Calcul – Cours 4: OpenMP – Parallélisation multitâches pour machines à mémoire partagée

Jonathan Rouzaud-Cornabas

LIRIS / Insa de Lyon – Inria Beagle

Cours inspiré de ceux de Frédéric Desprez (DR Inria – Grenoble)

Références

- Cours OpenMP, F. Roch (Grenoble)
- Cours OpenMP, J. Chergui & P.-F. Lavallee (IDRIS)

- `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

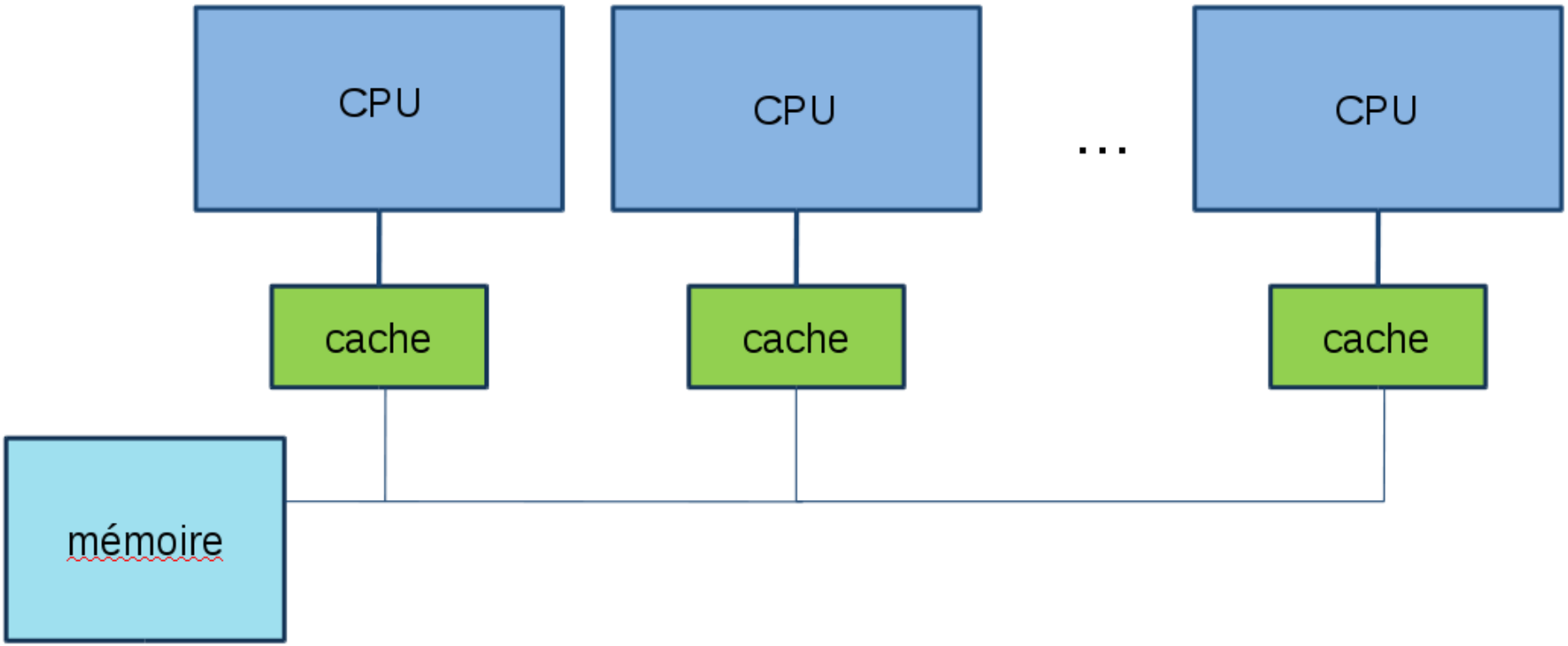
Un peu d'histoire

- Parallélisation multitâches existe depuis longtemps mais bibliothéques / langages spécifiques
- Augmentation des multi-core (machines à mémoire partagée) $\rightarrow$ nécessité d'un standart
- OpenMP-1 (1997), OpenMP-2 (2000), OpenMP-3 (2008), OpenMP-4.0 (2013), OpenMP-4.1 (2015), OpenMP-5.0 (2017)
- Support GPU dans OpenMP ($\approx$ 10% de perte)

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.
- Exemple: les processeurs multicoeurs généralistes partagent une mémoire commune les tâches peuvent être attribuées à des "cores" distincts

