Modèles et techniques en programmation parallèle hybride et multi-cœurs

Introduction au parallélisme multithreads

Marc Tajchman

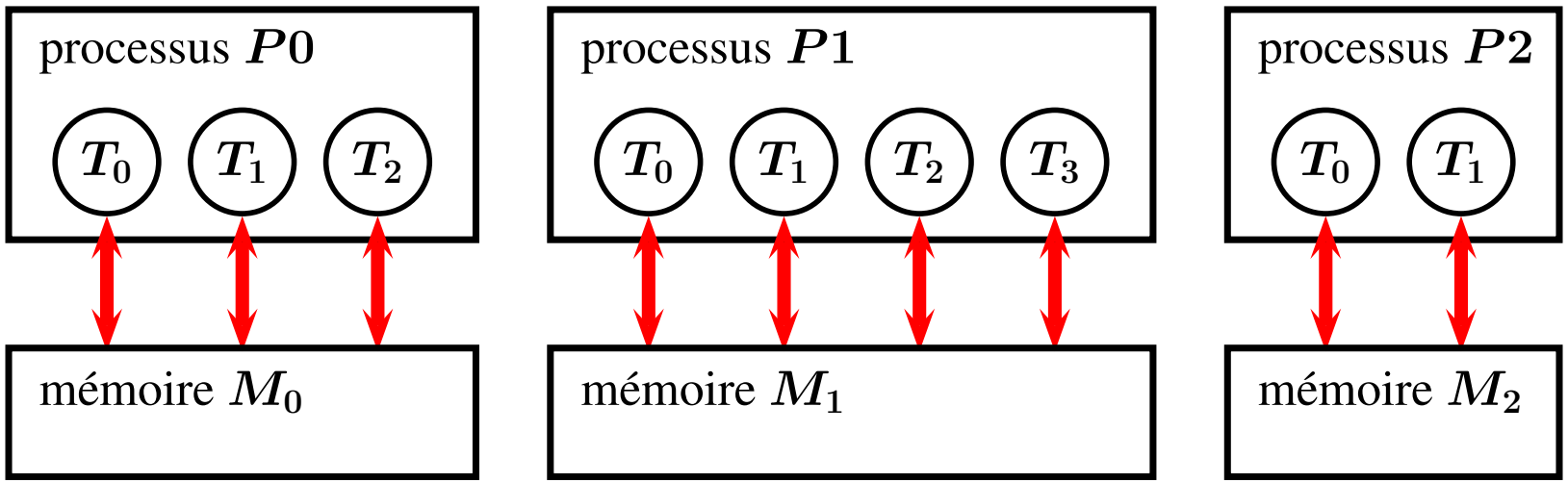
CEA - DEN/DM2S/STMF/LMES

16/11/2023

Notions de processus et de thread

- Un **processus** exécute un ensemble d'instructions en **réservant une partie de la mémoire exclusivement pour lui**.
- Un **thread** (en français: fil d'exécution) est une suite d'instructions exécutées à la suite les unes des autres sur un seul cœur. **Plusieurs threads travaillent souvent dans une zone de mémoire commune.**
- Un thread est toujours contenu dans un processus.
- Un processus contient toujours (au moins) un thread (appelé thread maître ou thread principal ou thread 0). Il peut en contenir plusieurs.
- Un thread n'existe pas seul, il faut d'abord démarrer un processus, et ensuite créer un ou plusieurs threads dans ce processus.

Notions de processus et de thread (2)



Chaque processus à sa mémoire privée et ne peut pas (facilement) accéder à la mémoire privée d'un autre processus. Les threads dans un même processus (T_0 ...) utilisent la mémoire privée du processus qui les contient.

Notions de processus et de thread (3)

Avantages des threads :

- Créer/détruire un thread est plus rapide que créer/détruire un processus.

- Échanger des informations entre threads est plus facile et plus rapide qu'échanger entre processus:
  - un thread peut écrire dans une zone de mémoire commune et un autre pourra lire cette zone plus tard,
  - pour communiquer, deux processus doivent utiliser des moyens plus lents (lire ou écrire des fichiers, utiliser MPI etc.).

MAIS ...

Inconvénients des threads :

- Les threads partagent une partie de la mémoire donc risque de “collisions mémoire” (voir plus loin).
  Alors qu'un processus a sa zone mémoire “cloisonnée” (donc protégée).

- Des threads qui travaillent ensemble doivent se trouver sur le même ordinateur (limitation du nombre de threads au nombre de cœurs) alors que deux processus qui travaillent ensemble peuvent se trouver sur 2 ordinateurs différents.

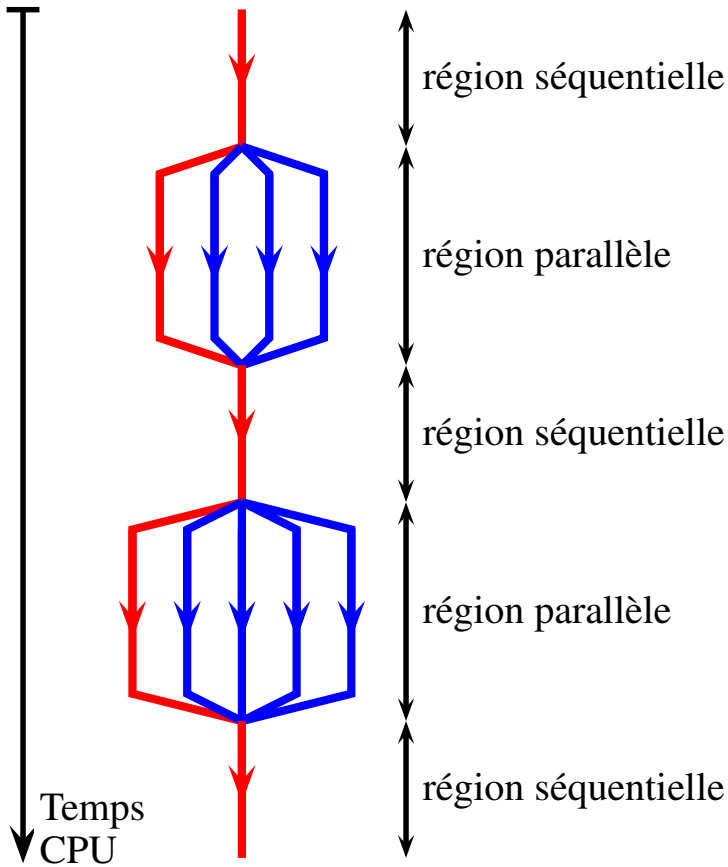
- les threads doivent utiliser une mémoire commune alors que les processus peuvent utiliser des mémoires sur des machines différentes.
  On peut donc résoudre des problèmes de (beaucoup) plus grande taille avec des processus qu'avec des threads.

