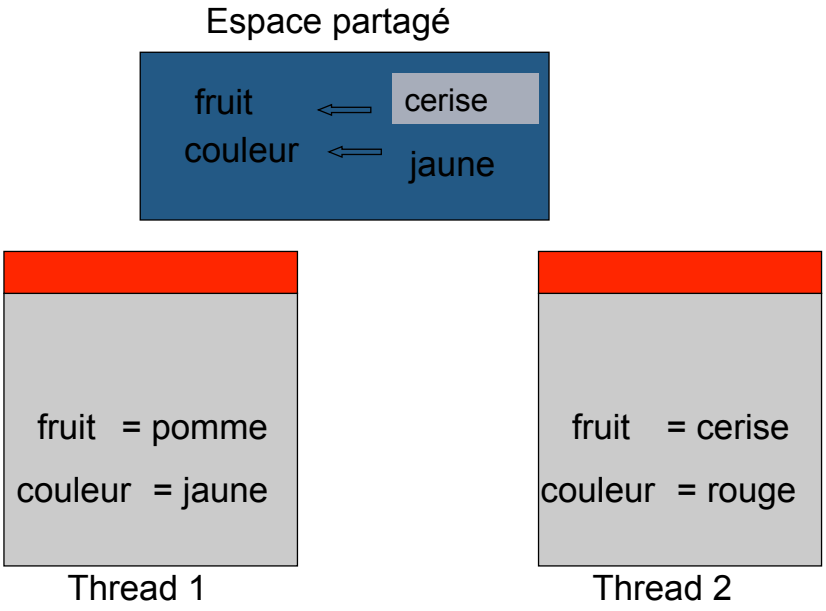
Pas de barrière

Synchronisation

Il peut être nécessaire d'introduire une synchronisation entre tâches concurrentes pour éviter que celles-ci modifient la valeur d'une variable dans un ordre quelconque

Ex :

2 threads ont un espace de mémoire partagé



Synchronisation : régions critiques

Directive CRITICAL

- Elle s'applique sur une portion de code
- Les tâches exécutent la *région critique* dans un ordre non-déterministe, une à la fois
- Garantit aux threads un accès en *exclusion mutuelle*
- Son étendue est dynamique

```
Integer,dimension(n) :: a=1
somme = 0
DO j=1,n
        somme = somme + a(i)
END DO
…
```

```
Integer somme = 0
Integre dimension(n) :: a=1
!$OMP PARALLEL DEFAULT(SHARED)  &
!$OMP PRIVATE(i,,j,somme_partielle)
somme_partielle = 0
!$OMP DO
DO j=1,n
        somme_partielle = somme_partielle + a(i)
END DO
!$OMP END DO
!$OMP CRITICAL
    somme = somme + somme_partielle
!$OMP END CRITICAL
!$OMP END PARALLEL
….
```

Synchronisation : mise à jour atomique

La directive ATOMIC s'applique seulement dans le cadre de la mise à jour d'un emplacement mémoire

Une variable partagée est lue ou modifiée en mémoire par un seul thread à la fois

Agit sur l'instruction qui suit immédiatement si elle est de la forme :

- **x = x (op) exp**
- **ou x = exp (op) x**
- **ou x = f(x,exp)**
- **ou x = f(exp,x)**

op : +, -, *, /, .AND., .OR., .EQV., .NEQV.

f : MAX, MIN, IAND, IOR, IEOR

x est une variable scalaire

```
Program atomic
  implicit none
  integer :: count, rang
  integer :: OMP_GET_THREAD_NUM
!$OMP PARALLEL PRIVATE(rang)
!$OMP ATOMIC
    count = count + 1
    rang=OMP_GET_THREAD_NUM()
    print *, “rang : ”, rang, “count:”, count
!$OMP END PARALLEL
print *, “count:”, count
End program atomic
```

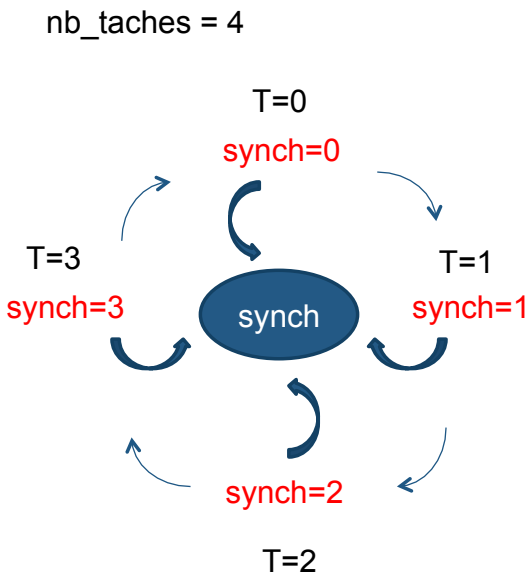

directive FLUSH

Les valeurs des variables partagées peuvent rester temporairement dans des registres pour des raisons de performances