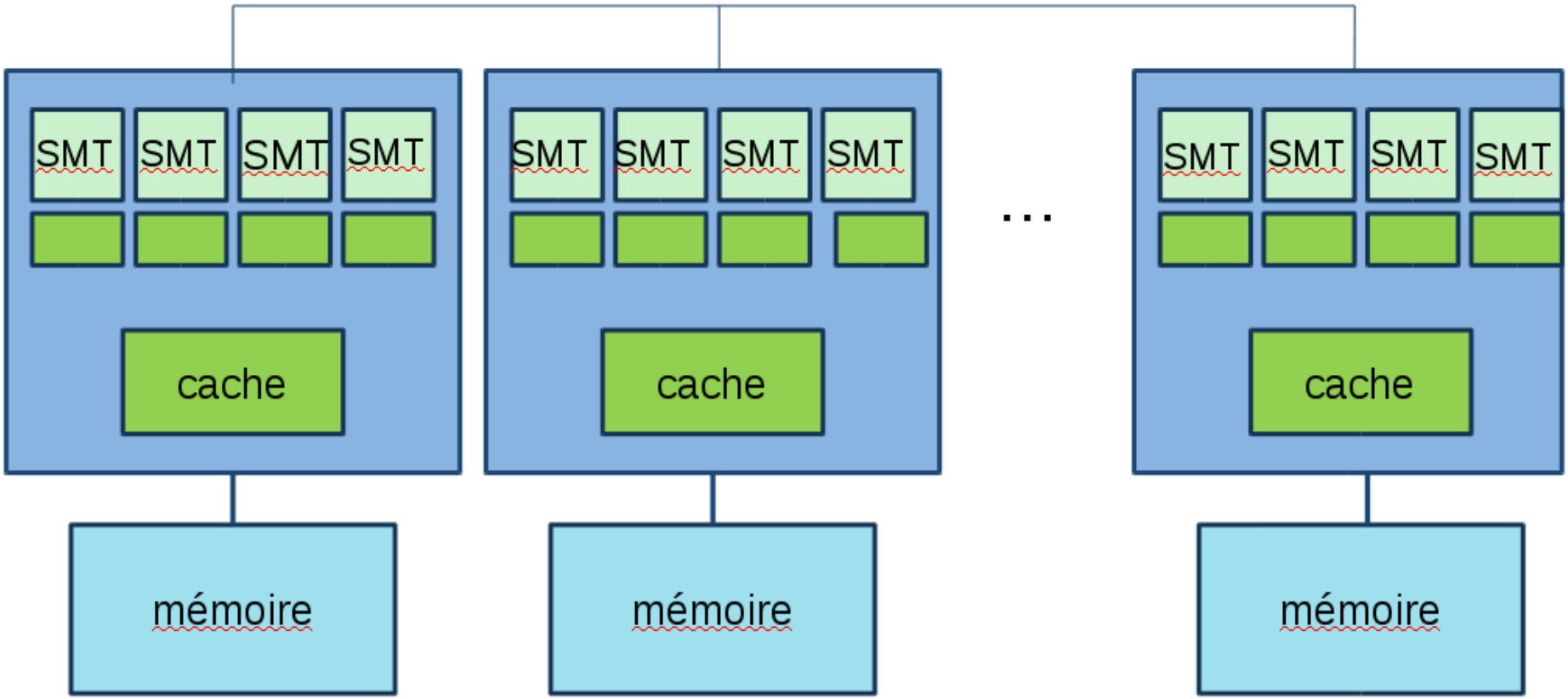
Programmation multi-tâches sur les architectures UMA



La mémoire est commune

- Architectures à accès mémoire uniforme (UMA)
- Un problème inhérent : les contentions mémoire

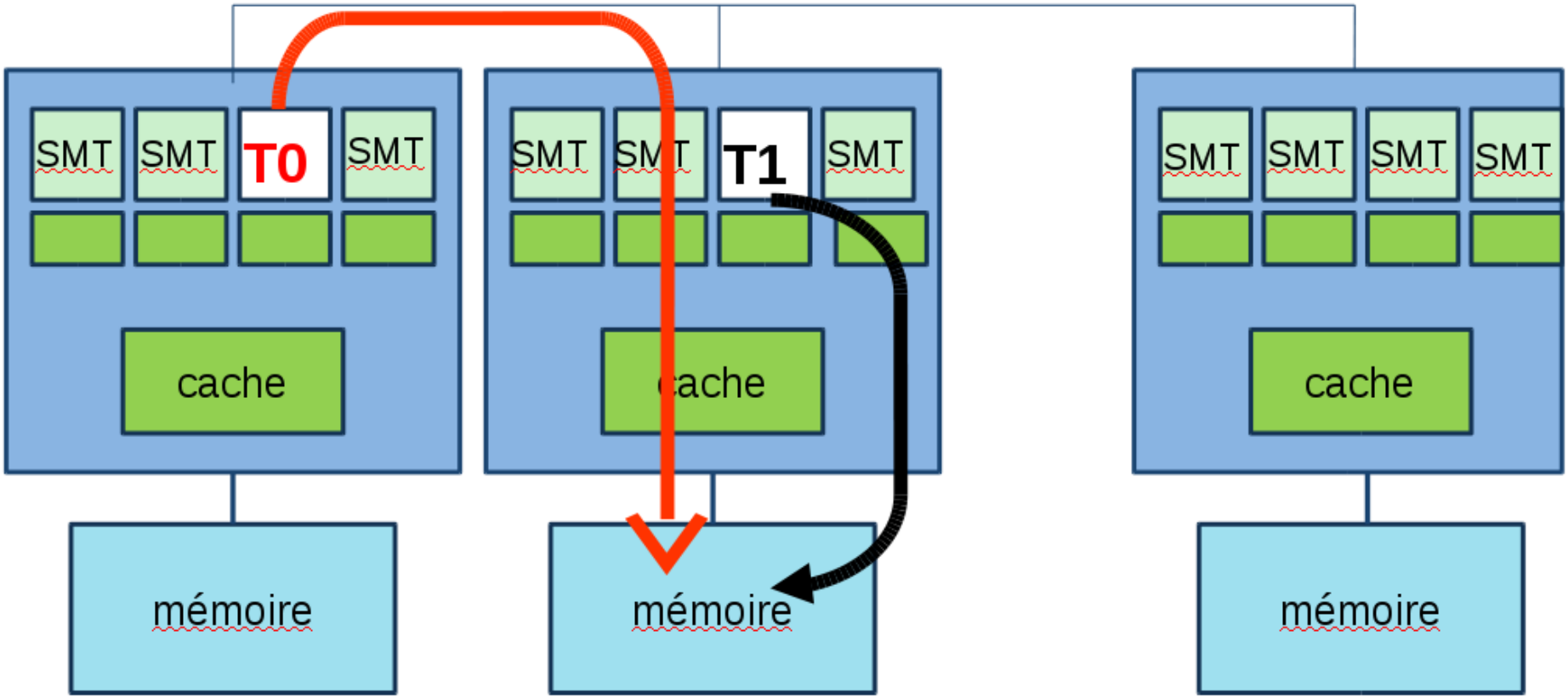
Programmation multi-tâches sur les architectures NUMA