Parallélisme multi-threads en mémoire partagée

- Le but du parallélisme multi-threads est de découper l'ensemble des instructions en plusieurs parties et d'exécuter (le plus possible) simultanément ces différentes parties par des threads (exécutions) sur des cœurs différents.
- On appellera la version du code non parallélisé : "séquentiel".
- Dans le cas le plus simple, le nombre de threads est égal au nombre de cœurs. Mais on peut utiliser un nombre de threads différent du nombre de cœurs.
- En mémoire partagée signifie que différents groupes d'instructions travaillent sur des données contenues dans la même mémoire. Il faut donc faire attention que les modifications faites par certaines instructions ne perturbent pas les données utilisées par d'autres instructions.

En général, il n'est pas possible/conseillé de rendre "multi-threads" la totalité d'un code. Il sera en général composé d'une succession de parties (ou régions) qui devront rester séquentielles et de parties (ou régions) multi-threadées.

On aura un enchaînement du type:



thread_0 : fil d'exécution rouge, actif durant toute l'exécution

thread_i $(i > 0)$: un des fils d'exécution bleus, actifs seulement dans des parties parallèles

Formellement dans une région parallèle :

On veut exécuter en parallèle, N instructions

$$I = \{I_i(u), i = 1..N\}$$

où u est une structure de données commune.

On regroupe l'ensemble des instructions en n_G groupes

$$G_k = \{I_{i_j}(u), \quad i_j \in (1..N), \quad j = 1, .., N_k\} \quad k = 1..n_G$$

de telle sorte que

$$\begin{aligned} G_{k'} \bigcap G_{k''} &= \varnothing \quad si\ k' \neq k'' \\ \bigcup_{k=1}^{k=n_G} G_k &= I \end{aligned}$$

Autrement dit:

Chaque instruction doit être exécutée une et une seule fois, par un seul thread.

- Autant que possible, il faut pouvoir exécuter les différents groupes d'instructions de façon indépendante (c'est le système qui décidera de démarrer un groupe, en fonction des cœurs disponibles)
- A l'intérieur d'un groupe, les instructions sont exécutées séquentiellement
- Il faut déterminer quelles sont les données partagées entre les groupes et celles qui sont privées (utilisées par un seul groupe).

Ce dernier point est souvent le plus complexe.

Exemple :

On peut découper une boucle:

```
for (i=0; i<N; i++)
  v[i] = f(a, u[i]);
```

en 3 parties (par exemple),
avec $0 = N0 \leq N1 \leq N2 \leq N3 = N$:

```
for (i=N0; i<N1; i++)
  v[i] = f(a, u[i]);
```

1^{er} groupe d'instructions à exécuter par le 1^{er} thread

```
for (i=N1; i<N2; i++)
  v[i] = f(a, u[i]);
```

$2^{ème}$ groupe d'instructions à exécuter par le $2^{ème}$ thread

```
for (i=N2; i<N3; i++)
  v[i] = f(a, u[i]);
```

$3^{ème}$ groupe d'instructions à exécuter par le $3^{ème}$ thread

Il faut examiner l'utilisation de toutes les données sinon on risque de tomber sur des erreurs difficiles à corriger.

Donnée	**Comportement**
N0, N1, N2, N3, a	Utilisées par plusieurs threads mais constantes dans l'algorithme, on peut les partager entre les threads
u	u est constant et donc peut être partagé
v	v varie dans l'algorithme, mais comme chaque thread modifie une partie différente du vecteur, v peut être partagé.
i	i prend des valeurs différentes dans les threads (i = N0 à N1-1 dans le premier thread, i=N1 à N2-1 dans le second thread, etc.), il faut donc utiliser des variables différentes qui représentent i dans les différents threads. i est dit "variable privée".

L'algorithme doit être modifié comme suit:

```
for (i0=N0; i0<N1; i0++)
        v[i0] = f(a, u[i0]);

for (i1=N1; i1<N2; i1++)
        v[i1] = f(a, u[i1]);

for (i2=N2; i2<N3; i2++)
        v[i2] = f(a, u[i2]);
```

Voir plusieurs variantes dans le répertoire Exemples02/Exemple_OpenMP0, codé en OpenMP.

Dans le même exemple, on trouvera la façon habituelle (et beaucoup plus simple) de coder une boucle en OpenMP.