La directive FLUSH garantit que chaque thread a accès aux valeurs des variables partagées modifiées par les autres threads.

```
Program anneau
  implicit none
  integer :: rang, nb_taches, synch=0
  integer :: OMP_GET_NUM_THREADS
  integer :: OMP_GET_THREAD_NUM
!$OMP PARALLEL PRIVATE(rang,nb_taches)
   rang=OMP_GET_THREAD_NUM()
   nb_taches=OMP_GET_NUM_THREADS()
      if (rang == 0) then ; do
            !$OMP FLUSH(synch)
            if (synch == nb_taches-1) exit
      end do
      else ; do
            !$OMP FLUSH(synch)
            if (synch == rang-1) exit
      end do
      end if
   print *, “Rang: “,rang,”synch = “,synch
   synch=rang
   !OMP FLUSH(synch)
!$OMP END PARALLEL
end program anneau
```

directive FLUSH




```
Program anneau
  implicit none
  integer :: rang, nb_taches, synch=0
  integer :: OMP_GET_NUM_THREADS
  integer :: OMP_GET_THREAD_NUM
!$OMP PARALLEL PRIVATE(rang,nb_taches)
   rang=OMP_GET_THREAD_NUM()
   nb_taches=OMP_GET_NUM_THREADS()
      if (rang == 0) then ; do
            !$OMP FLUSH(synch)
            if (synch == nb_taches-1) exit
      end do
      else ; do
            !$OMP FLUSH(synch)
            if (synch == rang-1) exit
      end do
      end if
   print *, "Rang: ",rang,"synch = ",synch
   synch = rang
   !OMP FLUSH(synch)
!$OMP END PARALLEL
end program anneau
```


directive FLUSH

- **FLUSH implicite dans certaines régions**
  - ✓Au niveau d'une BARRIER
  - ✓A l'entrée et à la sortie d'une région PARALLEL, CRITICAL, ou ORDERED, et d'une région PARALLEL de partage du travail
  - ✓A l'appel des fonctions de « lock »
  - ✓Immédiatement avant ou après chaque point d'ordonnancement de TASK
  - ✓A l'entrée et à la sortie de régions ATOMIC (s'applique sur les variables mise à jour par ATOMIC)

- **Pas de FLUSH implicite**
  - ✓A l'entrée d'une région de partage de travail
  - ✓A l'entrée ou la sortie d'une région MASTER

Performances et partage du travail

Minimiser le nombre de régions parallèles

Eviter de sortir d'un région parallèle pour la recréer immédiatement

```
!$OMP PARALLEL
   ...
   !$OMP DO
      do i=1, N
         func1(i)
      end do
   !$OMP END DO
!$OMP END PARALLEL
!$OMP PARALLEL
   !$OMP DO
      do j=1, M
         func2(j)
      end do
   !$OMP END DO
!$OMP END PARALLEL
```

Surcoût inutile

Ici, une construction PARALLEL suffit

```
!$OMP PARALLEL
   ...
   !$OMP DO
      do i=1, N
         func1(i)
      end do
   !$OMP END DO
   !$OMP DO
      do j=1, M
         func2(j)
      end do
   !$OMP END DO
!$OMP ENDPARALLEL
```


Performances et partage du travail

- Introduire un PARALLEL DO dans les boucles capables d'exécuter des itérations en //
- Si il y a des dépendances entre itérations, essayer de les supprimer en modifiant l'algorithme
- S'il reste des itérations dépendantes, introduire des constructions CRITICAL autour des variables concernées par les dépendances.
- Si possible, regrouper dans une unique région parallèle plusieurs structures DO.
- Dans la mesure du possible, paralléliser la boucle la plus externe
- Adapter le nombre de tâches à la taille du problème à traiter afin de minimiser les surcoûts de gestion des tâches par le système
- Utiliser SCHEDULE(RUNTIME) si besoin
- ATOMIC REDUCTION+ performant que CRITICAL

merci

LIEU
LOCALISATION

www.nomdedomaine.com