La mémoire est directement attachée aux puces multicoeurs

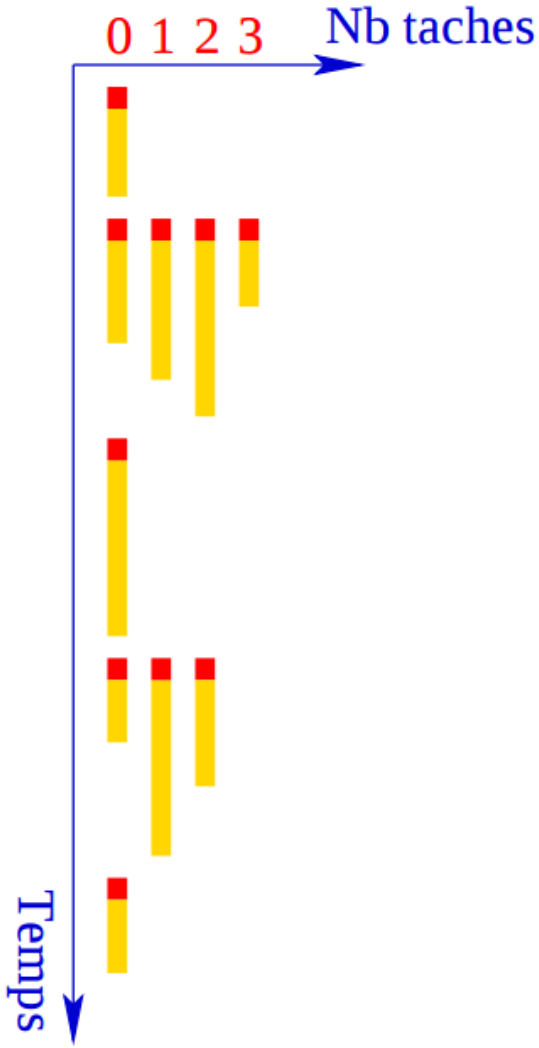
- Des architectures à accès mémoire non uniforme NUMA

Programmation multi-tâches sur les architectures NUMA



Concepts Généreaux

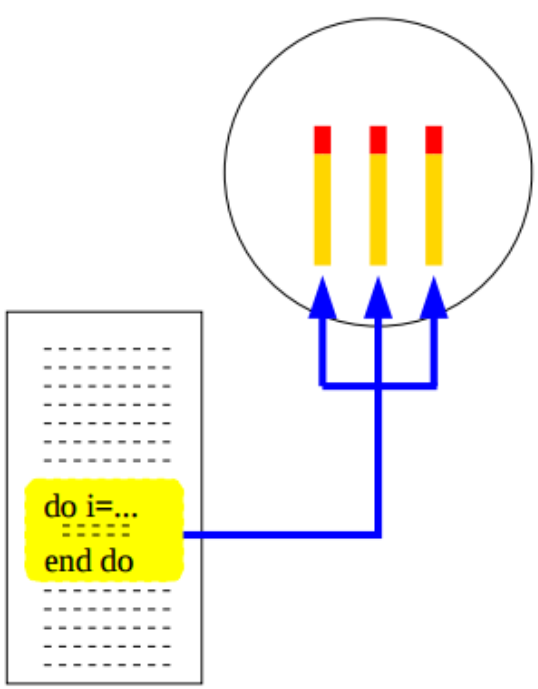
- Un programme OpenMP = Un processus avec des threads
- Conséquence: Variable partagée ou non
- Composition de partie séquentielle (toujours sur le thread 0) et paralléle (plusieurs threads)



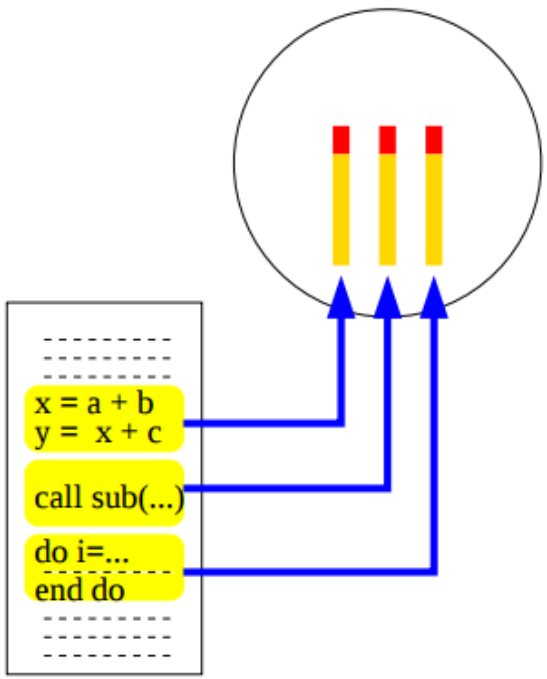
Type de Parallélisation

Le partage du travail consiste essentiellement à :

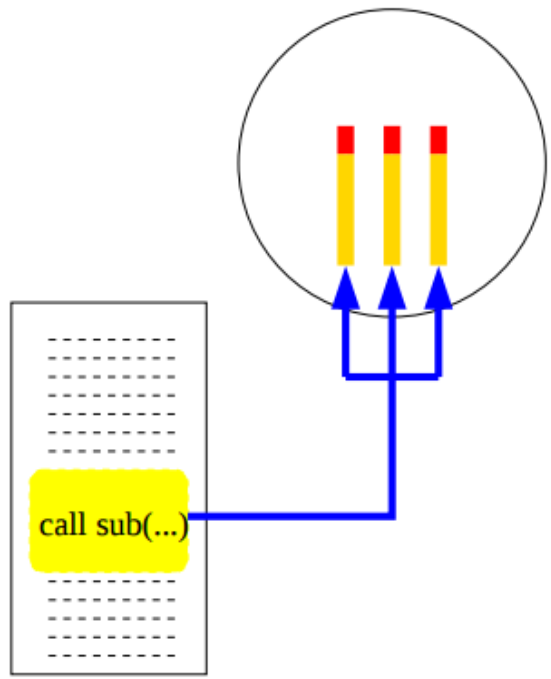
- exécuter une boucle par répartition des itérations entre les tâches
- exécuter plusieurs sections de code mais une seule par tâche
- exécuter plusieurs occurrences d'une même procédure par différentes tâches (orphaning)