Exemple : si `u` et `v` sont des vecteurs de taille `n > 4`, on veut calculer la boucle `for`:

$$\begin{aligned} &for(i = 1; i < n - 1; i++) \\ &\quad v_i = (u_{i-1} + 2u_i + u_{i+1})/4; \end{aligned} \tag{1}$$

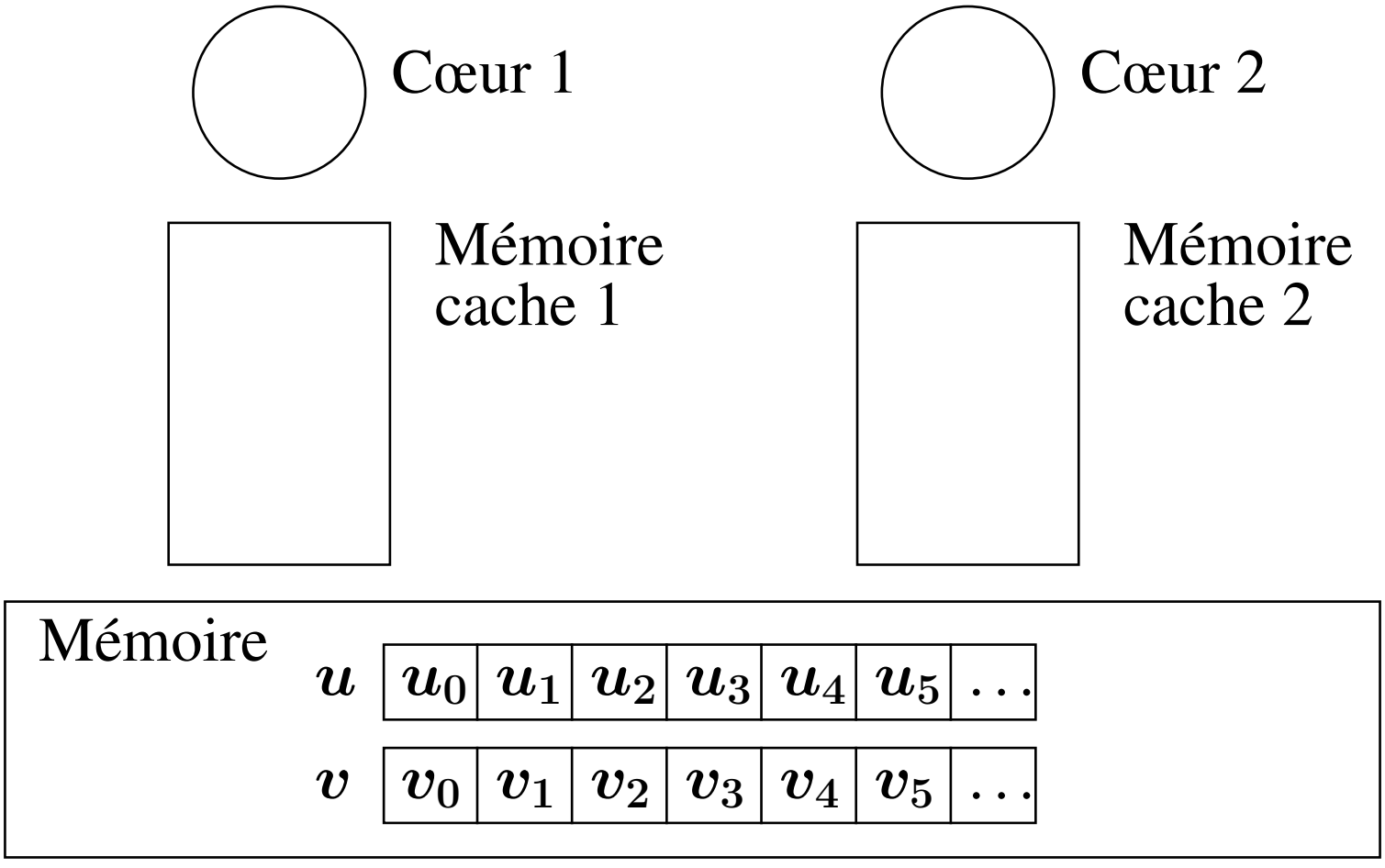
On va examiner en détail le calcul simultané sur 2 cœurs des 2 expressions

$$v_2 = (u_1 + 2u_2 + u_3)/4$$

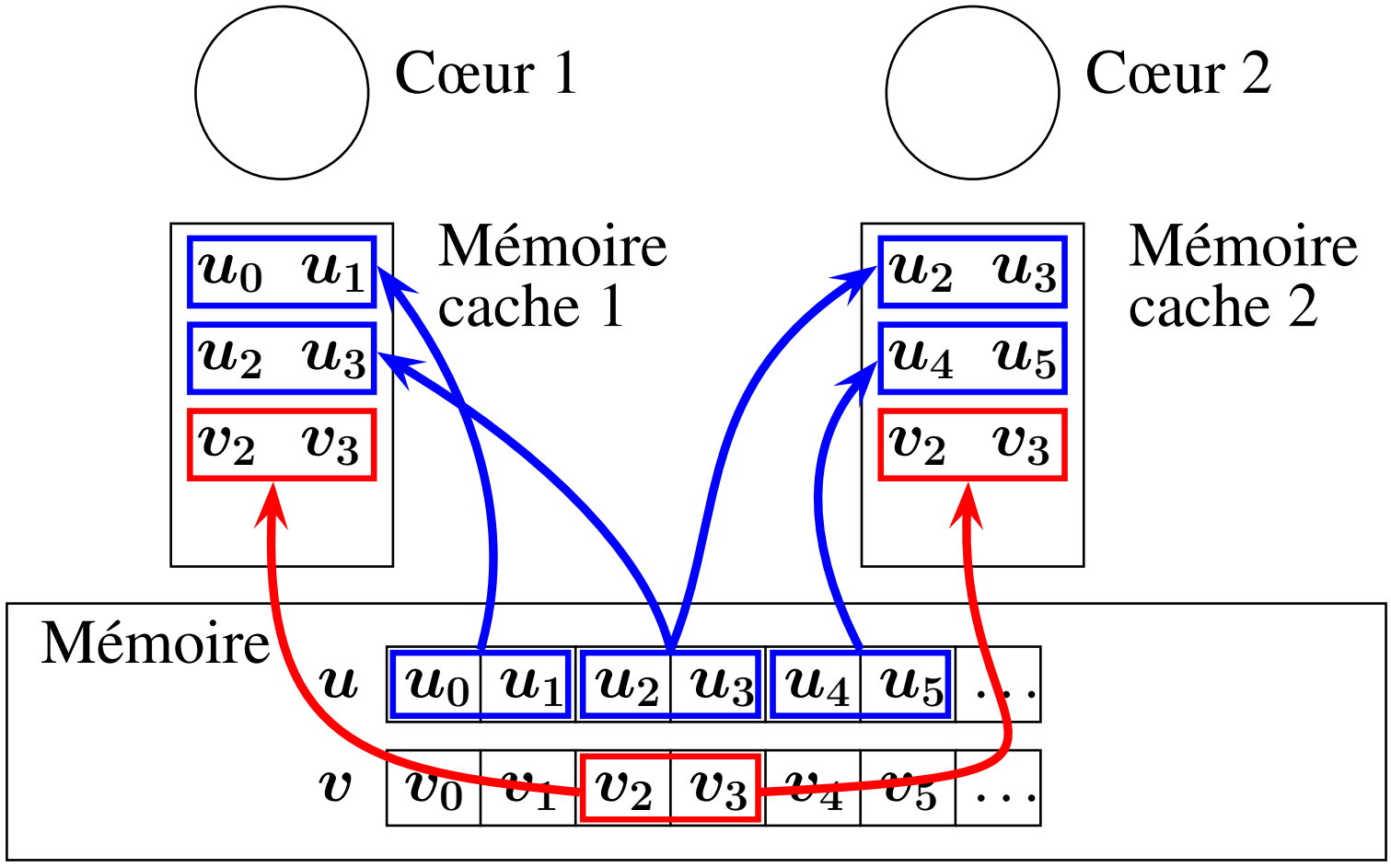
$$v_3 = (u_2 + 2u_3 + u_4)/4$$

On supposera que chaque cœur possède sa mémoire cache de taille 8 (4 lignes de cache de taille 2).

1. Avant d'exécuter les 2 instructions :

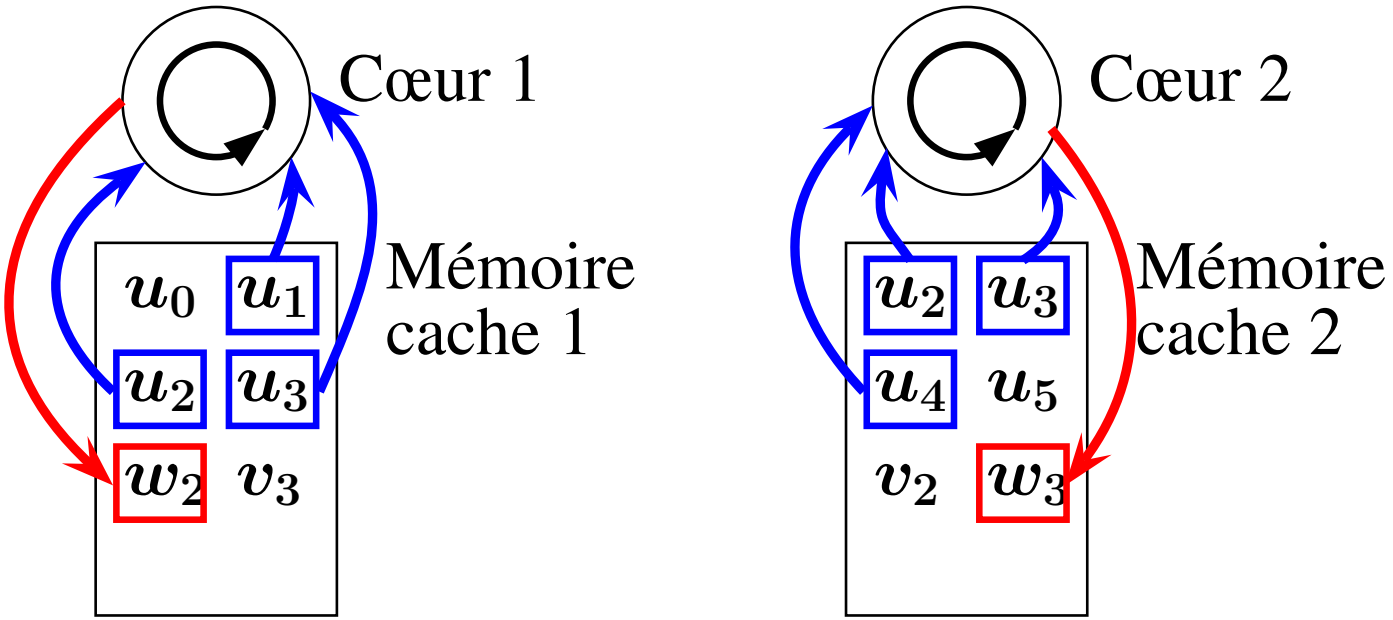


2. Les blocs qui contiennent les composantes utilisées sont recopiés dans les mémoires cache :



3. Les composantes de u sont recopiées dans les mémoires internes des cœurs, les expressions sont calculées et les résultats sont mis à la place de composantes de v:

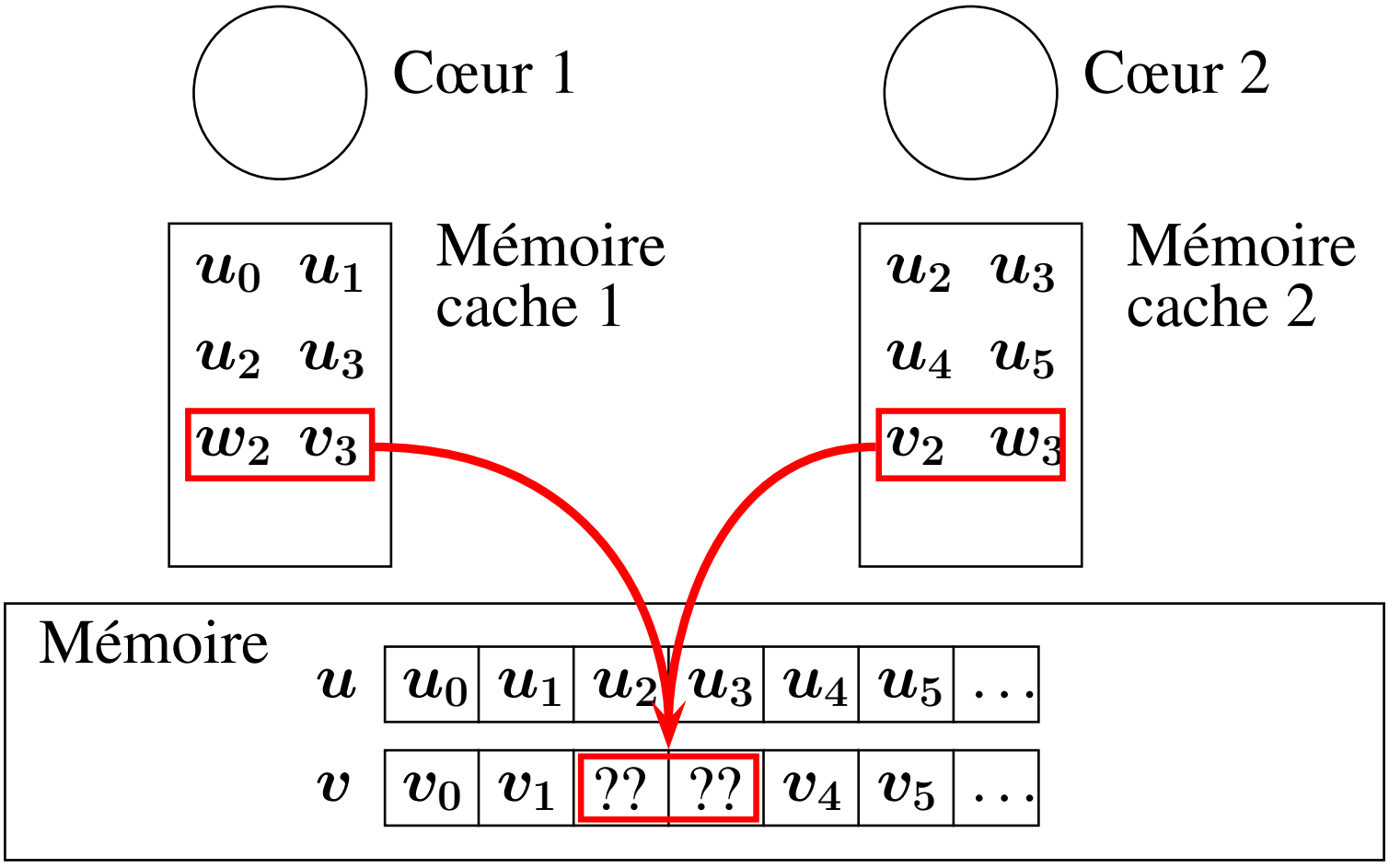
$$w_2 = (u_1 + 2u_2 + u_3)/4 \qquad w_3 = (u_2 + 2u_3 + u_4)/4$$



Mémoire

u	u_0	u_1	u_2	u_3	u_4	u_5	...
v	v_0	v_1	v_2	v_3	v_4	v_5	...