Boucle parallèle



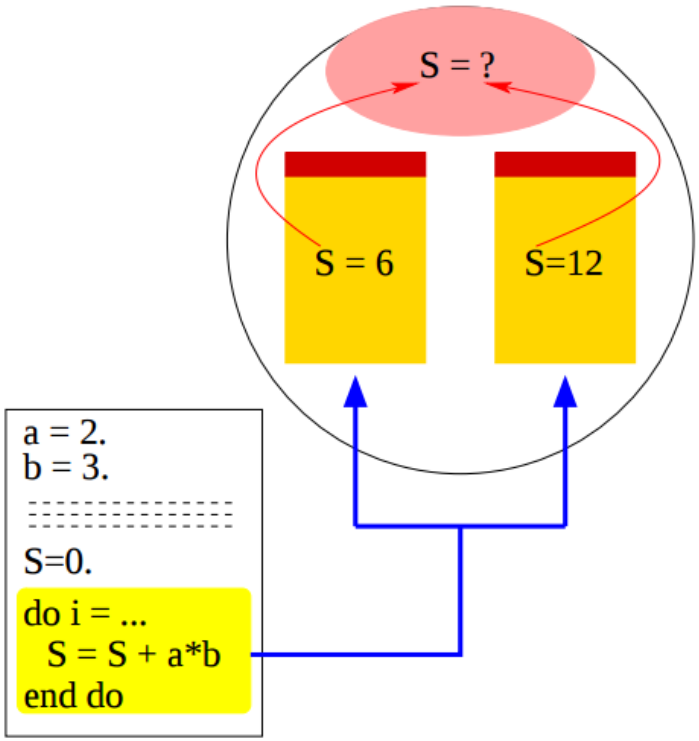
Sections parallèles



Procédure parallèle (orphaning)

Synchronisation

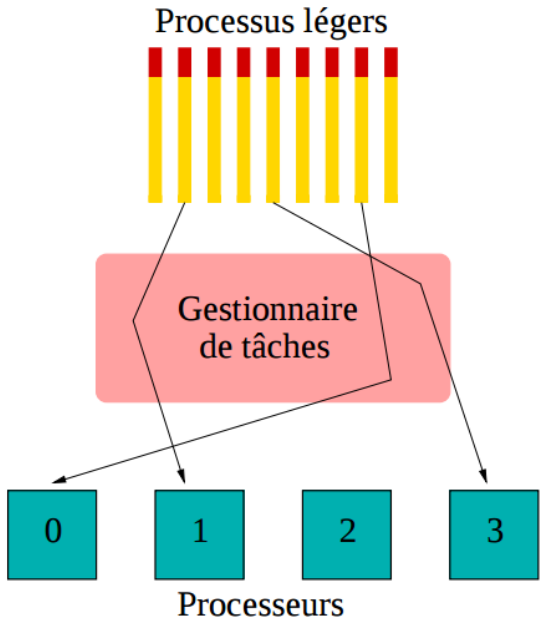
- Il est parfois nécessaire d'introduire une synchronisation entre les tâches concurrentes pour éviter, par exemple, que celles-ci modifient dans un ordre quelconque la valeur d'une même variable partagée (cas des opérations de réduction).



Répartition des tâches

Les tâches sont affectées aux processeurs par le système d'exploitation. Différents cas peuvent se produire :

- au mieux, à chaque instant, il existe une tâche par processeur avec autant de tâches que de processeurs dédiés pendant toute la durée du travail
- au pire, toutes les tâches sont traitées séquentiellement par un et un seul processeur
- en réalité, pour des raisons essentiellement d'exploitation sur une machine dont les processeurs ne sont pas dédiés, la situation est en général intermédiaire.



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 VS Threads

Deux méthodes de réfléchir la répartition

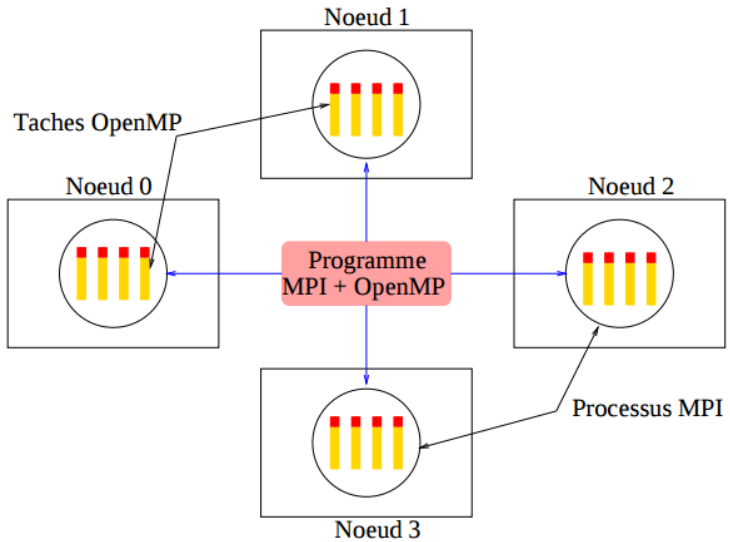
- Une tâche : un ensemble de calcul (concept commun)
- **Thread** : On définit explicitement la répartition des tâches entre les threads
- **Thread** : Code plus complexe (on doit écrire la gestion des threads)
- **OpenMP** : On ne définit pas la répartition
- **Thread** : On peut choisir l'ordre d'exécution
- **OpenMP** : Ordre d'exécution non déterministe

- **Thread** : Ordonnancement statique / à la charge du développeur
- **OpenMP** : Ordonnancement dynamique / à la charge du système

OpenMP VS MPI

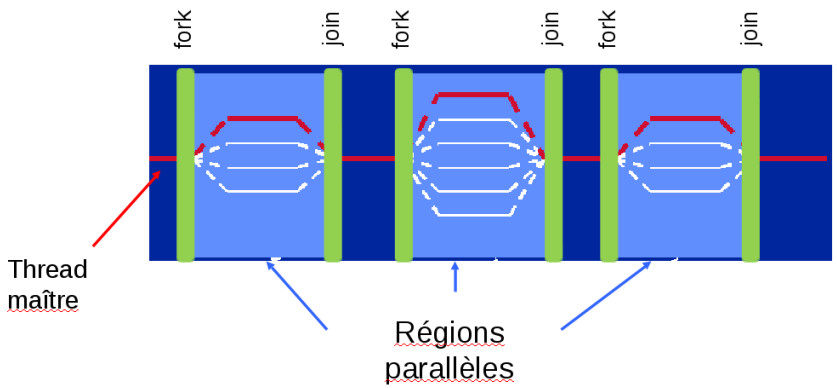
Ce sont deux modèles complémentaires de parallélisation.

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



Comment ça marche ?