4. Les blocs qui contiennent les résultats sont copiés dans la mémoire centrale. ⚠ Le résultat est indéterminé.

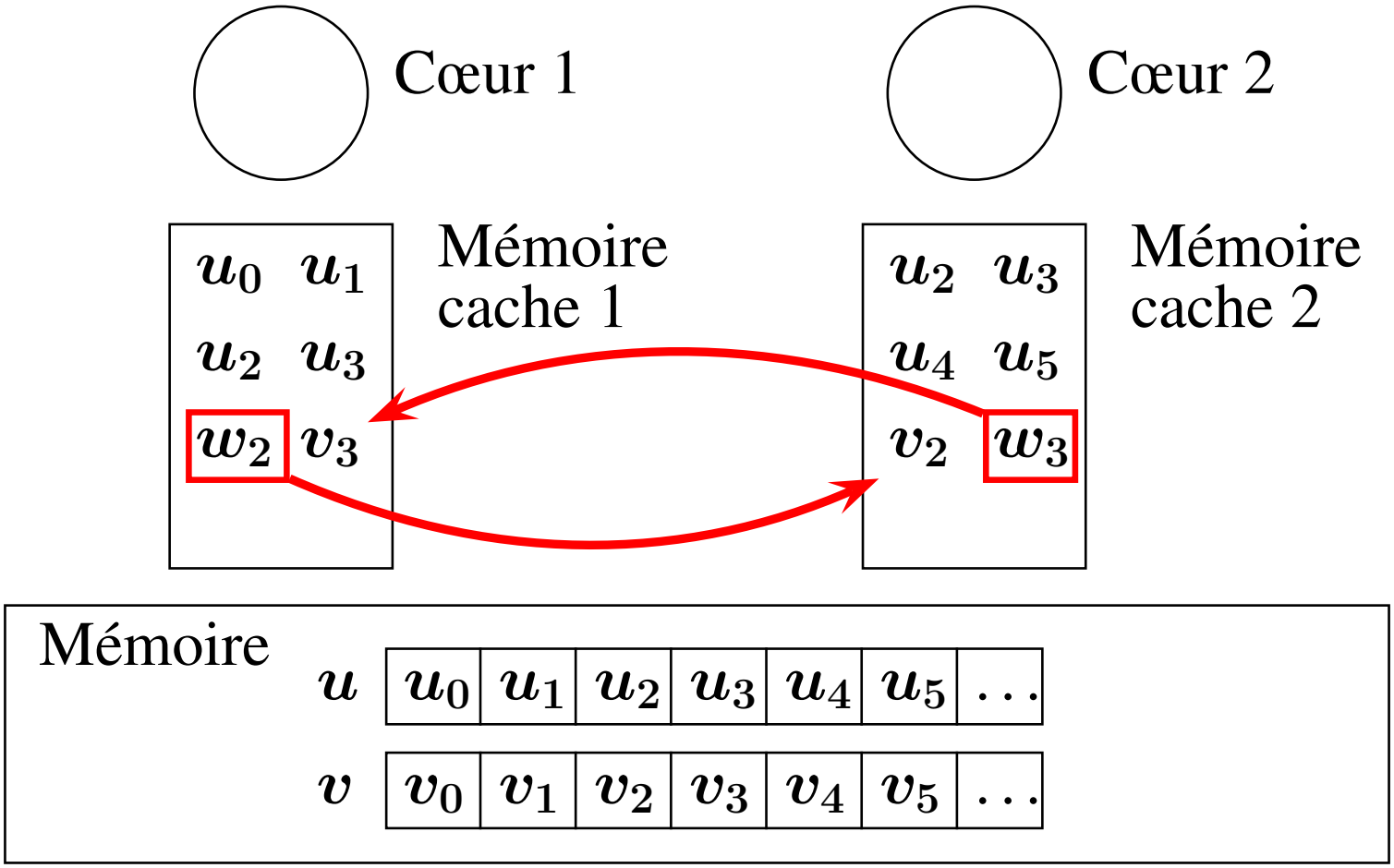


La collision vient du fait que 2 caches différents contiennent chacun un bloc (partiellement) différent qui sera recopié dans la mémoire centrale au même endroit.

Tous les ordinateurs dont les processeurs ont plusieurs cœurs, appliquent un algorithme pour vérifier et rétablir la cohérence de cache :

> *Si deux lignes de cache dans deux mémoires cache correspondent au même emplacement dans la mémoire centrale, on garde les valeurs les plus récentes.*

3 bis. On rétablit la cohérence de cache : w_2 dans le cache 1 est copié à la place de v_2 dans le cache 2 et w_3 dans le cache 2 est copié à la place de v_3 dans le cache 1



Grace à l'étape de cohérence de cache les résultats seront corrects mais le maintien de la cohérence de cache prend du temps et diminue l'efficacité du parallélisme multithreads.

On essaie de faire en sorte que les caches ne contiennent pas des copies des mêmes blocs mémoires (dans ce cas pas besoin de contrôler la cohérence des caches)

D'où la règle supplémentaire :

Règle:

Les zones mémoire modifiées par deux threads différents au même instant doivent être distinctes (sinon les résultats peuvent être faux) et assez éloignées (pour diminuer le cout du maintien de la cohérence de cache)

Exemple: Parcours d'un vecteur u de taille N (multiple de 2) par 2 threads (on suppose que les threads avancent à la même vitesse).
Le parcours suivant :

Thread0			Thread1		
u_0	$=$	$\ldots$	u_1	$=$	$\ldots$
u_2	$=$	$\ldots$	u_3	$=$	$\ldots$
u_4	$=$	$\ldots$	u_5	$=$	$\ldots$
$\ldots$		$\ldots$			
u_{N-2}	$=$	$\ldots$	u_{N-1}	$=$	$\ldots$

est moins performant que le suivant:

Thread0			Thread1		
u_0	$=$	$\ldots$	$u_{N/2}$	$=$	$\ldots$
u_1	$=$	$\ldots$	$u_{N/2+1}$	$=$	$\ldots$
u_2	$=$	$\ldots$	$u_{N/2+2}$	$=$	$\ldots$
$\ldots$			$\ldots$		
$u_{N/2-1}$	$=$	$\ldots$	u_{N-1}	$=$	$\ldots$

Dans le premier parcours, les 2 threads travaillent sur des composantes proches u_k, u_{k+1}

Dans le second parcours, les 2 threads travaillent sur des composantes éloignées (on suppose N grand) : $u_k, u_{k+N/2}$

Donc la cohérence de cache sera plus coûteuse pour le premier parcours.

Rappels sur OpenMP

Le principal (mais pas le seul) outil pour coder du parallélisme multithreads est OpenMP.

L'exposé ici ne présente que les aspects de base de OpenMP. Pour ceux qui veulent réellement utiliser OpenMP dans leurs codes, on conseille l'utilisation de la documentation OpenMP.

Par exemple:

Tim Mattson (Intel) "Introduction to OpenMP"

https://www.openmp.org/wp-content/uploads/Intro_To_OpenMP_Mattson.pdf

Formation "OpenMP" de l'IDRIS (CNRS)

http://www.idris.fr/media/formations/openmp/idris_openmp_cours-v23.03.pdf

OpenMP API 5.1 Specification

https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

Principe : on ajoute dans le code, des “pragma” (ligne qui commence par `#pragma`)

Par exemple:

```
#pragma omp parallel for

#pragma omp section

#pragma omp critical

#pragma omp master
```

Si on compile le code sans l'option de compilation OpenMP, les pragmas seront ignorés, donc on peut souvent utiliser le même code source pour un exécutable séquentiel et multithreads.

OpenMP propose aussi quelques fonctions pour gérer le nombre de threads (`omp_get_num_threads()`, `omp_set_num_threads()`) et pour connaître le numéro de thread courant (`omp_get_thread_num()`).

Ces fonctions ne sont utilisables que dans un code compilé avec OpenMP.
Pour garder un seul code, on peut utiliser la macro _OPENMP qui vaux vrai si on compile avec OpenMP et faux sinon:

```
#ifdef _OPENMP
// code parallele
#else
// code sequentiel
#endif
```

Avantages:

- un seul code source pour les versions parallèle ou séquentiel
- on peut choisir les boucles qu'on veut paralléliser ou non (parallélisation incrémentale)
- facile à coder

Désavantages et difficultés:

- les performances sont parfois décevantes
- attention aux variables partagées par différentes itérations d'une boucle
- pas beaucoup de contrôle sur le placement des threads et sur le découpage en sous-boucles

Pour compiler du code utilisant OpenMP, on utilise une option de compilation (qui dépend du compilateur : -fopenmp pour gcc/g++/gfortran).