- Il est à la charge du développeur d'introduire des directives OpenMP dans son code (du moins en l'absence d'outils de parallélisation automatique)
- A l'exécution du programme, le système d'exploitation construit une région parallèle sur le modèle "fork and join"
- A l'entrée d'une région parallèle, la tâche maitre crée/active (fork) des processus "fils" (processus légers) qui disparaissent/s'assoupissent en fin de région parallèle (join) pendant que la tâche maitre poursuit seule l'exécution du programme jusqu'à l'entrée de la région parallèle suivante



Format des directives/pragmas

- Sentinelle directive [clause[clause]..]

```
#pragma omp parallel private(a,b) firstprivate
{
   ...
}
```

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

Construction d'une région paralléle

```
#include <omp.h>

int main(int argc, char** argv) {
  int a,b,c;

  /* Code s quentiel ex cut par le ma tre */

  #pragma omp parallel private(a,b) \
    shared(c)
  {
      /* Zone parall le ex cut e par toutes les
         threads */
  }

  /* Code s quentiel */
}
```

Clause IF de la directive PARALLEL

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

```
#include <omp.h>

int main(int argc, char** argv) {
  int a,b,c;

  /* Code sequentiel execut par le maitre */

  #pragma omp parallel for if(para_low)
  {
        /* Zone parallele executee par toutes les
           threads ou sequentielle*/
  }

  /* Code sequentiel */
}
```

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

Include, Compilation et Exécution

- `#include <omp.h>`
- `gcc/g++ --fopenmp -o prog prog.c[pp]`
- `export OMP_NUM_THREADS=2 ; ./prog`

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
  - soit FIRSTPRIVATE, elle est dans la pile de chaque thread, sa valeur est définie à l'entrée de la zone (valeur initiale de la variable)
  - soit LASTPRIVATE, elle est dans la pile de chaque thread, sa valeur est conservé à la sortie (la valeur de la dernière tâche)

- Déclarer le statut d'une variable
  - `#pragma omp parallel PRIVATE(list)`
  - `#pragma omp parallel FIRSTPRIVATE(list)`
  - `#pragma omp parallel SHARED(list)`
- Déclarer un statut par défaut `#pragma omp default(SHARED|NONE)`

Clauses de la directive PARALLEL

- `NONE` : 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

- `THREADPRIVATE` :
  - 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

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

Allocation Mémoire

- L'option par défaut des compilateurs est généralement `PRIVATE` : variables locales allouées dans la stack $\Rightarrow$ 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 une 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

- La taille de la pile 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.
- OpenMP : Variable d'environnement OMP_STACKSIZE : définit le nombre d'octets que chaque thread OpenMP peut utiliser pour sa stack privée

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

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

Partage du travail

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

- `FOR`
- `SECTIONS`
- `SINGLE`
- `MASTER`
- `WORKSHARE` (seulement en Fortran)

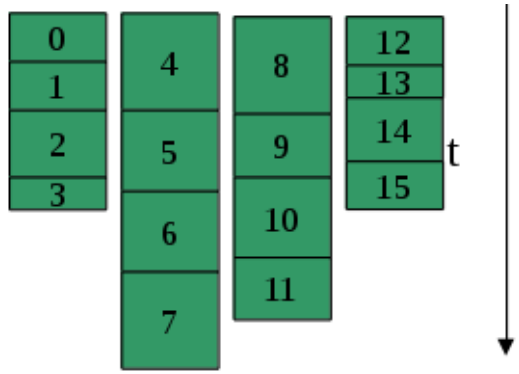
Partage du travail : boucle parallèle

Directive FOR : parallélisme par répartition des itérations d'une boucle.

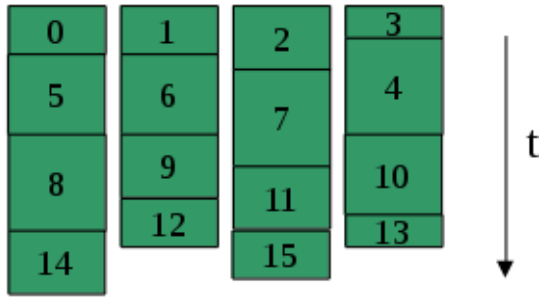
- Le mode de répartition des itérations peut être spécifié dans la clause SCHEDULE (codé dans le programme ou grâce à une variable d'environnement)
- Une synchronisation globale est effectuée en fin de construction END FOR (sauf si NOWAIT)
- Possibilité d'avoir plusieurs constructions FOR dans une région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

Répartition du travail : clause SCHEDULE

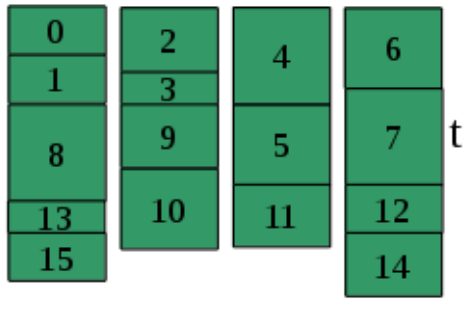
- `#pragma OMP FOR SCHEDULE(STATIC, taille_paquets)`
  - Avec par défaut *taille_paquets* = $\frac{nbre\_iterations}{nbre\_threads}$
  - Exemple : 16 itérations (0 à 15), 4 threads : la taille des paquets par défaut est de 4



- `#pragma OMP FOR 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.

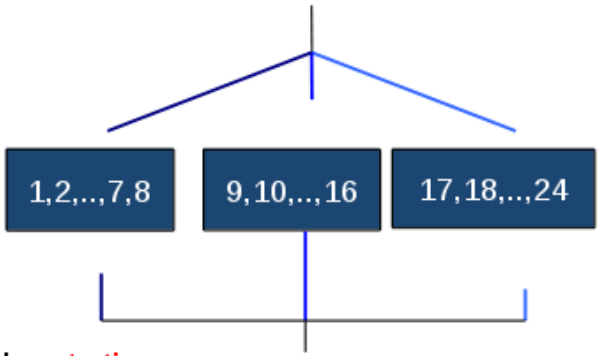


- `#pragma OMP FOR SCHEDULE(GUIDED, taille_paquets)`
  - *taille_paquets* : taille minimale des paquets (1 par défaut) sauf le dernier.
  - Taille des paquets maximale en début de boucle (ici 2) puis diminue pour équilibrer la charge.

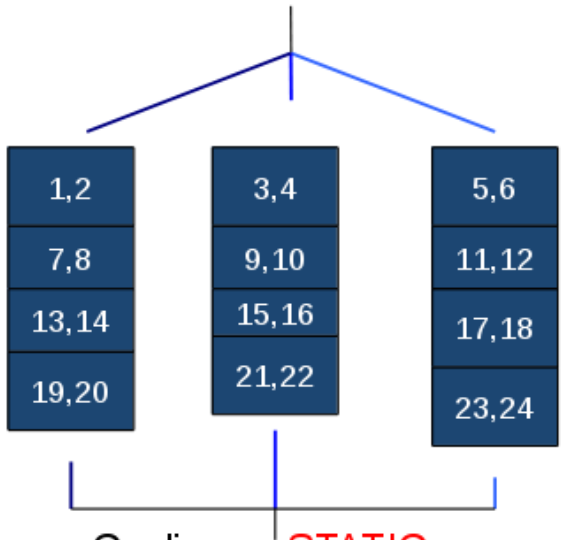


Répartition du travail : clause SCHEDULE

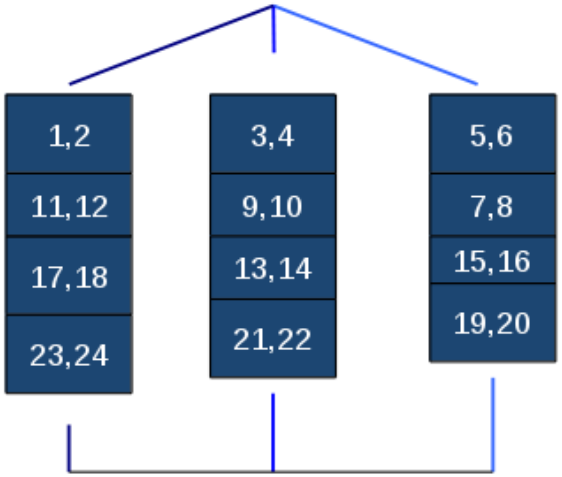
Ex: 24 itérations, 3 threads



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`
- Exemple : `export OMP_SCHEDULE=``DYNAMIC,400``

Reduction : pourquoi ?

En séquentiel

```
for (int i = 0;
      i < N; i++) {
   X=X+a(i)
}
```

En parallèle

```
#pragma omp for shared(X)
 for (int i = 0;
       i < N; i++) {
    X=X+a(i)
 }
```

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

Réduction en pratique

- Exemple : `#pragma omp for 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.

Réduction en pratique

```
 int factorial(int number)
{
  int fac = 1;
  #pragma omp parallel for reduction(*:fac)
  for(int n=2; n<=number; ++n)
    fac *= n;
  return fac;
}
```

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

```
#pragma omp for ordered \\
                schedule(dynamic)
for(int n=0; n<100; ++n)
{
  files[n].compress();

  #pragma omp ordered
  send(files[n]);
}
```

Dépliage de boucles imbriquées : directive COLLAPSE

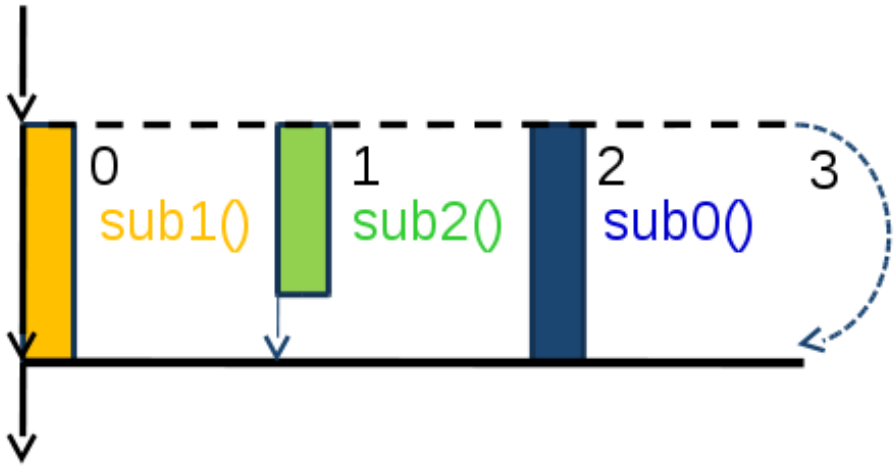
- La clause `COLLAPSE(N)` permet de spécifier un nombre de boucle à déplier pour créer un large espace des itérations
- Les boucles doivent être parfaitement imbriquées
- Exemple : Si les boucles en i et j peuvent être parallélisées, et si N et M sont petits, on peut ainsi paralléliser sur l'ensemble du travail correspondant aux 2 boucles

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

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`

```
#pragma omp sections
{
  #pragma omp section
  { sub0() }
  #pragma omp section
  { sub1() }
  #pragma omp section
  { sub2() }
}
```



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

```
#pragma omp parallel
{
   Work1();
   #pragma omp single
   {
      Work2();
   }
   Work3();
}
```

Partage du travail : exécution exclusive

Construction `SINGLE`

- 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

```
#pragma omp parallel
{
   Work1();

   #pragma omp master
   {
      Work2();
   }

   Work3();
}
```

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.

```
void increment_list_items
                (node* head)
{
  #pragma omp parallel
  {
    #pragma omp single
    {
      for(node*
        p = head; p;
        p = p->next)
        {
          #pragma omp task
          process(p);
// p: firstprivate by default
        }
    }
  }
}
```

Partage du travail : construction TASK

- **Remarque** : le concept existait avant la construction

- Un thread qui rencontre une construction `PARALLEL`
  - crée un ensemble de `TASK`s implicites (paquet code + données)
  - crée une équipe de threads
  - les `TASK`s 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 l'ordonnanceur, à un thread de l'équipe qui a terminé son travail.
  - La norme 3.0 impose des règles sur l'exécution des `TASK`
  - Exemple : 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`