Pour exécuter un code utilisant plusieurs threads, il y a plusieurs possibilités:

- définir une variable d'environnement `OMP_NUM_THREADS`, par exemple :

```
OMP_NUM_THREADS=5 ./code.exe
```

- appeler dans le code source la fonction

```
omp_set_num_threads(5);
```

Exemple OpenMP 1

```
#pragma omp parallel
{
  std::cout << "Bonjour" << std::endl;
}
```

- avant d'arriver au pragma, on est dans la partie séquentielle (seul le thread 0 est actif);
- quand l'exécution arrive sur la ligne `#pragma`, le système crée (ou active) $N-1$ threads (N est égal au nombre de cœurs ou à la valeur de `OMP_NUM_THREADS` si cette variable d'environnement existe);
- chacun des N threads exécute la partie parallèle (le bloc d'instructions contenu entre les accolades);
- quand tous les threads sont arrivés à la fin du bloc, les threads s'arrêtent (ou se mettent en pause) sauf le thread 0.

Le code ci-dessus est contenu dans le répertoire Exemples2/Exemple_OpenMP1.
Lire le fichier README.pdf pour compiler et exécuter le code.

Relancer plusieurs fois l'exécution, expliquer ce qui est affiché.

Exemple OpenMP 2 : variables partagées et privées

```
std::string prefix = "Ici␣le␣thread␣";
int iTh;

#pragma omp parallel
{
#ifdef _OPENMP
 iTh = omp_get_thread_num();
#else
 iTh = 0;
#endif
 std::cout<<prefix<<iTh<<std::endl;
}
```

Remarque: la fonction standard `omp_get_thread_num` n'est pas définie si on ne compile pas en mode OpenMP, il faut donc la protéger avec la macro `_OPENMP`

Dans une région parallèle, il faut déterminer la nature des variables:

- `prefix` est une chaîne de caractères constante dans la région parallèle, c'est donc une variable partagée entre les threads
- `iTh` est un entier qui a une valeur différente dans la région parallèle suivant les threads, c'est donc une variable privée par thread

Dans la pragma, on liste les variables privées et partagées :

```
#pragma omp parallel shared(v1, v2) private(w1, w2)
```

Si on ne le fait pas, les variables définies dans la région séquentielle et modifiées dans la région parallèle auront une valeur indéterminée.

Souvent les variables sont partagées par défaut, mais on conseille de l'écrire explicitement.

Exemple corrigé:

```
std::string prefix = "Ici␣le␣thread␣";
int iTh;

#pragma omp parallel shared(prefix)
    private(iTh)
{
#ifdef _OPENMP
  iTh = omp_get_thread_num();
#else
  iTh = 0;
#endif
  std::cout<< prefix << iTh <<std::endl;
}
```

Exemple corrigé (seconde version): on définit `iTh` à l'intérieur de la région parallèle (chaque région parallèle crée sa propre copie de `iTh`)

```
std::string prefix = "Ici le thread ";

#pragma omp parallel shared(prefix)
{
  int iTh;
#ifdef _OPENMP
  iTh = omp_get_thread_num();
#else
  iTh = 0;
#endif
  std::cout << prefix << iTh <<std::endl;
}
```

Le code ci-dessus est contenu dans le répertoire Exemples2/Exemple_OpenMP2.
Lire le fichier README.pdf pour compiler et exécuter le code.

Relancer plusieurs fois l'exécution, expliquer ce qui est affiché.

Exemple OpenMP 3

Soient u, v et w des vecteurs de taille n, a et b des scalaires, le code ci-dessous calcule $w = au + bv$

```
for (i=0; i<n; i++)
  w[i] = a*u[i] + b*v[i];
```

Pour découper la boucle en T boucles partielles (où T est le nombre de threads), OpenMP propose de le faire lui-même:

```
#pragma omp parallel \
  shared(u,v,w,a,b,n) private(i)
{
#pragma omp for
  for (i=0; i<n; i++)
    w[i] = a*u[i] + b*v[i];
}
```

ou plus simplement:

```
#pragma omp parallel for \
  shared(u,v,w,a,b,n) private(i)
  for (i=0; i<n; i++)
    w[i] = a*u[i] + b*v[i];
```

ou encore, si les variables sont partagées par défaut:

```
#pragma omp parallel for
  for (i=0; i<n; i++)
    w[i] = a*u[i] + b*v[i];
```

L'indice de boucle qui suit immédiatement le pragma openmp for est implicitement privé

Le code ci-dessus est contenu dans le répertoire
Exemples2/Exemple_OpenMP3.
Lire le fichier README.pdf pour compiler et exécuter le code.

Exemple OpenMP 4

Soient A, B et C des matrices de taille $n \times m$, a et b des scalaires, le code ci-dessous calcule $C = aA + bB$

```
for (i=0; i<n; i++)
  for (j=0; j<m; j++)
    C(i,j) = a*A(i,j) + b*B(i,j);
```

Il y a deux boucles, on a le choix de découper

- la boucle externe sur i
- la boucle interne sur j
- les 2 boucles

Parallélisation sur la boucle externe :

```
#pragma omp parallel for \
    default(shared) private(j)
  for (i=0; i<n; i++)
    for (j=0; j<m; j++)
      C(i,j) = a*A(i,j) + b*B(i,j);
```

Ne pas oublier que seul l'indice *i* de boucle qui est découpée est implicitement privé.

L'indice *j* de la boucle interne doit être privé pour que les résultats soient corrects.