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

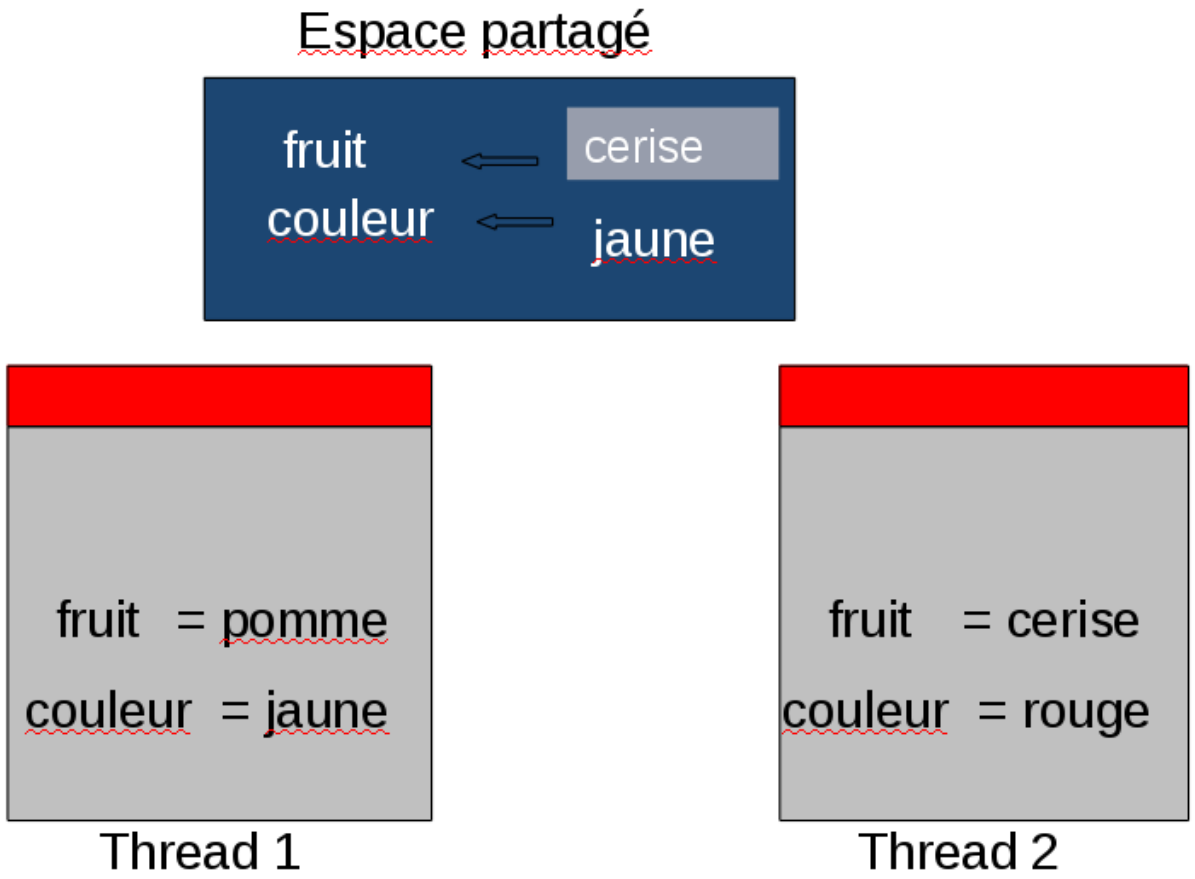
```
#pragma omp parallel
{
  /* Tous les threads executent. */
  SomeCode();
  #pragma omp barrier
  // Tous les threads executent mais
  // ils attendent tous les autres avant
  SomeMoreCode();
}

#pragma omp parallel
{
  #pragma omp for
  for(int n=0; n<10; ++n)
   Work();
  // Implicite barri re, tous les threads
  // attendent la fin du FOR
  SomeMoreCode();
}
```

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

```
int main(int argc, char** argv) {
int x;
x = 0;
#pragma omp parallel shared(x)
{
  #pragma omp critical
  x = x + 1;
}  /* end of parallel section */
}
```

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

```
#pragma omp atomic
        count = count+1;
```

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.

```
      /* presumption: int a = 0
      /* First thread */
/* Second thread */
      b = 1;
a = 1;
      #pragma omp flush(a,b)
#pragma omp flush(a,b)
      if(a == 0)
if(b == 0)
      {
{
        /* Critical section */
/* Critical section */
      }
}
```

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`

Performance et partage de travail

- Minimiser le nombre de régions parallèles
- Eviter de sortir d'une région parallèle pour la recréer immédiatement

```
#pragma omp parallel
{
 #pragma omp for
 for(int n=0; n<10; ++n)
   printf("␣%d", n);
}
printf(".\n");
```

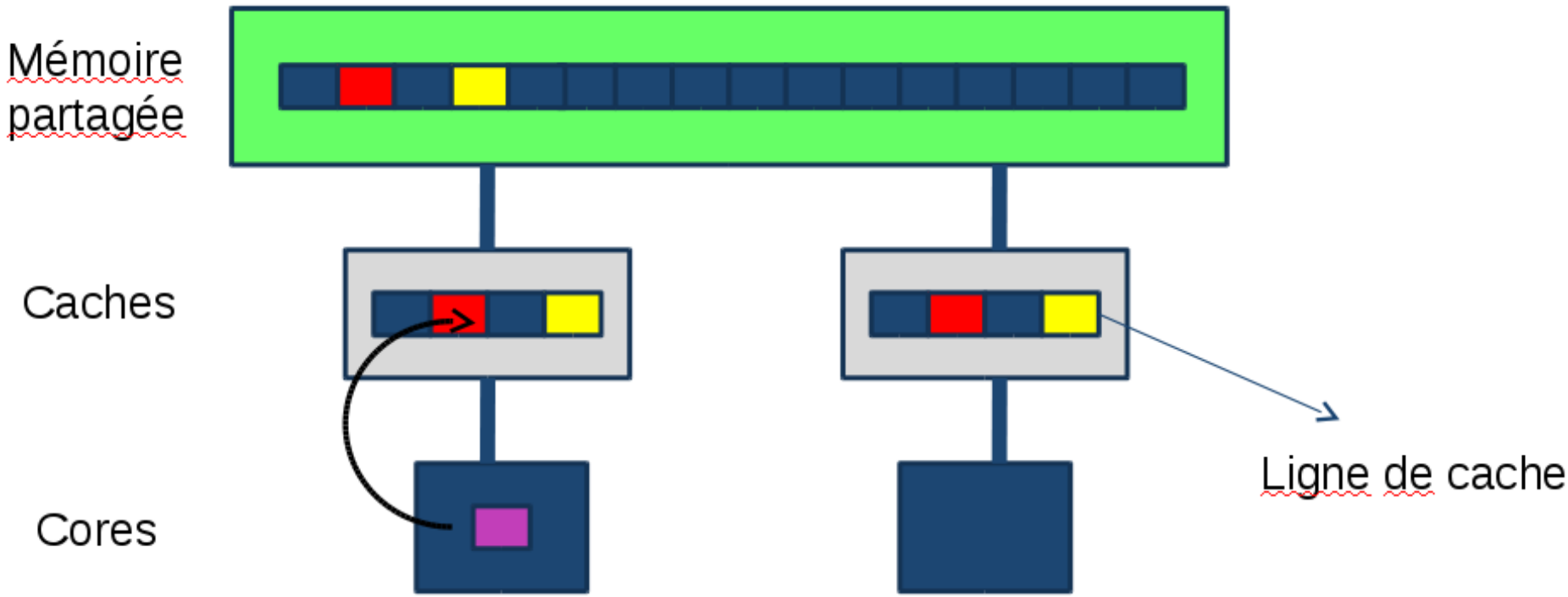
```
#pragma omp parallel for
for(int n=0; n<10; ++n)
 printf("␣%d", n);
printf(".\n");
```

Performance et partage de travail

- Introduire un `PARALLEL FOR` 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 `FOR`
- 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` est plus performant que `CRITICAL`

Performance : les effets du False Sharing

La mise en cohérence des caches et les effets négatifs du “false sharing“ peuvent avoir un fort impact sur les performances



Une opération de chargement d'une ligne de cache partagée invalide les autres copies de cette ligne.

Performance : les effets du False Sharing

- L'utilisation des structures en mémoire partagée peut induire une diminution de performance et une forte limitation de l'extensibilité
  - Pour des raisons de performance, utilisation du cache
  - Si plusieurs processeurs manipulent des données différentes mais adjacentes en mémoire, la mise à jour d'éléments individuels peut provoquer un chargement complet d'une ligne de cache, pour que les caches soient en cohérence avec la mémoire

- Le False sharing dégrade les performances lorsque toutes les conditions suivantes sont réunies
  - Des données partagées sont modifiées sur # cores
  - Plusieurs threads, sur # cores mettent à jour des données qui se trouvent dans la même ligne de cache
  - Ces mises à jour ont lieu très fréquemment et simultanément

Performance : les effets du False Sharing

- Lorsque les données partagées ne sont que lues, cela ne génère pas de false sharing

- En général, le phénomène de false sharing peut être réduit en
  - En privatisant éventuellement des variables
  - Parfois en augmentant la taille des tableaux (taille des problèmes ou augmentation artificielle) ou en faisant du "padding"
  - Parfois en modifiant la façon dont les itérations d'une boucle sont partagées entre les threads (augmenter la taille des paquets)

Performance : les effets du False Sharing

```
int a[nthreads];

#pragma parallel for shared(nthreads,a) schedule(static,1)
for (int i=0; i < nthreads; i++) a[i] = i
```

- **Nthreads** : nombre de thread exécutant la boucle
- Supposons que chaque thread possède un copie de **a** dans son cache local. La taille de paquet de 1 provoque un phénomène de false sharing à chaque mise à jour
- Si une ligne de cache peut contenir C éléments du vecteur **a**, on peut résoudre le problème en étendant artificiellement les dimensions du tableau (“array padding“) : on déclare un tableau **a(C,n)** et on remplace **a(i)** par **a(1,i)**

Performances sur architectures multicoeurs : ordonnancer les threads efficacement

- Maximiser le rendement de chaque CPU est fortement dépendant de la localité des accès mémoire
- Les variables sont stockées dans une zone mémoire (et cache) au moment de leur initialisation
  - paralléliser l'initialisation des éléments (*e.g.*, tableau) de la même façon (même mode et même taille de paquets) que le travail
- L'idéal serait de pouvoir spécifier des contraintes de placement des threads en fonction de affinités threads/mémoire

Performances sur architectures multicoeurs : problématique de la diversité des architectures

- types de processeurs
- nombres de cœurs
- niveaux de cache
- architecture mémoire
- types d'interconnexion des différents composants
- Des stratégies d'optimisation qui peuvent différer d'une configuration à l'autre

Performance : parallélisation conditionnelle

- Utiliser la clause `IF` pour mettre en place une parallélisation conditionnelle
- Exemple : ne paralléliser une boucle que si sa taille est suffisamment grande

OpenMP

- Nécessite une machine multi-processeurs à mémoire partagée
- Mise en œuvre relativement facile, même dans un programme à l'origine séquentiel
- Permet la parallélisation progressive d'un programme séquentiel
- Tout le potentiel des performances parallèles se trouve dans les régions parallèles
- Au sein de ces régions parallèles, le travail peut être partagé grâce aux boucles et aux sections parallèles. Mais on peut aussi singulariser une tâche pour un travail particulier
- Des synchronisations explicites globales ou point à point sont parfois nécessaires dans les régions parallèles
- Un soin tout particulier doit être apporté à la définition du statut des variables utilisées dans une construction

OpenMP versus MPI

- OpenMP exploite la mémoire commune à tous les processus. Toute communication se fait en lisant et en écrivant dans cette mémoire (et en utilisant des mécanismes de synchronisation)
- MPI est une bibliothèques de routines permettant la communication entre différents processus (situés sur différentes machines ou non), les communications se font par des envois ou réceptions explicites de messages
- Pour le programmeur : réécriture du code avec MPI, simple ajout de directives dans le code séquentiel avec OpenMP
- Possibilité de mixer les deux approches dans le cas de cluster de machines avec des nœuds multi-coeurs
- Choix fortement dépendant : de la machine, du code, du temps que veut y consacrer le développeur et du gain recherché
- Extensibilité supérieure avec MPI