Parallélisation sur la boucle interne :

```
  for (i=0; i<n; i++)
  {
#pragma omp parallel for default(shared)
    for (j=0; j<m; j++)
      C(i,j) = a*A(i,j) + b*B(i,j);
  }
```

Il y a n régions parallèles (pour chaque itération de la boucle externe).

L'indice de boucle externe i est constant dans les régions parallèles, donc il peut rester partagé.

L'indice de boucle interne j est implicitement privé.

Parallélisation sur les 2 boucles :

```
#pragma omp parallel for \
    default(shared) collapse(2)
for (i=0; i<n; i++)
  for (j=0; j<m; j++)
    C(i,j) = a*A(i,j) + b*B(i,j);
```

Il y a 1 région parallèle.

Les 2 indices de boucle i et j sont implicitement privés.

Cela ne fonctionne que si les 2 instructions `for` se suivent sans instruction intermédiaire.

Le code ci-dessus est contenu dans le répertoire
Exemples2/Exemple_OpenMP4.
Lire le fichier README.pdf pour compiler et exécuter le code.

L'exemple OpenMP 4 suggère les règles suivantes pour améliorer l'efficacité de la parallélisation multithreads:

Règle :

La quantité d'instructions dans les régions parallèles doit être la plus grande possible par rapport aux instructions dans les régions séquentielles (c'est l'application de la loi d'Amdahl).

Règle :

Le nombre de régions parallèles doit être le plus petit possible.

Règle :

Pour une région parallèle donnée, le nombre de threads qui calcule cette région, doit être déterminé de façon "optimale".

Le nombre de threads optimal pour calculer une région parallèle est difficile à définir dans l'absolu.

Cela dépend du temps mis par le système pour gérer les threads (création, activation, désactivation), pour maintenir la cohérence de cache, etc.

Il faut que le temps calcul à l'intérieur du découpage de la région parallèle soit beaucoup plus grand que le temps de gestion multithreads.

Cette règle a une signification similaire avec le ratio calculs/communications pour un code MPI.

Dans TBB (une des librairies multithreads) par exemple, la documentation conseille de commencer avec un découpage ou chaque thread passe $\sim 10^5$ cycles dans la région parallèle.

Le plus raisonnable est de tester : augmenter le nombre de threads (inférieur ou égal au nombre de cœurs) tant que la parallélisation améliore le temps calcul.

Exemple OpenMP 5

Soit v un vecteur de taille n.

On calcule la moyenne et la variance de v par le code suivant

```
double s, s2;
for (i=0; i<n; i++) {
  s += v[i];
  s2 += v[i]*v[i];
}
moy = s/n;
var = s2/n - moy*moy;
```

Si on divise l'ensemble de itérations en paquets, il faudra faire les sommes de v_i et v_i^2 sur chaque paquet puis combiner ces sommes partielles pour obtenir les sommes sur l'ensemble des itérations.

Une première version parallélisée serait

```
double s, s_partiel[nThreads];
double s2, s2_partiel[nThreads];
int iTh;
#pragma omp parallel for \
    default(shared) private(iTh)
for (i=0; i<n; i++) {
  iTh = omp_get_thread_num();
  s_partiel[iTh] += v[i];
  s2_partiel[iTh] += v[i]*v[i];
}

s = 0.0; s2 = 0.0;
for (iTh=0; iTh < nThreads; iTh++)
{
  s += s_partiel[iTh];
  s2 += s2_partiel[iTh];
}
moy=s/n; var=s2/n-moy*moy;
```

Cette version est très fortement pénalisée par une mauvaise utilisation de la mémoire cache.
On peut introduire un décalage entre les positions mémoires des vecteurs sommes partielles pour améliorer son comportement.

Une seconde version utilise les section critiques OpenMP (ou mieux les sections “atomiques”)

```
double s = 0, s_partiel;
double s2 = 0, s2_partiel;

#pragma omp parallel default(shared) \
        private(s_partiel, s2_partiel)
{
 s_partiel = 0.0;s2_partiel = 0.0;
#pragma omp for
 for (i=0; i<n; i++) {
   s_partiel+=v[i];s2_partiel+=v[i]*v[i];
 }
#pragma critical
 {
   s+=s_partiel;s2+=s2_partiel;
 }
}
moy=s/n; var=s2/n-moy*moy;
```

Une section critique protège des instructions qui ne peuvent pas être exécutées par plusieurs threads au même moment.

Quand plusieurs threads arrivent au début de la section critique, ils rentrent dans la section critique l'un après l'autre.

Donc une section critique diminue le parallélisme du code mais permet d'éviter des collisions mémoire (plusieurs threads veulent mettre à jour les mêmes variables)

Une dernière version utilise le mécanisme de réduction proposé par OpenMP:

```
double s = 0 ;
double s2 = 0;

# pragma omp parallel for \
      default(shared) reduction(+: s, s2)
for (i=0; i<n; i++) {
  s += u[i];
  s2 += u[i]*u[i];
}
moy = s / n ;
var = s2 /n - moy * moy ;
```

Quelques autres pragmas

`#pragma omp barrier`

Dans une région parallèle, les threads avancent indépendamment les uns des autres.
Il faut parfois définir des barrières intermédiaires : si certains threads atteignent une position dans le code avant les autres, ils attendent les autres threads.
Les pragma `omp parallel` *et* `omp for` *sont suivis d'une barrière implicite.*

`#pragma omp single`

Définit une sous-région dans une région parallèle. Le premier thread qui atteint le début de la sous-région, exécute les instructions qui sont dedans.
Les autres threads sautent au-dessus de cette sous-région.
Il y a une barrière implicite à la fin de la sous-région.

`#pragma omp master`

Définit une sous-région dans une région parallèle. Seul le thread 0 peut entrer dans la sous-région, et exécuter les instructions qui sont dedans.
Les autres threads sautent au-dessus de cette sous-région.
Il y a pas de barrière implicite à la fin de la sous-région.